You Don't Know JS: Scope & Closures

Appendix A: Dynamic Scope

In Chapter 2, we talked about "Dynamic Scope" as a contrast to the "Lexical Scope" model, which is how scope works in JavaScript (and in fact, most other languages).

We will briefly examine dynamic scope, to hammer home the contrast. But, more importantly, dynamic scope actually is a near cousin to another mechanism (`this`) in JavaScript, which we covered in the "*this & Object Prototypes*" title of this book series.
As we saw in Chapter 2, lexical scope is the set of rules about how the *Engine* can look-up a variable and where it will find it. The key characteristic of lexical scope is that it is defined at author-time, when the code is written (assuming you don't cheat with `eval()` or `with`).
Dynamic scope seems to imply, and for good reason, that there's a model whereby scope can be determined dynamically at runtime, rather than statically at author-time. That is in fact the case. Let's illustrate via code:

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

function bar() {
        var a = 3;
        foo();
}

var a = 2;

bar();
```

Lexical scope holds that the RHS reference to `a` in `foo()` will be resolved to the global variable `a`, which will result in value `2` being output.
Dynamic scope, by contrast, doesn't concern itself with how and where functions and scopes are declared, but rather **where they are called from**. In other words, the scope chain is based on the call-stack, not the nesting of scopes in code.

So, if JavaScript had dynamic scope, when `foo()` is executed, **theoretically** the code below would instead result in `3` as the output.

```
function foo() {
        console.log( a ); // 3  (not 2!)
}

function bar() {
```

```
        var a = 3;
        foo();
}

var a = 2;

bar();
```

How can this be? Because when `foo()` cannot resolve the variable reference for `a`, instead of stepping up the nested (lexical) scope chain, it walks up the call-stack, to find where `foo()` was *called from*. Since `foo()` was called from `bar()`, it checks the variables in scope for `bar()`, and finds an `a` there with value `3`.
Strange? You're probably thinking so, at the moment.

But that's just because you've probably only ever worked on (or at least deeply considered) code which is lexically scoped. So dynamic scoping seems foreign. If you had only ever written code in a dynamically scoped language, it would seem natural, and lexical scope would be the odd-ball.

To be clear, JavaScript **does not, in fact, have dynamic scope**. It has lexical scope. Plain and simple. But the `this` mechanism is kind of like dynamic scope.
The key contrast: **lexical scope is write-time, whereas dynamic scope (and `this`!) are runtime**. Lexical scope cares *where a function was declared*, but dynamic scope cares where a function was *called from*.
Finally: `this` cares *how a function was called*, which shows how closely related the `this` mechanism is to the idea of dynamic scoping. To dig more into `this`, read the title "*this & Object Prototypes*".