David Shariff
MANAGE THE UI PLATFORM TEAM @ AMAZON.COM

Follow @davidshariff 2,185 followers

I'm hiring! Send your resumé to:
primenow-hiring@amazon.com

Previously at Yahoo, RBS, Richi and Trend Micro.

Blog Homepage View my Github See my LinkedIn

Views are my own, and don't represent those of my employer.

Identifier Resolution and Closures in the JavaScript Scope Chain

From my previous post, we now know that every function has an associated `execution context` that contains a `variable object [VO]`, which is composed of all the variables, functions and parameters defined inside that given local function.

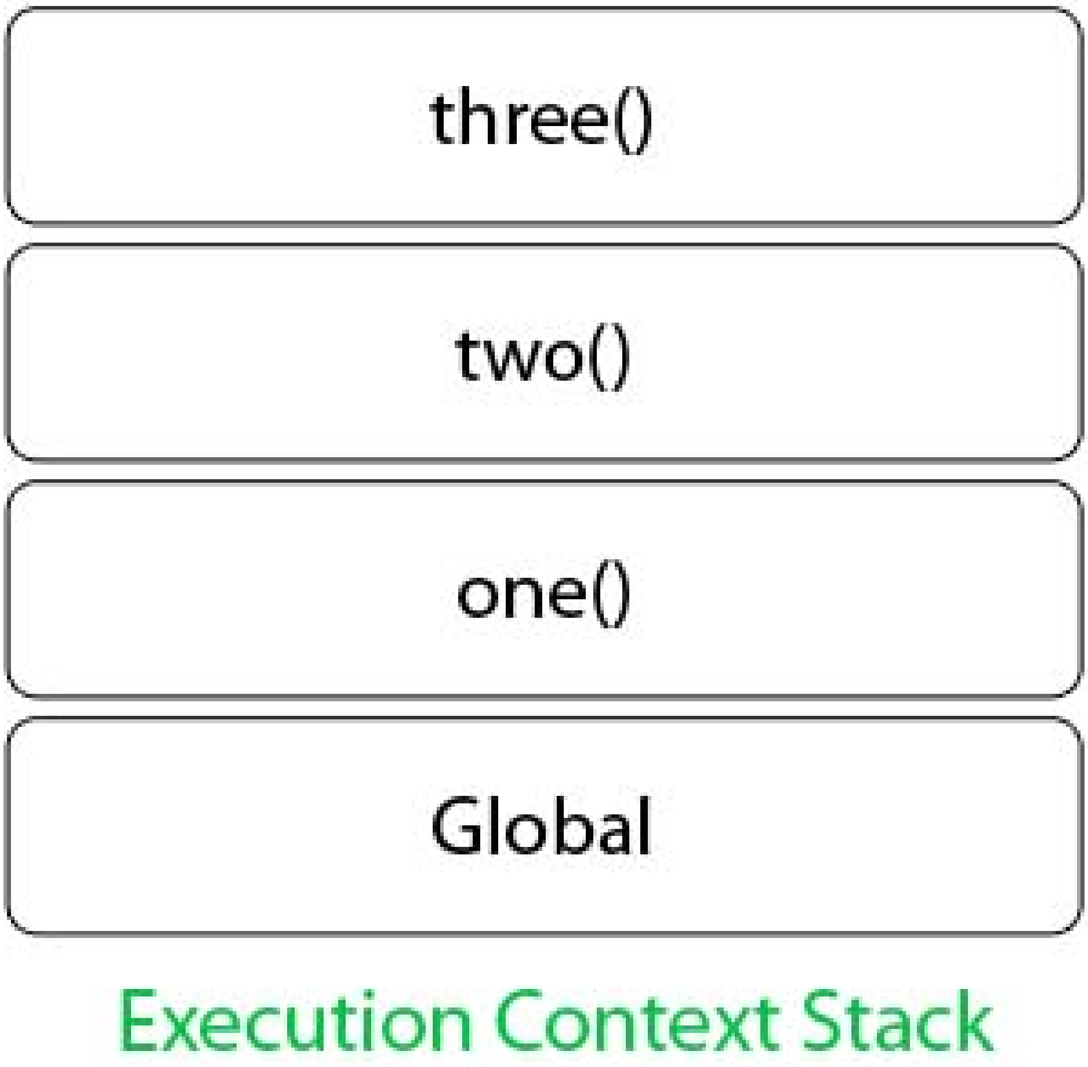
The `scope chain` property of each `execution context` is simply a collection of the current `context's` `[VO]` + all parent's lexical `[VO]`s.

```
Scope = VO + All Parent VOs
Eg: scopeChain = [ [VO] + [VO1] + [VO2] + [VO n+1] ];
```

Determining a Scope Chain's Variable Objects [VO]s

We now know that the first `[VO]` of the `scope chain` belongs to the current `execution context`, and we can find the remaining parent [VO]s by looking at the parent context's `scope chain`:

```
function one() {

    two();

    function two() {

        three();

        function three() {
            alert('I am at function three');
        }

    }

}

one();
```



The example is straight forward, starting from the global context we call `one()`, `one()` calls `two()`, which in turn calls `three()`, thus alerting that it is at function three. The image above shows the call stack at function `three` at the time `alert('I am at function three')` is fired. We can see that the `scope chain` at this point in time looks as follows:

```
three() Scope Chain = [ [three() VO] + [two() VO] + [one() VO] +
[Global VO] ];
```

Lexical Scope

An important feature of JavaScript to note, is that the interpreter uses `Lexical Scoping`, as opposed to Dynamic Scoping. This is just a complicated way of saying all inner functions, are statically (lexically) bound to the parent context in which the inner function was physically defined in the program code.

In our previous example above, it does not matter in which sequence the inner functions are called. `three()` will always be statically bound to `two()`, which in turn will always be bound to `one()` and so on and so forth. This gives a chaining effect where all inner functions can access the outer functions `VO` through the statically bound `Scope Chain`.

This `lexical scope` is the source of confusion for many developers. We know that every invocation of a function will create a new `execution context` and associated `VO`, which holds the values of variables evaluated in the current context.

It is this dynamic, runtime evaluation of the `VO` paired with the lexical (static) defined scope of each context that leads unexpected results in program behaviour. Take the following classic example:

```
var myAlerts = [];

for (var i = 0; i < 5; i++) {
    myAlerts.push(
        function inner() {
            alert(i);
        }
    );
}

```

At

```
11  myAlerts[0](); // 5
12  myAlerts[1](); // 5
13  myAlerts[2](); // 5
14  myAlerts[3](); // 5
15  myAlerts[4](); // 5
```

first glance, those new to JavaScript would assume `alert(i);` to be the value of `i` on each increment where the function was physically defined in the source code, alerting 1, 2, 3, 4 and 5 respectively.

This is the most common point of confusion. Function `inner` was created in the global context, therefore its scope chain is statically bound to the global context.

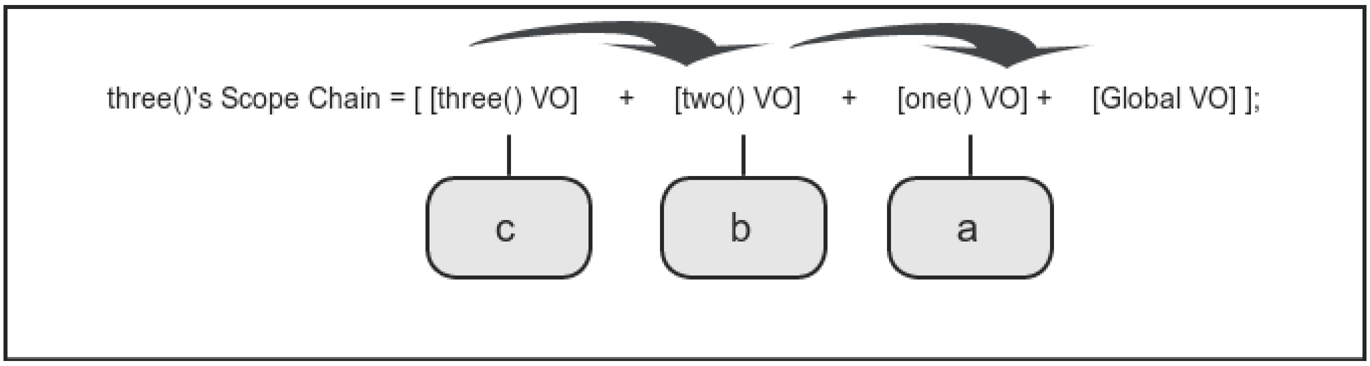
Lines 11 ~ 15 invoke `inner()`, which looks in `inner.ScopeChain` to resolve `i`, which is located in the `global` context. At the time of each invocation, `i`, has already been incremented to 5, giving the same result every time `inner()` is called. The statically bound scope chain, which holds `[VOs]` from each `context` containing live variables, often catches developers by surprise.

Resolving the value of variables

The following example alerts the value of variables `a`, `b` and `c`, which gives us a result of 6.

```
function one() {

    var a = 1;
    two();

    function two() {

        var b = 2;
        three();

        function three() {

            var c = 3;
            alert(a + b + c); // 6

        }

        }

}

one();
```

Line 14 is intriguing, at first glance it seems that `a` and `b` are not “inside” function three, so how can this code still work? To understand how the interpreter evaluates this code, we need to look at the scope chain of function three at the time line 14 was executed:


three()'s Scope Chain = [[three() VO] + [two() VO] + [one() VO] + [Global VO]];
c
b
a

When the interpreter executes line 14: `alert(a + b + c)`, it resolves `a` first by looking into the scope chain and checking the first variable object, `three's [VO]`. It checks to see if `a` exists inside `three's [VO]` but can not find any property with that name, so moves on to check the next `[VO]`.

The interpreter keeps checking each `[VO]` in sequence for the existence of the variable name, in which case the value will be returned to the original evaluated code, or the program will throw a `ReferenceError` if none is found. Therefore, given the example above, you can see that `a`, `b` and `c` are all resolvable given function three’s scope chain.

How does this work with closures?

In JavaScript, closures are often regarded as some sort of magical unicorn that only advanced developers can really understand, but truth be told it is just a simple understanding of the scope chain. A closure, as Crockford says, is simply:

> *An inner function always has access to the vars and parameters of its outer function, even after the outer function has returned…*

The code below is an example of a closure:

```
function foo() {
    var a = 'private variable';
    return function bar() {
        alert(a);
    }
}

var callAlert = foo();

callAlert(); // private variable
```

The `global context` has a function named `foo()` and a variable named `callAlert`, which holds the returned value of `foo()`. What often surprises and confuses developers is that the private variable, `a`, is still available even after `foo()` has finished executing.

However, if we look at each of the context in detail, we will see the following:

```
// Global Context when evaluated
global.VO = {
    foo: pointer to foo(),
    callAlert: returned value of global.VO.foo
    scopeChain: [global.VO]
}

// Foo Context when evaluated
foo.VO = {
    bar: pointer to bar(),
    a: 'private variable',
    scopeChain: [foo.VO, global.VO]
}

```

```
// Bar Context when evaluated
bar.VO = {
    scopeChain: [bar.VO, foo.VO, global.VO]
}
```

Now we can see by invoking `callAlert()`, we get the function `foo()`, which returns the pointer to `bar()`. On entering `bar()`, `bar.VO.scopeChain` is `[bar.VO, foo.VO, global.VO]`.

By alerting `a`, the interpreter checks the first VO in the `bar.VO.scopeChain` for a property named `a` but can not find a match, so promptly moves on to the next VO, `foo.VO`.

It checks for the existence of the property and this time finds a match, returning the value back to the `bar` context, which explains why the `alert` gives us `'private variable'` even though `foo()` had finished executing sometime ago.

By this point in the article, we have covered the details of the `scope chain` and its `lexical` environment, along with how `closures` and `variable resolution` work. The rest of this article looks at some interesting situations in relation to those covered above.

Wait, how does the prototype chain affect variable resolution?

JavaScript is prototypal by nature and almost everything in the language, except for `null` and `undefined`, are `objects`. When trying to access a property on an `object`, the interpreter will try to resolve it by looking for the existence of the property in the `object`. If it can't find the property, it will continue to look up the prototype chain, which is an inherited chain of objects, until it finds the property, or traversed to the end of the chain.

This leads to an interesting question, does the interpreter resolve an object property using the `scope chain` or `prototype chain` first ? It uses both. When trying to resolve a property or identifier, the `scope chain` will be used first to locate the `object`. Once the `object` has been found, the `prototype chain` of that `object` will then be traversed looking for the property name. Let’s look at an example:

```
1   var bar = {};
2
3   function foo() {
4
5       bar.a = 'Set from foo()';
6
7       return function inner() {
8           alert(bar.a);
9       }
10
11  }
12
13  foo()(); // 'Set from foo()'
```

Line 5 creates the property `a` on the global object `bar`, and sets its value to `'Set from foo()'`. The interpreter looks into the `scope chain` and as expected finds `bar.a` in the `global context`. Now, lets consider the following:

```
1   var bar = {};
2
3   function foo() {
4
5       Object.prototype.a = 'Set from prototype';
6
7       return function inner() {
8           alert(bar.a);
9       }
10
11  }
12
13  foo()(); // 'Set from prototype()'
```

At runtime, we invoke `inner()`, which tries to resolve `bar.a` by looking in its scope chain for the existence of `bar`. It finds `bar` in the global context, and proceeds to search `bar` for a property named `a`. However, `a` was never set on `bar`, so the interpreter traverses the object's prototype chain and finds `a` was set on `Object.prototype`.

It is this exact behavior which explains identifier resolution; locate the `object` in the `scope chain`, then proceed up the object's `prototype chain` until the property is found, or returned `undefined`.

When to use Closures?

Closures are a powerful concept given to JavaScript and some of the most common situations to use them are:

Encapsulation

Allows us to hide the implementation details of a context from outside scopes, while exposing a controlled public interface. This is commonly referred to as the module pattern or revealing module pattern.

Callbacks

Perhaps one of the most powerful uses for closures are callbacks. JavaScript, in the browser, typically runs in a single threaded event loop, blocking other events from starting until one event has finished. Callbacks allow us to defer the invocation of a function, typically in response to an event completing, in a non-blocking manner. An example of this is when making an AJAX call to the server, using a callback to handle to response, while still maintaining the bindings in which it was created.

Closures as arguments

We can also pass closures as arguments to a function, which is a powerful functional paradigm for creating more graceful solutions for complex code. Take for example a minimum sort function. By passing closures as parameters, we could define the implementation for different types of data sorting, while still reusing a single function body as a schematic.

When not to use Closures ?

Although closures are powerful, they should be used sparingly due to some performance concerns:

Large scope lengths

Multiple nested functions are a typical sign that you might run into some performance issues. Remember, every time you need to evaluate a variable, the Scope Chain must be traversed to find the identifier, so it goes without saying that the further down the chain the variable is defined, the longer to lookup time.

Garbage collection

JavaScript is a `garbage collected` language, which means developers generally don’t have to worry about memory management, unlike lower level programming languages. However, this automatic garbage collection often leads developers application to suffer from poor performance and memory leaks.

Different JavaScript engines implement garbage collection slightly different, since ECMAScript does not define how the implementation should be handled, but the same philosophy can apply across engines when trying to create high performance, leak free JavaScript code. Generally speaking, the garbage collector will try to free the memory of objects when they can not be referenced by any other live object running in the program, or are unreachable.

Circular references

This leads us to closures, and the possibility of circular references in a program, which is a term used to describe a situation where one object references another object, and that object points back to the first object. Closures are especially susceptible to leaks, remember that an inner function can reference a variable defined further up the scope chain even after the parent has finished executing and returned. Most JavaScript engines handle these situations quite well (damn you IE), but it’s still worth noting and taking into consideration when doing your development.

For older versions of IE, referencing a DOM element would often cause you memory leaks. Why? In IE, the JavaScript (JScript ?) engine and DOM both have their own individual garbage collector. So when referencing a DOM element from JavaScript, the native collector hands off to the DOM and the DOM collector points back to native, resulting in neither collector knowing about the circular reference.

Summary

From working with many developers over the past few years, I often found that the concepts of `scope chain` and `closures` were known about, but not truly understood in detail. I hope this article has helped to take you from knowing the basic concept, to an understanding in more detail and depth.

Going forward, you should be armed with all the knowledge you need to determine how the resolution of variables, in any situation, works when writing your JavaScript. Happy coding !

Related Posts...

1. What is the Execution Context & Stack in JavaScript?
2. JavaScript Inheritance Patterns
3. JavaScript’s Undefined Explored
4. Chaining Variable Assignments in JavaScript: Words of Caution
5. Futures and Promises in JavaScript

Comments

mauro77 ***said*** **on 19/01/2013 at 11:55 am:**

Nice and clear. Good job !

Reply ↓

jackie ***said*** **on 24/01/2013 at 10:06 am:**

thank u again for another great article!

Reply ↓

zhuang ***said*** **on 05/02/2013 at 4:50 pm:**

Really appreciate for your article

Reply ↓

Willson ***said*** **on 26/04/2013 at 12:50 am:**

This is another awesome article – thank you for so clearly explaining how scope chains are generated and they interact with the prototype chain!

Reply ↓

wangtno ***said*** **on 27/04/2013 at 8:18 pm:**

thanks! I have been looking for a clear explaination of closure for a long time.

Reply ↓

Stephan ***said*** **on 29/04/2013 at 6:55 pm:**

Maybe I am wrong, but to my opinion the foo() context VO is empty (foo has no params, no vars, no function decl.), so the line :

"Line 5 creates the property a on the global object bar, and sets its value to 'Set from foo()'. The interpreter looks into the scope chain and as expected finds bar.a in the foo context. Now, lets consider the following:"

–> should be:

"Line 5 creates the property a on the global object bar, and sets its value to 'Set from foo()'. The interpreter looks into the scope chain and as expected finds bar.a in the GLOBAL context. Now, lets consider the following:"

—
Regards Stephan

Reply ↓

David Shariff *said* on 29/04/2013 at 7:09 pm:

@Stephen,

Good spot, it was a typo, I updated that line

Reply ↓

Apurv ***said*** **on** 23/08/2013 at 4:41 pm:

Correct me if I am wrong. But I don't think foo's context VO would be empty. It would contain the definition of inner(). Because,

```
return function inner(){..}
```

is equivalent to

```
var inner = function(){..}
return inner;
```

Reply ↓

David Shariff *said* on 23/08/2013 at 6:24 pm:

@Apurv

Are you referring to a code example from the blog post? Please provide a little more detail.

With regards to your code sample, the 2 are not the same. During the "Creation Stage":

```
return function inner(){..}
```

will reference the function inner() but:

```
var inner = function(){..}
return inner;
```

at the same stage will be undefined. One is function declaration, the other is an variable assignment.

Reply ↓

Apurv *said* on 26/08/2013 at 9:31 pm:

I was referring to the comment made by Stephan.

You are right. The statements are not equivalent.

But, won't there be a pointer to the function inner created in the foo context's VO? i.e. foo's context VO won't be empty after creation stage has been completed.

This is wrt Stephan's comment "Maybe I am wrong, but to my opinion the foo() context VO is empty (foo has no params, no vars, no function decl.)"

Reply ↓

tapir ***said*** **on** 07/11/2013 at 10:16 pm**:**

best explanation

, thanks man

Reply ↓

Carter ***said*** **on** 02/12/2013 at 10:17 am**:**

This is one of most in depth post on this topic I ever read. Thanks @David for shine on me

Reply ↓

Rick ***said*** **on** **12/12/2013 at 1:41 am:**

What an excellent post. I'd like to ask an intelligent question but … you answered them.

(thanks)

Reply ↓

Robert Green ***said*** **on** **25/12/2013 at 8:18 am:**

Great article to explain closures.

There's a typo: "none blocking manner" should be "non–blocking manner".

Reply ↓

David Shariff ***said*** **on 22/01/2014 at 4:42 am:**

Good catch!

Reply ↓

Agradip ***said*** **on 09/12/2014 at 5:23 pm:**

Very nice article

Reply ↓

ankur ***said*** **on 02/04/2015 at 3:39 am:**

very well explained

Reply ↓

wangrl ***said*** **on 04/05/2015 at 5:12 pm:**

Hi! Thanks for your excellent work! I wonder if there is possible to get the JS function's scope chain by setting hooks inside the V8 engine? If it is, can you telll me the relation function in the source code?

Reply ↓

Sangram ***said*** **on 21/08/2015 at 12:12 am:**

Hi David,

The exact thing I was looking for ‼. Thanks so much.
If I am not wrong the Scope chain is fundamental in understanding the Asynchronous programming in JavaScript , which is single threaded. Do you have an article to explain the Asynchronicity in JavaScript

. I know I am asking for more

, but simply love the way you put it !!

Thanks and kind regards,
S

Reply ↓

Ian *said* on 10/09/2015 at 2:36 am**:**

aha… this was helpful too…. http://dmitryfrank.com/articles/js_closures

Reply ↓

Aswin *said* on 16/08/2016 at 11:54 pm**:**

```
var myAlerts = [];

for (var i = 0; i < 5; i++) {
myAlerts.push(
function inner() {
alert(i);
}
);
}

myAlerts[0](); // 5
myAlerts[1](); // 5
myAlerts[2](); // 5
myAlerts[3](); // 5
myAlerts[4](); // 5
```

I am a little confused with this example since you mentioned the inner function is created in global scope,to be precise how does the push method call deal with the inner function declaration.Is it creating a fresh "inner" function(closure) for every push method call?

Reply ↓

Aswin *said* on 17/08/2016 at 12:01 am:

Shouldn't the push call create a new execution context and thus a new inner function each time ?

Reply ↓

Anon *said* on 18/08/2017 at 5:16 am:

"Shouldn't the push call create a new execution context and thus a new inner function each time ?"

Yes, and it does that. Every time the push method is called it declares a new inner function in its own VO.

The thing is, every single one of those VO's (one for every context created during each of the push method calls) share the same parent context.

That is, despite their VO's being different, every time the interpreter can't resolve a name (in this case, "i") it will go through the scope chain looking for it, and, since all those contexts share the same parents, they will end up finding a reference to the same variable ("i"), declared, in this case, in the global context.

Cheers.

Reply ↓

MirSujat ***said*** **on 26/09/2016 at 6:30 pm:**

Every JS developer must read this… Though it's late but finally i have find my Anwser….) Thanks

Reply ↓

Natasha ***said*** **on 04/01/2017 at 11:50 am:**

Hi, David!
I have problems whrn I leave my comments. I follow the rulls but I don't see my comments in the stack. I have some ideas I want to share.
Thanks!
Regards,
Natasha.

Reply ↓

Natasha ***said*** **on 04/01/2017 at 12:02 pm:**

Hi, David!
I found out some moments in this article I'd like to share with you. In

what way can I do it besides this forum?
Thanks!
Regards,
Natasha.

Reply ↓

S. Neha ***said*** **on** **04/04/2017 at 3:23 am:**

Hi,

Can anyone please help me understand this concept -

The scope chain property of each execution context is simply a collection of the current context's [VO] + all parent's lexical [VO]s..
but from where does this execution context gets the value of parent's lexical [VO] ?

where are all these lexical VO's stored for reference and as what object ? and how do they store for same constructor having multiple instances ?
how to context know which scope-chain to refer…

how will this be explained ? –

```
function module(num) {
var u=num;
return {
changeU: function (val) {
u=val;
},
getU: function () {
console.log(u);
},
}
};
```

```
var a=module(11);
var b=module(19);

a.getU(); //11
b.getU(); //19

a.changeU(8);

a.getU(); //8
b.getU(); //19 — ? shoudn't it be shared or something….
```

Reply ↓

Pradeep ***said*** **on** **05/11/2017 at 3:29 pm:**

I Think here a, b and module are three saperate global variables having its own scope chain so there is no interaction between "u" variable of a and b context.

Thanks for the greate article,
correct me if am wrong………..

Reply ↓

Spike ***said*** **on** **24/12/2017 at 3:49 pm:**

good article, thank!

Reply ↓

anson ***said*** **on** **22/10/2016 at 10:11 am:**

Sorry about typo, the last result should be

```
[[Scopes]]:Scopes[1]
0:Global
```

instead of

[[Scopes]]:Scopes[2]
0:Global

Reply ↓

Leave a Reply

Your email address will not be published. Required fields are marked *

Name *	
Email *	
Website	
[Spam Check] What is: * 1 ×	= one
Comment	

Post Comment

← READ MORE