You Don't Know JS: Scope & Closures

Chapter 1: What is Scope?

One of the most fundamental paradigms of nearly all programming languages is the ability to store values in variables, and later retrieve or modify those values. In fact, the ability to store values and pull values out of variables is what gives a program *state*.

Without such a concept, a program could perform some tasks, but they would be extremely limited and not terribly interesting.

But the inclusion of variables into our program begets the most interesting questions we will now address: where do those variables *live*? In other words, where are they stored? And, most importantly, how does our program find them when it needs them?

These questions speak to the need for a well-defined set of rules for storing variables in some location, and for finding those variables at a later time. We'll call that set of rules: *Scope*.

But, where and how do these *Scope* rules get set?

Compiler Theory

It may be self-evident, or it may be surprising, depending on your level of interaction with various languages, but despite the fact that JavaScript falls under the general category of "dynamic" or "interpreted" languages, it is in fact a compiled language. It is *not* compiled well in advance, as are many traditionally-compiled languages, nor are the results of compilation portable among various distributed systems.

But, nevertheless, the JavaScript engine performs many of the same steps, albeit in more sophisticated ways than we may commonly be aware, of any traditional language-compiler.

In a traditional compiled-language process, a chunk of source code, your program, will undergo typically three steps *before* it is executed, roughly called "compilation":

1. **Tokenizing/Lexing:** breaking up a string of characters into meaningful (to the language) chunks, called tokens. For instance, consider the program: `var a = 2;`. This program would likely be broken up into the following tokens: `var`, `a`, `=`, `2`, and `;`. Whitespace may or may not be persisted as a token, depending on whether it's meaningful or not.

   Note: The difference between tokenizing and lexing is subtle and academic, but it centers on whether or not these tokens are identified in a *stateless*

or *stateful* way. Put simply, if the tokenizer were to invoke stateful parsing rules to figure out whether `a` should be considered a distinct token or just part of another token, *that* would be **lexing**.

2. **Parsing:** taking a stream (array) of tokens and turning it into a tree of nested elements, which collectively represent the grammatical structure of the program. This tree is called an "AST" (Abstract Syntax Tree).

   The tree for `var a = 2;` might start with a top-level node called `VariableDeclaration`, with a child node called `Identifier` (whose value is `a`), and another child called `AssignmentExpression` which itself has a child called `NumericLiteral` (whose value is `2`).

3. **Code-Generation:** the process of taking an AST and turning it into executable code. This part varies greatly depending on the language, the platform it's targeting, etc.

   So, rather than get mired in details, we'll just handwave and say that there's a way to take our above described AST for `var a = 2;` and turn it into a set of machine instructions to actually *create* a variable called `a` (including reserving memory, etc.), and then store a value into `a`.

   Note: The details of how the engine manages system resources are deeper than we will dig, so we'll just take it for granted that the engine is able to create and store variables as needed.

The JavaScript engine is vastly more complex than *just* those three steps, as are most other language compilers. For instance, in the process of parsing and code-generation, there are certainly steps to optimize the performance of the execution, including collapsing redundant elements, etc.

So, I'm painting only with broad strokes here. But I think you'll see shortly why *these* details we *do* cover, even at a high level, are relevant.

For one thing, JavaScript engines don't get the luxury (like other language compilers) of having plenty of time to optimize, because JavaScript compilation doesn't happen in a build step ahead of time, as with other languages.

For JavaScript, the compilation that occurs happens, in many cases, mere microseconds (or less!) before the code is executed. To ensure the fastest performance, JS engines use all kinds of tricks (like JITs, which lazy compile and even hot re-compile, etc.) which are well beyond the "scope" of our discussion here.

Let's just say, for simplicity's sake, that any snippet of JavaScript has to be compiled before (usually *right* before!) it's executed. So, the JS compiler will take the program `var a = 2;` and compile it *first*, and then be ready to execute it, usually right away.

Understanding Scope

The way we will approach learning about scope is to think of the process in terms of a conversation. But, *who* is having the conversation?

The Cast

Let's meet the cast of characters that interact to process the program `var a = 2;`, so we understand their conversations that we'll listen in on shortly:

1. *Engine*: responsible for start-to-finish compilation and execution of our JavaScript program.
2. *Compiler*: one of *Engine*'s friends; handles all the dirty work of parsing and code-generation (see previous section).
3. *Scope*: another friend of *Engine*; collects and maintains a look-up list of all the declared identifiers (variables), and enforces a strict set of rules as to how these are accessible to currently executing code.

For you to *fully understand* how JavaScript works, you need to begin to *think* like *Engine* (and friends) think, ask the questions they ask, and answer those questions the same.

Back & Forth

When you see the program `var a = 2;`, you most likely think of that as one statement. But that's not how our new friend *Engine* sees it. In fact, *Engine* sees two distinct statements, one which *Compiler* will handle during compilation, and one which *Engine* will handle during execution.

So, let's break down how *Engine* and friends will approach the program `var a = 2;`.

The first thing *Compiler* will do with this program is perform lexing to break it down into tokens, which it will then parse into a tree. But when *Compiler* gets to code-generation, it will treat this program somewhat differently than perhaps assumed.

A reasonable assumption would be that *Compiler* will produce code that could be summed up by this pseudo-code: "Allocate memory for a variable, label it `a`, then stick the value `2` into that variable." Unfortunately, that's not quite accurate.

Compiler will instead proceed as:

1. Encountering `var a`, *Compiler* asks *Scope* to see if a variable `a` already exists for that particular scope collection. If so, *Compiler* ignores this

declaration and moves on. Otherwise, *Compiler* asks *Scope* to declare a new variable called `a` for that scope collection.

2. *Compiler* then produces code for *Engine* to later execute, to handle the `a = 2` assignment. The code *Engine* runs will first ask *Scope* if there is a variable called `a` accessible in the current scope collection. If so, *Engine* uses that variable. If not, *Engine* looks *elsewhere* (see nested *Scope* section below).

If *Engine* eventually finds a variable, it assigns the value `2` to it. If not, *Engine* will raise its hand and yell out an error!

To summarize: two distinct actions are taken for a variable assignment: First, *Compiler* declares a variable (if not previously declared in the current scope), and second, when executing, *Engine* looks up the variable in *Scope* and assigns to it, if found.

Compiler Speak

We need a little bit more compiler terminology to proceed further with understanding.

When *Engine* executes the code that *Compiler* produced for step (2), it has to look-up the variable `a` to see if it has been declared, and this look-up is consulting *Scope*. But the type of look-up *Engine* performs affects the outcome of the look-up.

In our case, it is said that *Engine* would be performing an "LHS" look-up for the variable `a`. The other type of look-up is called "RHS".

I bet you can guess what the "L" and "R" mean. These terms stand for "Left-hand Side" and "Right-hand Side".

Side... of what? **Of an assignment operation.**

In other words, an LHS look-up is done when a variable appears on the left-hand side of an assignment operation, and an RHS look-up is done when a variable appears on the right-hand side of an assignment operation.

Actually, let's be a little more precise. An RHS look-up is indistinguishable, for our purposes, from simply a look-up of the value of some variable, whereas the LHS look-up is trying to find the variable container itself, so that it can assign. In this way, RHS doesn't *really* mean "right-hand side of an assignment" per se, it just, more accurately, means "not left-hand side".

Being slightly glib for a moment, you could also think "RHS" instead means "retrieve his/her source (value)", implying that RHS means "go get the value of...".

Let's dig into that deeper.

When I say:

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

The reference to `a` is an RHS reference, because nothing is being assigned to `a` here. Instead, we're looking-up to retrieve the value of `a`, so that the value can be passed to `console.log(..)`.

By contrast:

```
a = 2;
```

The reference to `a` here is an LHS reference, because we don't actually care what the current value is, we simply want to find the variable as a target for the `= 2` assignment operation.

Note: LHS and RHS meaning "left/right-hand side of an assignment" doesn't necessarily literally mean "left/right side of the `=` assignment operator". There are several other ways that assignments happen, and so it's better to conceptually think about it as: "who's the target of the assignment (LHS)" and "who's the source of the assignment (RHS)".

Consider this program, which has both LHS and RHS references:

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

foo( 2 );
```

The last line that invokes `foo(..)` as a function call requires an RHS reference to `foo`, meaning, "go look-up the value of `foo`, and give it to me." Moreover, `(..)` means the value of `foo` should be executed, so it'd better actually be a function!

There's a subtle but important assignment here. **Did you spot it?**

You may have missed the implied `a = 2` in this code snippet. It happens when the value `2` is passed as an argument to the `foo(..)` function, in which case the `2` value is **assigned** to the parameter `a`. To (implicitly) assign to parameter `a`, an LHS look-up is performed.

There's also an RHS reference for the value of `a`, and that resulting value is passed to `console.log(..)`. `console.log(..)` needs a reference to execute. It's an RHS look-up for the `console` object, then a property-resolution occurs to see if it has a method called `log`.

Finally, we can conceptualize that there's an LHS/RHS exchange of passing the value `2` (by way of variable `a`'s RHS look-up) into `log(..)`. Inside of the native implementation of `log(..)`, we can assume it has parameters, the first of which (perhaps called `arg1`) has an LHS reference look-up, before assigning `2` to it.

Note: You might be tempted to conceptualize the function declaration `function foo(a) {...` as a normal variable declaration and assignment, such as `var foo` and `foo = function(a){...`. In so doing, it would be tempting to think of this function declaration as involving an LHS look-up.

However, the subtle but important difference is that *Compiler* handles both the declaration and the value definition during code-generation, such that when *Engine* is executing code, there's no processing necessary to "assign" a function value to `foo`. Thus, it's not really appropriate to think of a function declaration as an LHS look-up assignment in the way we're discussing them here.

Engine/Scope Conversation

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

foo( 2 );
```

Let's imagine the above exchange (which processes this code snippet) as a conversation. The conversation would go a little something like this:

> ***Engine***: Hey *Scope*, I have an RHS reference for `foo`. Ever heard of it?
>
> ***Scope***: Why yes, I have. *Compiler* declared it just a second ago. He's a function. Here you go.
>
> ***Engine***: Great, thanks! OK, I'm executing `foo`.
>
> ***Engine***: Hey, *Scope*, I've got an LHS reference for `a`, ever heard of it?
>
> ***Scope***: Why yes, I have. *Compiler* declared it as a formal parameter to `foo` just recently. Here you go.
>
> ***Engine***: Helpful as always, *Scope*. Thanks again. Now, time to assign `2` to `a`.
>
> ***Engine***: Hey, *Scope*, sorry to bother you again. I need an RHS look-up for `console`. Ever heard of it?
>
> ***Scope***: No problem, *Engine*, this is what I do all day. Yes, I've got `console`. He's built-in. Here ya go.
>
> ***Engine***: Perfect. Looking up `log(..)`. OK, great, it's a function.
>
> ***Engine***: Yo, *Scope*. Can you help me out with an RHS reference to `a`. I think I remember it, but just want to double-check.
>
> ***Scope***: You're right, *Engine*. Same guy, hasn't changed. Here ya go.

> ***Engine***: Cool. Passing the value of `a`, which is `2`, into `log(..)`.
>
> ...

Quiz

Check your understanding so far. Make sure to play the part of *Engine* and have a "conversation" with the *Scope*:

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. Identify all the LHS look-ups (there are 3!).
2. Identify all the RHS look-ups (there are 4!).

Note: See the chapter review for the quiz answers!

Nested Scope

We said that *Scope* is a set of rules for looking up variables by their identifier name. There's usually more than one *Scope* to consider, however.

Just as a block or function is nested inside another block or function, scopes are nested inside other scopes. So, if a variable cannot be found in the immediate scope, *Engine* consults the next outer containing scope, continuing until found or until the outermost (aka, global) scope has been reached.

Consider:

```
function foo(a) {
    console.log( a + b );
}

var b = 2;

foo( 2 ); // 4
```

The RHS reference for `b` cannot be resolved inside the function `foo`, but it can be resolved in the *Scope* surrounding it (in this case, the global).

So, revisiting the conversations between *Engine* and *Scope*, we'd overhear:

> ***Engine***: "Hey, *Scope* of `foo`, ever heard of `b`? Got an RHS reference for it."

Scope: "Nope, never heard of it. Go fish."

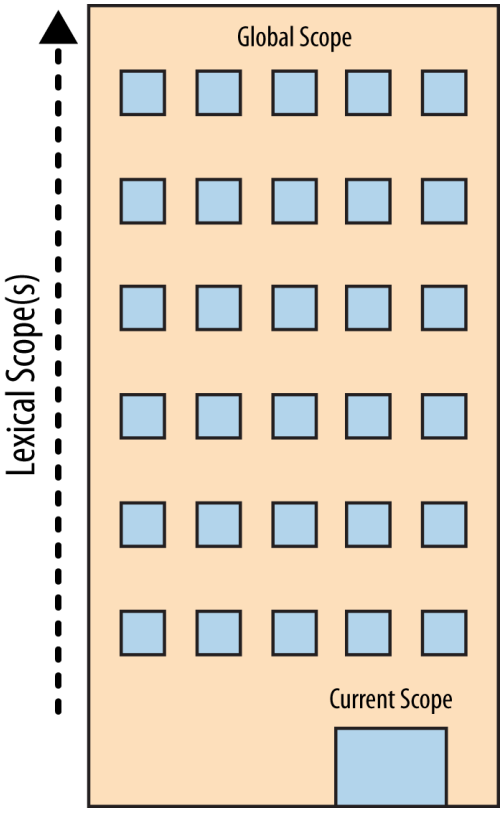
Engine: "Hey, *Scope* outside of `foo`, oh you're the global *Scope*, ok cool. Ever heard of `b`? Got an RHS reference for it."

Scope: "Yep, sure have. Here ya go."

The simple rules for traversing nested *Scope*: *Engine* starts at the currently executing *Scope*, looks for the variable there, then if not found, keeps going up one level, and so on. If the outermost global scope is reached, the search stops, whether it finds the variable or not.

Building on Metaphors

To visualize the process of nested *Scope* resolution, I want you to think of this tall building.



The building represents our program's nested *Scope* rule set. The first floor of the building represents your currently executing *Scope*, wherever you are. The top level of the building is the global *Scope*.

You resolve LHS and RHS references by looking on your current floor, and if you don't find it, taking the elevator to the next floor, looking there, then the next, and so on. Once you get to the top floor (the global *Scope*), you either find what you're looking for, or you don't. But you have to stop regardless.

Errors

Why does it matter whether we call it LHS or RHS?

Because these two types of look-ups behave differently in the circumstance where the variable has not yet been declared (is not found in any consulted *Scope*).

Consider:

```
function foo(a) {
    console.log( a + b );
    b = a;
}

foo( 2 );
```

When the RHS look-up occurs for `b` the first time, it will not be found. This is said to be an "undeclared" variable, because it is not found in the scope.

If an RHS look-up fails to ever find a variable, anywhere in the nested *Scope*s, this results in a `ReferenceError` being thrown by the *Engine*. It's important to note that the error is of the type `ReferenceError`.

By contrast, if the *Engine* is performing an LHS look-up and arrives at the top floor (global *Scope*) without finding it, and if the program is not running in "Strict Mode" [1], then the global *Scope* will create a new variable of that name **in the global scope**, and hand it back to *Engine*.

"No, there wasn't one before, but I was helpful and created one for you."

"Strict Mode" [2], which was added in ES5, has a number of different behaviors from normal/relaxed/lazy mode. One such behavior is that it disallows the automatic/implicit global variable creation. In that case, there would be no global *Scope*'d variable to hand back from an LHS look-up, and *Engine* would throw a `ReferenceError` similarly to the RHS case.

Now, if a variable is found for an RHS look-up, but you try to do something with its value that is impossible, such as trying to execute-as-function a non-function value, or reference a property on a `null` or `undefined` value, then *Engine* throws a different kind of error, called a `TypeError`.

[1] MDN: Strict Mode
[2] MDN: Strict Mode

`ReferenceError` is *Scope* resolution-failure related, whereas `TypeError` implies that *Scope* resolution was successful, but that there was an illegal/impossible action attempted against the result.

Review (TL;DR)

Scope is the set of rules that determines where and how a variable (identifier) can be looked-up. This look-up may be for the purposes of assigning to the variable, which is an LHS (left-hand-side) reference, or it may be for the purposes of retrieving its value, which is an RHS (right-hand-side) reference.

LHS references result from assignment operations. *Scope*-related assignments can occur either with the `=` operator or by passing arguments to (assign to) function parameters.

The JavaScript *Engine* first compiles code before it executes, and in so doing, it splits up statements like `var a = 2;` into two separate steps:

1. First, `var a` to declare it in that *Scope*. This is performed at the beginning, before code execution.
2. Later, `a = 2` to look up the variable (LHS reference) and assign to it if found.

Both LHS and RHS reference look-ups start at the currently executing *Scope*, and if need be (that is, they don't find what they're looking for there), they work their way up the nested *Scope*, one scope (floor) at a time, looking for the identifier, until they get to the global (top floor) and stop, and either find it, or don't.

Unfulfilled RHS references result in `ReferenceError`s being thrown. Unfulfilled LHS references result in an automatic, implicitly-created global of that name (if not in "Strict Mode" [3]), or a `ReferenceError` (if in "Strict Mode" [4]).

Quiz Answers

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. Identify all the LHS look-ups (there are 3!).

   `c = ..`, `a = 2` (implicit param assignment) and `b = ..`

[3] MDN: Strict Mode
[4] MDN: Strict Mode

2. Identify all the RHS look-ups (there are 4!).

   `foo(2.., = a;, a + .. and .. + b`

You Don't Know JS: Scope & Closures

Chapter 2: Lexical Scope

In Chapter 1, we defined "scope" as the set of rules that govern how the *Engine* can look up a variable by its identifier name and find it, either in the current *Scope*, or in any of the *Nested Scopes* it's contained within.

There are two predominant models for how scope works. The first of these is by far the most common, used by the vast majority of programming languages. It's called **Lexical Scope**, and we will examine it in-depth. The other model, which is still used by some languages (such as Bash scripting, some modes in Perl, etc.) is called **Dynamic Scope**.

Dynamic Scope is covered in Appendix A. I mention it here only to provide a contrast with Lexical Scope, which is the scope model that JavaScript employs.

Lex-time

As we discussed in Chapter 1, the first traditional phase of a standard language compiler is called lexing (aka, tokenizing). If you recall, the lexing process examines a string of source code characters and assigns semantic meaning to the tokens as a result of some stateful parsing.

It is this concept which provides the foundation to understand what lexical scope is and where the name comes from.

To define it somewhat circularly, lexical scope is scope that is defined at lexing time. In other words, lexical scope is based on where variables and blocks of scope are authored, by you, at write time, and thus is (mostly) set in stone by the time the lexer processes your code.

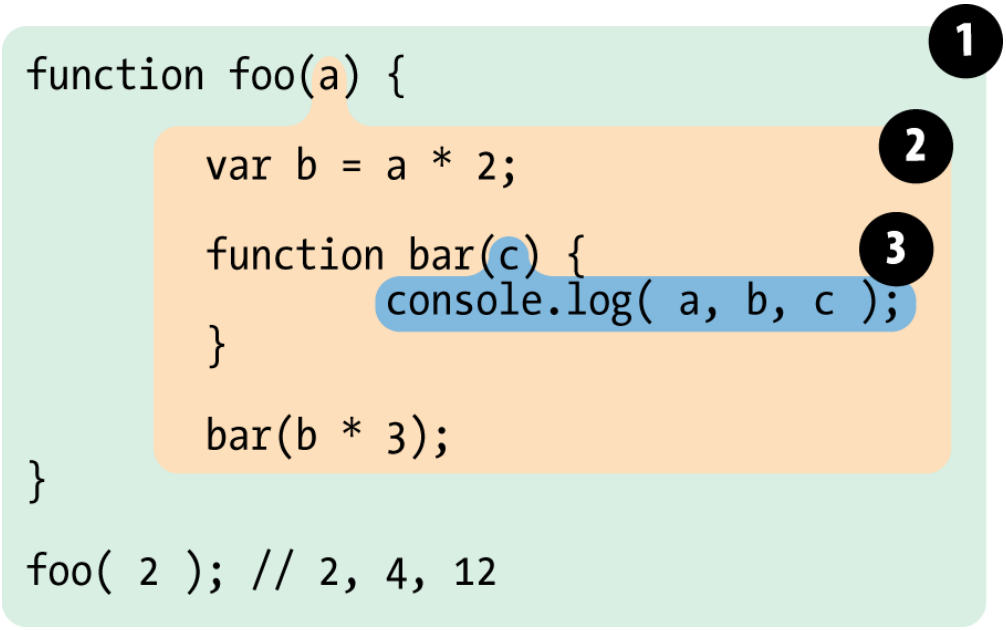
Note: We will see in a little bit there are some ways to cheat lexical scope, thereby modifying it after the lexer has passed by, but these are frowned upon. It is considered best practice to treat lexical scope as, in fact, lexical-only, and thus entirely author-time in nature.

Let's consider this block of code:

```
function foo(a) {

    var b = a * 2;
```

```
    function bar(c) {
        console.log( a, b, c );
    }

    bar(b * 3);
}

foo( 2 ); // 2 4 12
```

There are three nested scopes inherent in this code example. It may be helpful to think about these scopes as bubbles inside of each other.



Bubble 1 encompasses the global scope, and has just one identifier in it: `foo`.

Bubble 2 encompasses the scope of `foo`, which includes the three identifiers: `a`, `bar` and `b`.

Bubble 3 encompasses the scope of `bar`, and it includes just one identifier: `c`.

Scope bubbles are defined by where the blocks of scope are written, which one is nested inside the other, etc. In the next chapter, we'll discuss different units of scope, but for now, let's just assume that each function creates a new bubble of scope.

The bubble for `bar` is entirely contained within the bubble for `foo`, because (and only because) that's where we chose to define the function `bar`.

Notice that these nested bubbles are strictly nested. We're not talking about

Venn diagrams where the bubbles can cross boundaries. In other words, no bubble for some function can simultaneously exist (partially) inside two other outer scope bubbles, just as no function can partially be inside each of two parent functions.

Look-ups

The structure and relative placement of these scope bubbles fully explains to the *Engine* all the places it needs to look to find an identifier.

In the above code snippet, the *Engine* executes the `console.log(..)` statement and goes looking for the three referenced variables `a`, `b`, and `c`. It first starts with the innermost scope bubble, the scope of the `bar(..)` function. It won't find `a` there, so it goes up one level, out to the next nearest scope bubble, the scope of `foo(..)`. It finds `a` there, and so it uses that `a`. Same thing for `b`. But `c`, it does find inside of `bar(..)`.

Had there been a `c` both inside of `bar(..)` and inside of `foo(..)`, the `console.log(..)` statement would have found and used the one in `bar(..)`, never getting to the one in `foo(..)`.

Scope look-up stops once it finds the first match. The same identifier name can be specified at multiple layers of nested scope, which is called "shadowing" (the inner identifier "shadows" the outer identifier). Regardless of shadowing, scope look-up always starts at the innermost scope being executed at the time, and works its way outward/upward until the first match, and stops.

Note: Global variables are also automatically properties of the global object (`window` in browsers, etc.), so it *is* possible to reference a global variable not directly by its lexical name, but instead indirectly as a property reference of the global object.

```
window.a
```

This technique gives access to a global variable which would otherwise be inaccessible due to it being shadowed. However, non-global shadowed variables cannot be accessed.

No matter *where* a function is invoked from, or even *how* it is invoked, its lexical scope is **only** defined by where the function was declared.

The lexical scope look-up process *only* applies to first-class identifiers, such as the `a`, `b`, and `c`. If you had a reference to `foo.bar.baz` in a piece of code, the lexical scope look-up would apply to finding the `foo` identifier, but once it locates that variable, object property-access rules take over to resolve the `bar` and `baz` properties, respectively.

Cheating Lexical

If lexical scope is defined only by where a function is declared, which is entirely an author-time decision, how could there possibly be a way to "modify" (aka, cheat) lexical scope at run-time?

JavaScript has two such mechanisms. Both of them are equally frowned-upon in the wider community as bad practices to use in your code. But the typical arguments against them are often missing the most important point: **cheating lexical scope leads to poorer performance.**

Before I explain the performance issue, though, let's look at how these two mechanisms work.

`eval`

The `eval(..)` function in JavaScript takes a string as an argument, and treats the contents of the string as if it had actually been authored code at that point in the program. In other words, you can programmatically generate code inside of your authored code, and run the generated code as if it had been there at author time.

Evaluating `eval(..)` (pun intended) in that light, it should be clear how `eval(..)` allows you to modify the lexical scope environment by cheating and pretending that author-time (aka, lexical) code was there all along.

On subsequent lines of code after an `eval(..)` has executed, the *Engine* will not "know" or "care" that the previous code in question was dynamically interpreted and thus modified the lexical scope environment. The *Engine* will simply perform its lexical scope look-ups as it always does.

Consider the following code:

```
function foo(str, a) {
    eval( str ); // cheating!
    console.log( a, b );
}

var b = 2;

foo( "var b = 3;", 1 ); // 1 3
```

The string `"var b = 3;"` is treated, at the point of the `eval(..)` call, as code that was there all along. Because that code happens to declare a new variable `b`, it modifies the existing lexical scope of `foo(..)`. In fact, as mentioned above, this code actually creates variable `b` inside of `foo(..)` that shadows the `b` that was declared in the outer (global) scope.

When the `console.log(..)` call occurs, it finds both `a` and `b` in the scope of `foo(..)`, and never finds the outer `b`. Thus, we print out "1 3" instead of "1 2" as would have normally been the case.

Note: In this example, for simplicity's sake, the string of "code" we pass in was a fixed literal. But it could easily have been programmatically created by adding characters together based on your program's logic. `eval(..)` is usually used to execute dynamically created code, as dynamically evaluating essentially static code from a string literal would provide no real benefit to just authoring the code directly.

By default, if a string of code that `eval(..)` executes contains one or more declarations (either variables or functions), this action modifies the existing lexical scope in which the `eval(..)` resides. Technically, `eval(..)` can be invoked "indirectly", through various tricks (beyond our discussion here), which causes it to instead execute in the context of the global scope, thus modifying it. But in either case, `eval(..)` can at runtime modify an author-time lexical scope.

Note: `eval(..)` when used in a strict-mode program operates in its own lexical scope, which means declarations made inside of the `eval()` do not actually modify the enclosing scope.

```
function foo(str) {
   "use strict";
   eval( str );
   console.log( a ); // ReferenceError: a is not defined
}

foo( "var a = 2" );
```

There are other facilities in JavaScript which amount to a very similar effect to `eval(..)`. `setTimeout(..)` and `setInterval(..)` *can* take a string for their respective first argument, the contents of which are `eval`uated as the code of a dynamically-generated function. This is old, legacy behavior and long-since deprecated. Don't do it!

The `new Function(..)` function constructor similarly takes a string of code in its **last** argument to turn into a dynamically-generated function (the first argument(s), if any, are the named parameters for the new function). This function-constructor syntax is slightly safer than `eval(..)`, but it should still be avoided in your code.

The use-cases for dynamically generating code inside your program are incredibly rare, as the performance degradations are almost never worth the capability.

`with`

The other frowned-upon (and now deprecated!) feature in JavaScript which cheats lexical scope is the `with` keyword. There are multiple valid ways that `with` can be explained, but I will choose here to explain it from the perspective of how it interacts with and affects lexical scope.

`with` is typically explained as a short-hand for making multiple property references against an object *without* repeating the object reference itself each time.

For example:

```
var obj = {
    a: 1,
    b: 2,
    c: 3
};

// more "tedious" to repeat "obj"
obj.a = 2;
obj.b = 3;
obj.c = 4;

// "easier" short-hand
with (obj) {
    a = 3;
    b = 4;
    c = 5;
}
```

However, there's much more going on here than just a convenient short-hand for object property access. Consider:

```
function foo(obj) {
    with (obj) {
        a = 2;
    }
}

var o1 = {
    a: 3
};

var o2 = {
    b: 3
};
```

```
foo( o1 );
console.log( o1.a ); // 2

foo( o2 );
console.log( o2.a ); // undefined
console.log( a ); // 2 -- Oops, leaked global!
```

In this code example, two objects `o1` and `o2` are created. One has an `a` property, and the other does not. The `foo(..)` function takes an object reference `obj` as an argument, and calls `with (obj) { .. }` on the reference. Inside the `with` block, we make what appears to be a normal lexical reference to a variable `a`, an LHS reference in fact (see Chapter 1), to assign to it the value of `2`.

When we pass in `o1`, the `a = 2` assignment finds the property `o1.a` and assigns it the value `2`, as reflected in the subsequent `console.log(o1.a)` statement. However, when we pass in `o2`, since it does not have an `a` property, no such property is created, and `o2.a` remains `undefined`.

But then we note a peculiar side-effect, the fact that a global variable `a` was created by the `a = 2` assignment. How can this be?

The `with` statement takes an object, one which has zero or more properties, and **treats that object as if *it* is a wholly separate lexical scope**, and thus the object's properties are treated as lexically defined identifiers in that "scope".

Note: Even though a `with` block treats an object like a lexical scope, a normal `var` declaration inside that `with` block will not be scoped to that `with` block, but instead the containing function scope.

While the `eval(..)` function can modify existing lexical scope if it takes a string of code with one or more declarations in it, the `with` statement actually creates a **whole new lexical scope** out of thin air, from the object you pass to it.

Understood in this way, the "scope" declared by the `with` statement when we passed in `o1` was `o1`, and that "scope" had an "identifier" in it which corresponds to the `o1.a` property. But when we used `o2` as the "scope", it had no such `a` "identifier" in it, and so the normal rules of LHS identifier look-up (see Chapter 1) occurred.

Neither the "scope" of `o2`, nor the scope of `foo(..)`, nor the global scope even, has an `a` identifier to be found, so when `a = 2` is executed, it results in the automatic-global being created (since we're in non-strict mode).

It is a strange sort of mind-bending thought to see `with` turning, at runtime, an object and its properties into a "scope" *with* "identifiers". But that is the clearest explanation I can give for the results we see.

Note: In addition to being a bad idea to use, both `eval(..)` and `with` are affected (restricted) by Strict Mode. `with` is outright disallowed, whereas various forms of indirect or unsafe `eval(..)` are disallowed while retaining the core functionality.

Performance

Both `eval(..)` and `with` cheat the otherwise author-time defined lexical scope by modifying or creating new lexical scope at runtime.

So, what's the big deal, you ask? If they offer more sophisticated functionality and coding flexibility, aren't these *good* features? **No.**

The JavaScript *Engine* has a number of performance optimizations that it performs during the compilation phase. Some of these boil down to being able to essentially statically analyze the code as it lexes, and pre-determine where all the variable and function declarations are, so that it takes less effort to resolve identifiers during execution.

But if the *Engine* finds an `eval(..)` or `with` in the code, it essentially has to *assume* that all its awareness of identifier location may be invalid, because it cannot know at lexing time exactly what code you may pass to `eval(..)` to modify the lexical scope, or the contents of the object you may pass to `with` to create a new lexical scope to be consulted.

In other words, in the pessimistic sense, most of those optimizations it *would* make are pointless if `eval(..)` or `with` are present, so it simply doesn't perform the optimizations *at all*.

Your code will almost certainly tend to run slower simply by the fact that you include an `eval(..)` or `with` anywhere in the code. No matter how smart the *Engine* may be about trying to limit the side-effects of these pessimistic assumptions, **there's no getting around the fact that without the optimizations, code runs slower.**

Review (TL;DR)

Lexical scope means that scope is defined by author-time decisions of where functions are declared. The lexing phase of compilation is essentially able to know where and how all identifiers are declared, and thus predict how they will be looked-up during execution.

Two mechanisms in JavaScript can "cheat" lexical scope: `eval(..)` and `with`. The former can modify existing lexical scope (at runtime) by evaluating a string of "code" which has one or more declarations in it. The latter essentially creates a whole new lexical scope (again, at runtime) by treating an object reference *as* a "scope" and that object's properties as scoped identifiers.

The downside to these mechanisms is that it defeats the *Engine*'s ability to perform compile-time optimizations regarding scope look-up, because the *Engine* has to assume pessimistically that such optimizations will be invalid. Code *will* run slower as a result of using either feature. **Don't use them.**

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" [5], 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:

[5] Principle of Least Privilege

```
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" [6] 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?

[6] Principle of Least Privilege

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 redefinition, 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 develop-

ers can and should use both function-scope and block-scope techniques where respectively appropriate to produce better, more readable/maintainable code.

You Don't Know JS: Scope & Closures

Chapter 4: Hoisting

By now, you should be fairly comfortable with the idea of scope, and how variables are attached to different levels of scope depending on where and how they are declared. Both function scope and block scope behave by the same rules in this regard: any variable declared within a scope is attached to that scope.

But there's a subtle detail of how scope attachment works with declarations that appear in various locations within a scope, and that detail is what we will examine here.

Chicken Or The Egg?

There's a temptation to think that all of the code you see in a JavaScript program is interpreted line-by-line, top-down in order, as the program executes. While that is substantially true, there's one part of that assumption which can lead to incorrect thinking about your program.

Consider this code:

```
a = 2;

var a;

console.log( a );
```

What do you expect to be printed in the `console.log(..)` statement?

Many developers would expect `undefined`, since the `var a` statement comes after the `a = 2`, and it would seem natural to assume that the variable is re-defined, and thus assigned the default `undefined`. However, the output will be `2`.

Consider another piece of code:

```
console.log( a );

var a = 2;
```

You might be tempted to assume that, since the previous snippet exhibited some less-than-top-down looking behavior, perhaps in this snippet, `2` will also be

printed. Others may think that since the `a` variable is used before it is declared, this must result in a `ReferenceError` being thrown.

Unfortunately, both guesses are incorrect. `undefined` is the output.

So, what's going on here? It would appear we have a chicken-and-the-egg question. Which comes first, the declaration ("egg"), or the assignment ("chicken")?

The Compiler Strikes Again

To answer this question, we need to refer back to Chapter 1, and our discussion of compilers. Recall that the *Engine* actually will compile your JavaScript code before it interprets it. Part of the compilation phase was to find and associate all declarations with their appropriate scopes. Chapter 2 showed us that this is the heart of Lexical Scope.

So, the best way to think about things is that all declarations, both variables and functions, are processed first, before any part of your code is executed.

When you see `var a = 2;`, you probably think of that as one statement. But JavaScript actually thinks of it as two statements: `var a;` and `a = 2;`. The first statement, the declaration, is processed during the compilation phase. The second statement, the assignment, is left **in place** for the execution phase.

Our first snippet then should be thought of as being handled like this:

```
var a;

a = 2;

console.log( a );
```

... where the first part is the compilation and the second part is the execution.

Similarly, our second snippet is actually processed as:

```
var a;

console.log( a );

a = 2;
```

So, one way of thinking, sort of metaphorically, about this process, is that variable and function declarations are "moved" from where they appear in the flow of the code to the top of the code. This gives rise to the name "Hoisting".

In other words, **the egg (declaration) comes before the chicken (assignment)**.

Note: Only the declarations themselves are hoisted, while any assignments or other executable logic are left *in place*. If hoisting were to re-arrange the executable logic of our code, that could wreak havoc.

```
foo();

function foo() {
    console.log( a ); // undefined

    var a = 2;
}
```

The function `foo`'s declaration (which in this case *includes* the implied value of it as an actual function) is hoisted, such that the call on the first line is able to execute.

It's also important to note that hoisting is **per-scope**. So while our previous snippets were simplified in that they only included global scope, the `foo(..)` function we are now examining itself exhibits that `var a` is hoisted to the top of `foo(..)` (not, obviously, to the top of the program). So the program can perhaps be more accurately interpreted like this:

```
function foo() {
    var a;

    console.log( a ); // undefined

    a = 2;
}

foo();
```

Function declarations are hoisted, as we just saw. But function expressions are not.

```
foo(); // not ReferenceError, but TypeError!

var foo = function bar() {
    // ...
};
```

The variable identifier `foo` is hoisted and attached to the enclosing scope (global) of this program, so `foo()` doesn't fail as a `ReferenceError`. But `foo` has no value yet (as it would if it had been a true function declaration instead of expression). So, `foo()` is attempting to invoke the `undefined` value, which is a `TypeError` illegal operation.

Also recall that even though it's a named function expression, the name identifier is not available in the enclosing scope:

```
foo(); // TypeError
bar(); // ReferenceError

var foo = function bar() {
    // ...
};
```

This snippet is more accurately interpreted (with hoisting) as:

```
var foo;

foo(); // TypeError
bar(); // ReferenceError

foo = function() {
    var bar = ...self...
    // ...
}
```

Functions First

Both function declarations and variable declarations are hoisted. But a subtle detail (that *can* show up in code with multiple "duplicate" declarations) is that functions are hoisted first, and then variables.

Consider:

```
foo(); // 1

var foo;

function foo() {
    console.log( 1 );
}

foo = function() {
    console.log( 2 );
};
```

1 is printed instead of 2! This snippet is interpreted by the *Engine* as:

```
function foo() {
    console.log( 1 );
}

foo(); // 1

foo = function() {
```

```
    console.log( 2 );
};
```

Notice that `var foo` was the duplicate (and thus ignored) declaration, even though it came before the `function foo()...` declaration, because function declarations are hoisted before normal variables.

While multiple/duplicate `var` declarations are effectively ignored, subsequent function declarations *do* override previous ones.

```
foo(); // 3

function foo() {
    console.log( 1 );
}

var foo = function() {
    console.log( 2 );
};

function foo() {
    console.log( 3 );
}
```

While this all may sound like nothing more than interesting academic trivia, it highlights the fact that duplicate definitions in the same scope are a really bad idea and will often lead to confusing results.

Function declarations that appear inside of normal blocks typically hoist to the enclosing scope, rather than being conditional as this code implies:

```
foo(); // "b"

var a = true;
if (a) {
   function foo() { console.log( "a" ); }
}
else {
   function foo() { console.log( "b" ); }
}
```

However, it's important to note that this behavior is not reliable and is subject to change in future versions of JavaScript, so it's probably best to avoid declaring functions in blocks.

Review (TL;DR)

We can be tempted to look at `var a = 2;` as one statement, but the JavaScript *Engine* does not see it that way. It sees `var a` and `a = 2` as two separate statements, the first one a compiler-phase task, and the second one an execution-phase task.

What this leads to is that all declarations in a scope, regardless of where they appear, are processed *first* before the code itself is executed. You can visualize this as declarations (variables and functions) being "moved" to the top of their respective scopes, which we call "hoisting".

Declarations themselves are hoisted, but assignments, even assignments of function expressions, are *not* hoisted.

Be careful about duplicate declarations, especially mixed between normal var declarations and function declarations – peril awaits if you do!

You Don't Know JS: Scope & Closures

Chapter 5: Scope Closure

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.

Modules

There are other code patterns which leverage the power of closure but which do not on the surface appear to be about callbacks. Let's examine the most powerful of them: *the module*.

```
function foo() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }
}
```

As this code stands right now, there's no observable closure going on. We simply have some private data variables `something` and `another`, and a couple of inner functions `doSomething()` and `doAnother()`, which both have lexical scope (and thus closure!) over the inner scope of `foo()`.

But now consider:

```
function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
}

var foo = CoolModule();

foo.doSomething(); // cool
```

```
foo.doAnother(); // 1 ! 2 ! 3
```

This is the pattern in JavaScript we call *module*. The most common way of implementing the module pattern is often called "Revealing Module", and it's the variation we present here.

Let's examine some things about this code.

Firstly, `CoolModule()` is just a function, but it *has to be invoked* for there to be a module instance created. Without the execution of the outer function, the creation of the inner scope and the closures would not occur.

Secondly, the `CoolModule()` function returns an object, denoted by the object-literal syntax `{ key: value, ... }`. The object we return has references on it to our inner functions, but *not* to our inner data variables. We keep those hidden and private. It's appropriate to think of this object return value as essentially a **public API for our module**.

This object return value is ultimately assigned to the outer variable `foo`, and then we can access those property methods on the API, like `foo.doSomething()`.

Note: It is not required that we return an actual object (literal) from our module. We could just return back an inner function directly. jQuery is actually a good example of this. The `jQuery` and `$` identifiers are the public API for the jQuery "module", but they are, themselves, just a function (which can itself have properties, since all functions are objects).

The `doSomething()` and `doAnother()` functions have closure over the inner scope of the module "instance" (arrived at by actually invoking `CoolModule()`). When we transport those functions outside of the lexical scope, by way of property references on the object we return, we have now set up a condition by which closure can be observed and exercised.

To state it more simply, there are two "requirements" for the module pattern to be exercised:

1. There must be an outer enclosing function, and it must be invoked at least once (each time creates a new module instance).
2. The enclosing function must return back at least one inner function, so that this inner function has closure over the private scope, and can access and/or modify that private state.

An object with a function property on it alone is not *really* a module. An object which is returned from a function invocation which only has data properties on it and no closured functions is not *really* a module, in the observable sense.

The code snippet above shows a standalone module creator called `CoolModule()` which can be invoked any number of times, each time creating a new module instance. A slight variation on this pattern is when you only care to have one instance, a "singleton" of sorts:

```
var foo = (function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
})();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

Here, we turned our module function into an IIFE (see Chapter 3), and we *immediately* invoked it and assigned its return value directly to our single module instance identifier `foo`.

Modules are just functions, so they can receive parameters:

```
function CoolModule(id) {
    function identify() {
        console.log( id );
    }

    return {
        identify: identify
    };
}

var foo1 = CoolModule( "foo 1" );
var foo2 = CoolModule( "foo 2" );

foo1.identify(); // "foo 1"
foo2.identify(); // "foo 2"
```

Another slight but powerful variation on the module pattern is to name the object you are returning as your public API:

```
var foo = (function CoolModule(id) {
    function change() {
```

```
        // modifying the public API
        publicAPI.identify = identify2;
    }

    function identify1() {
        console.log( id );
    }

    function identify2() {
        console.log( id.toUpperCase() );
    }

    var publicAPI = {
        change: change,
        identify: identify1
    };

    return publicAPI;
})( "foo module" );

foo.identify(); // foo module
foo.change();
foo.identify(); // FOO MODULE
```

By retaining an inner reference to the public API object inside your module instance, you can modify that module instance **from the inside**, including adding and removing methods, properties, *and* changing their values.

Modern Modules

Various module dependency loaders/managers essentially wrap up this pattern of module definition into a friendly API. Rather than examine any one particular library, let me present a *very simple* proof of concept **for illustration purposes (only)**:

```
var MyModules = (function Manager() {
    var modules = {};

    function define(name, deps, impl) {
        for (var i=0; i<deps.length; i++) {
            deps[i] = modules[deps[i]];
        }
        modules[name] = impl.apply( impl, deps );
    }

    function get(name) {
```

```
        return modules[name];
    }

    return {
        define: define,
        get: get
    };
})();
```

The key part of this code is `modules[name] = impl.apply(impl, deps)`. This is invoking the definition wrapper function for a module (passing in any dependencies), and storing the return value, the module's API, into an internal list of modules tracked by name.

And here's how I might use it to define some modules:

```
MyModules.define( "bar", [], function(){
    function hello(who) {
        return "Let me introduce: " + who;
    }

    return {
        hello: hello
    };
} );

MyModules.define( "foo", ["bar"], function(bar){
    var hungry = "hippo";

    function awesome() {
        console.log( bar.hello( hungry ).toUpperCase() );
    }

    return {
        awesome: awesome
    };
} );

var bar = MyModules.get( "bar" );
var foo = MyModules.get( "foo" );

console.log(
    bar.hello( "hippo" )
); // Let me introduce: hippo

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Both the "foo" and "bar" modules are defined with a function that returns a

public API. "foo" even receives the instance of "bar" as a dependency parameter, and can use it accordingly.

Spend some time examining these code snippets to fully understand the power of closures put to use for our own good purposes. The key take-away is that there's not really any particular "magic" to module managers. They fulfill both characteristics of the module pattern I listed above: invoking a function definition wrapper, and keeping its return value as the API for that module.

In other words, modules are just modules, even if you put a friendly wrapper tool on top of them.

Future Modules

ES6 adds first-class syntax support for the concept of modules. When loaded via the module system, ES6 treats a file as a separate module. Each module can both import other modules or specific API members, as well export their own public API members.

Note: Function-based modules aren't a statically recognized pattern (something the compiler knows about), so their API semantics aren't considered until run-time. That is, you can actually modify a module's API during the run-time (see earlier `publicAPI` discussion).

By contrast, ES6 Module APIs are static (the APIs don't change at run-time). Since the compiler knows *that*, it can (and does!) check during (file loading and) compilation that a reference to a member of an imported module's API *actually exists*. If the API reference doesn't exist, the compiler throws an "early" error at compile-time, rather than waiting for traditional dynamic run-time resolution (and errors, if any).

ES6 modules **do not** have an "inline" format, they must be defined in separate files (one per module). The browsers/engines have a default "module loader" (which is overridable, but that's well-beyond our discussion here) which synchronously loads a module file when it's imported.

Consider:

bar.js

```
function hello(who) {
    return "Let me introduce: " + who;
}

export hello;
```

foo.js

```
// import only `hello()` from the "bar" module
import hello from "bar";
```

```
var hungry = "hippo";

function awesome() {
    console.log(
        hello( hungry ).toUpperCase()
    );
}

export awesome;
```

```
// import the entire "foo" and "bar" modules
module foo from "foo";
module bar from "bar";

console.log(
    bar.hello( "rhino" )
); // Let me introduce: rhino

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Note: Separate files **"foo.js"** and **"bar.js"** would need to be created, with the contents as shown in the first two snippets, respectively. Then, your program would load/import those modules to use them, as shown in the third snippet.

`import` imports one or more members from a module's API into the current scope, each to a bound variable (`hello` in our case). `module` imports an entire module API to a bound variable (`foo`, `bar` in our case). `export` exports an identifier (variable, function) to the public API for the current module. These operators can be used as many times in a module's definition as is necessary.

The contents inside the *module file* are treated as if enclosed in a scope closure, just like with the function-closure modules seen earlier.

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!