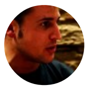

Nick Balestra Follow
Jan 17, 2015 · 6 min read

Javascript's lexical scope, hoisting and closures without mystery.

First, although javascript falls under the category of 'dynamic' or 'interpreted' languages it is actually a compiled language. I'm no compiler-guy, so I won't even try to go deeper into details, but is important to know this to better understand some of the logic governing the language. So, let me state it again: Javascript is a compiled language. Why this is so important? Well, if you just dig deeper into how a compiler works, you will see how JS scope is defined, leading you to understand not just scope, but also hoisting and closures without the mystery or magic that so often come attached to those topics.

This article is part of a series I'm mainly writing for myself during my learning journey, as I'm steering myself from being a product designer with a passion for code into a software engineer with a passion for design. I picked JS to be my mother tongue for many reasons: some falls under Axel Rauschmayer's post.
Treat this post as a summary/note on Kyle Simpson's book on scope & closures and his advanced JS workshop, in that sense this post is not comprehensive and I strongly encourage you to dig both of Kyle original resources as the amount of details he shovels there are worth every second of your time and every penny of your money. My thanks goes to him, for his awesome book on scope and closures and his ability to explain complex concepts easily enough for me to get.

Scope meets compiler

So, I mentioned that javascript is a compiled language, but what does this mean? Shortly, just before execution, the source code is sent by the engine trough a compiler, in which, during an early phase called lexing (or tokenizing), scope get defined. This doesn't just tell us what's in a name, but also remember us that lexical scope is based on where variables and blocks of scope have been authored in the source code. In other words, lexical scope is defined by you during author time and frozen by the lexer during compilation. Let’s show this with the help of a little example:

```
var a = ‘something’;
```

As the compiler encounter the variable declaration for *a (*var a) it asks scope to see if the variable *a* already exists for that particular scope. If so, it just ignore it and move forward, otherwise it asks scope to declare (create) a new variable named *a* for that scope.

Engine meets scope

When engine executes the code that the compiler produced it will see our above example’s code to be something like:

```
a = ‘something’;
```

Spot the difference? Yes, the var keyword is missing, this is because variables declaration was already handled by compiler, and engine will, therefore, proceed to the assignment. To do so, it will first do a scope LHS (left-hand side) look-up for variable *a.* As Kyle suggests in his book the conversation between Engine and Scope could sounds like:

> **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 variable just recently. Here you go.
> **Engine***:* Helpful as always, Scope. Thanks again. Now, time to assign the string ‘something’ to *a.*

At this point, we can all infer what scope is. Scope is where to look for things, or more precisely, scope is where engine will look for things. But what happen in the case of nested scopes? Where will engine look for things? Engine will perform an LHS look-up on the current scope and if he doesn’t find the variable in there it will then perform the LHS look-up on the closest outer scope, and so on, traversing nested scopes until it will get to the outermost scope, the global scope. There his search will terminate. If nothing is found in the global scope two things could happen depending if you are running in strict mode or not:

1. If you are running in ES5 strict mode: engine would throw a ReferenceError as you would logically imagine,
2. otherwise global scope will just create a variable with that name in the global scope and hand it back to the engine. We can imagine global scope's reply to sounds like:

> ***Global Scope***: No engine, there wasn't one before, but I was helpful and created one for you. Here it is.

Think of scopes as strictly nested bubbles, not like Venn diagrams where the bubbles can cross boundaries. As a takeaway, write code in strict mode.

Function and Block level scope

What create a scope then? Although not completely true the common wisdom is that javascript has function-based scope only. Meaning that each declared function create its own scope. Let's take the following example:

```
var a = 1;
function foo() {
    var a = 10;
    console.log(a);
}
console.log(a); // 1
foo();          // 10
```

Scope of *foo* is nested inside global scope, hiding what gets defined inside of it from the outer world. In such cases we can also speak of 'shadowing *a*'. Scope-based hiding techniques can be useful to build the API for a module/object, where you should expose only what is minimally necessary, and "hide" everything else, following the software design principle called "Principle of Least Privilege", also known as "Least Authority" or "Least Exposure.

What are then the mechanics that create block-level scope? *with* and *try/catch* and coming up in ES6 *const* and *let*. I'll give here some more details about the latest one mentioned: *let*. The *let* keyword can be used for variable declaration instead of *var*, and in fact attaches the variable declaration to the scope of whatever block (usually a { .. } pair) it's contained in. In other words, *let* implicitly hijacks any block's scope for its variable declaration.

```
{
    let foo = 10;
    console.log( foo ); // 10
}

console.log( foo ); // ReferenceError
```

let also come packed with a special behavior when used inside a for loop header:

```
for (let i = 0; i < 5; i++) {
    console.log( i ); // 0 1 2 3 4
}

console.log( i ); // ReferenceError
```

This is very useful not only for avoiding collisions and scope-pollution but also for closures as we'll see later as *let* will do its binding on every iteration of the loop.

Hoisting

First, let see hoisting in action

```
foo(); // 'hello hoisting'

function foo() {
    console.log('hello hoisting');
}
```

As you can see we can invoke *foo* before it has been declared and the code will run just as if the opposite happened. Well, by now something should start ringing a bell, wasn't this indeed happening during compilation? Right! During a compilation, every declaration (of variables or function) get added to the relative scope. Function declaration hoisting differs from variables as the content of the function get hoisted too. That's it. I've read so many posts about hosting and many seemed to refer to something more or less mystical, you know, when variables declaration are moved on the top following some unknown reason…

Remember that functions declaration and not function expression are hoisted, and that functions are hoisted first, and then variables:

```
foo(); // 3

var foo;

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

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

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

Also, notice how function declaration can override each other (instead, if we were going to declare multiple times var foo nothing will have happened). This should be pretty straightforward as variables declaration just create a variable in a specific scope with their name only while function declaration also bring with them their content.

However, declarations made with let will *not* hoist to the entire scope of the block they appear in. Such declarations will not observable "exist" in the block until the declaration statement.

```
{
   console.log( foo ); // ReferenceError!
   let foo = 10;
}
```

Closures

In his book Kyle gives a very straightforward definition for what a closure is:

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

If you think about lexical scope, this is so straight forward, cause scope is defined on author time and set in stone during compilation, therefore it have nothing to do with runtime and the call-stack(different will be if javascript will have been based on dynamic scope, but perhaps I will write a post about '*this*' in the near future…). This mean that a function *bar* defined inside a function *foo* will have access to the outer scope of *foo*, also if being returned within it and invoked outside of *foo*, or in other words, outside of its scope,yes, its lexical scope! If it sounds complex, this latest example will make it clearer.

```
function foo() {  // 'scope of foo' aka lexical scope for
bar
   var memory = 'hello closure';
   return function bar() {
      console.log(memory);
   }
}

var memory = null,
    baz = foo();

baz(); // 'hello closure'
```

Thanks to Bojan, Stian and Ryan for reviews and feedbacks.

If you have any question, comment or feedback, please get in touch on twitter or drop me a line at nick at balestra dot ch.