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 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" (**A**bstract **S**yntax **T**ree).

   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."

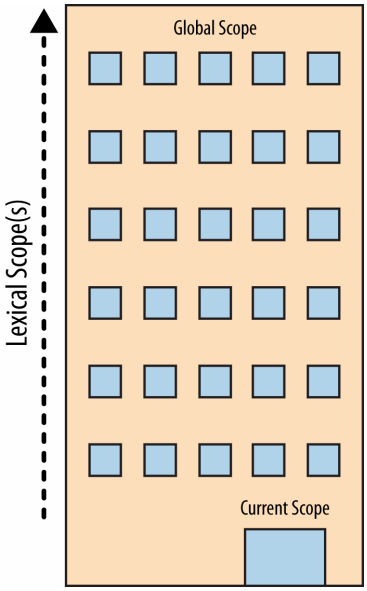
> ***Engine***: "Hey, *Scope* outside of `foo`, oh you're the global *Scope*, ok cool. Ever heard of `b`? Got an RHS reference for it."

> ***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" [^note-strictmode], 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" [^note-strictmode], 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`.

`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" [^note-strictmode]), or a `ReferenceError` (if in "Strict Mode" [^note-strictmode]).

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 = ..`

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

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

[^note-strictmode]: MDN: Strict Mode