[adequately good(/)](/)

decent programming advice

written by [ben cherry(http://twitter.com/bcherry)](http://twitter.com/bcherry)

[home(/)](/)
[archives(#)](#)
[about(/about.html)](/about.html)
[feed(/feeds/atom.xml)](/feeds/atom.xml)

posts by year [2009(/2009)](/2009) [2010(/2010)](/2010) [2011(/2011)](/2011)

2010-02-08

[JavaScript Scoping and Hoisting(/JavaScript-Scoping-and-Hoisting.html)](/JavaScript-Scoping-and-Hoisting.html)

Do you know what value will be alerted if the following is executed as a JavaScript program?

```
var foo = 1;
function bar() {
        if (!foo) {
                var foo = 10;
        }
        alert(foo);
}
bar();
```

If it surprises you that the answer is “10”, then this one will probably really throw you for a loop:

```
var a = 1;
function b() {
        a = 10;
        return;
        function a() {}
}
b();
alert(a);
```

Here, of course, the browser will alert “1”. So what’s going on here? While it might seem strange, dangerous, and confusing, this is actually a powerful and expressive feature of the language. I don’t know if there is a standard name for this specific behavior, but I’ve come to like the term “hoisting”. This article will try to shed some light on this mechanism, but first lets take a necessary detour to understand JavaScript’s scoping.

Scoping in JavaScript

One of the sources of most confusion for JavaScript beginners is scoping. Actually, it's not just beginners. I've met a lot of experienced JavaScript programmers who don't fully understand scoping. The reason scoping is so confusing in JavaScript is because it looks like a C-family language. Consider the following C program:

```
#include <stdio.h>
int main() {
        int x = 1;
        printf("%d, ", x); // 1
        if (1) {
                int x = 2;
                printf("%d, ", x); // 2
        }
        printf("%d\n", x); // 1
}
```

The output from this program will be `1, 2, 1`. This is because C, and the rest of the C family, has **block-level scope**. When control enters a block, such as the `if` statement, new variables can be declared within that scope, without affecting the outer scope. This is not the case in JavaScript. Try the following in Firebug:

```
var x = 1;
console.log(x); // 1
if (true) {
        var x = 2;
        console.log(x); // 2
}
console.log(x); // 2
```

In this case, Firebug will show `1, 2, 2`. This is because JavaScript has **function-level scope**. This is radically different from the C family. Blocks, such as `if` statements, **do not** create a new scope. Only functions create a new scope.

To a lot of programmers who are used to languages like C, C++, C#, or Java, this is unexpected and unwelcome. Luckily, because of the flexibility of JavaScript functions, there is a workaround. If you must create temporary scopes within a function, do the following:

```
function foo() {
        var x = 1;
        if (x) {
                (function () {
                        var x = 2;
                        // some other code
                }());
        }
        // x is still 1.
}
```

This method is actually quite flexible, and can be used anywhere you need a temporary scope, not just within block statements. However, I strongly recommend that you take the time to really understand and appreciate JavaScript scoping. It's quite powerful, and one of my favorite features of the language. If you understand scoping, hoisting will make a lot more sense to you.

Declarations, Names, and Hoisting

In JavaScript, a name enters a scope in one of four basic ways:

1. **Language-defined:** All scopes are, by default, given the names `this` and `arguments`.

2. **Formal parameters:** Functions can have named formal parameters, which are scoped to the body of that function.
3. **Function declarations:** These are of the form `function foo() {}`.
4. **Variable declarations:** These take the form `var foo;`.

Function declarations and variable declarations are always moved ("hoisted") invisibly to the top of their containing scope by the JavaScript interpreter. Function parameters and language-defined names are, obviously, already there. This means that code like this:

```
function foo() {
        bar();
        var x = 1;
}
```

is actually interpreted like this:

```
function foo() {
        var x;
        bar();
        x = 1;
}
```

It turns out that it doesn't matter whether the line that contains the declaration would ever be executed. The following two functions are equivalent:

```
function foo() {
        if (false) {
                var x = 1;
        }
        return;
        var y = 1;
}
function foo() {
        var x, y;
        if (false) {
                x = 1;
        }
        return;
        y = 1;
}
```

Notice that the assignment portion of the declarations were not hoisted. Only the name is hoisted. This is not the case with function declarations, where the entire function body will be hoisted as well. But remember that there are two normal ways to declare functions. Consider the following JavaScript:

```
function test() {
        foo(); // TypeError "foo is not a function"
        bar(); // "this will run!"
        var foo = function () { // function expression assigned to local variable 'foo'
                alert("this won't run!");
        }
        function bar() { // function declaration, given the name 'bar'
                alert("this will run!");
        }
}
test();
```

In this case, only the function declaration has its body hoisted to the top. The name ‘foo’ is hoisted, but the body is left behind, to be assigned during execution.

That covers the basics of hoisting, which is not as complex or confusing as it seems. Of course, this being JavaScript, there is a little more complexity in certain special cases.

Name Resolution Order

The most important special case to keep in mind is name resolution order. Remember that there are four ways for names to enter a given scope. The order I listed them above is the order they are resolved in. In general, if a name has already been defined, it is never overridden by another property of the same name. This means that a function declaration takes priority over a variable declaration. This does not mean that an assignment to that name will not work, just that the declaration portion will be ignored. There are a few exceptions:

- The built-in name `arguments` behaves oddly. It seems to be declared following the formal parameters, but before function declarations. This means that a formal parameter with the name `arguments` will take precedence over the built-in, even if it is undefined. This is a bad feature. Don’t use the name `arguments` as a formal parameter.
- Trying to use the name `this` as an identifier anywhere will cause a SyntaxError. This is a good feature.
- If multiple formal parameters have the same name, the one occurring latest in the list will take precedence, even if it is undefined.

Named Function Expressions

You can give names to functions defined in function expressions, with syntax like a function declaration. This does not make it a function declaration, and the name is not brought into scope, nor is the body hoisted. Here’s some code to illustrate what I mean:

```
foo(); // TypeError "foo is not a function"
bar(); // valid
baz(); // TypeError "baz is not a function"
spam(); // ReferenceError "spam is not defined"

var foo = function () {}; // anonymous function expression ('foo' gets hoisted)
function bar() {}; // function declaration ('bar' and the function body get hoisted)
var baz = function spam() {}; // named function expression (only 'baz' gets hoisted)

foo(); // valid
bar(); // valid
baz(); // valid
spam(); // ReferenceError "spam is not defined"
```

How to Code With This Knowledge

Now that you understand scoping and hoisting, what does that mean for coding in JavaScript? The most important thing is to always declare your variables with a `var` statement. I **strongly** recommend that you have *exactly one* `var` statement per scope, and that it be at the top. If you force yourself to do this, you will never have hoisting-related confusion. However, doing this can make it hard to keep track of which variables have actually been declared in the current scope. I recommend using **JSLint(http://www.jslint.com)** with the `onevar` option to enforce this. If you’ve done all of this, your code should look something like this:

```
/*jslint onevar: true [...] */
function foo(a, b, c) {
    var x = 1,
        bar,
        baz = "something";
}
```

What the Standard Says

I find that it's often useful to just consult the **ECMAScript Standard (pdf)(http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf)** directly to understand how these things work. Here's what it has to say about variable declarations and scope (section 12.2.2 in the older version):

> *If the variable statement occurs inside a FunctionDeclaration, the variables are defined with function-local scope in that function, as described in section 10.1.3. Otherwise, they are defined with global scope (that is, they are created as members of the global object, as described in section 10.1.3) using property attributes { DontDelete }. Variables are created when the execution scope is entered. A Block does not define a new execution scope. Only Program and FunctionDeclaration produce a new scope. Variables are initialised to undefined when created. A variable with an Initialiser is assigned the value of its AssignmentExpression when the VariableStatement is executed, not when the variable is created.*

I hope this article has shed some light on one of the most common sources of confusion to JavaScript programmers. I have tried to be as thorough as possible, to avoid creating more confusion. If I have made any mistakes or have large omissions, please let me know.

filed under **javascript(/tag/javascript)**

171 Comments **Adequately Good** 1 **Login**

Recommend 179 Tweet Share **Sort by Oldest**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Awesome Bob • 9 years ago

Wow, this was a real eye opener for me. Thanks for this article, it will make developing complex objects less troublesome for me!

55 • Reply • Share ›

Shehi Awesome Bob • 5 years ago

"Eyeopener" is an understatement here :) This language is proving to "have been an underestimated and misunderstood language" by the day.

3 • Reply • Share ›

Mark van Gulik Shehi • 5 years ago

I believe you misspelled both "overestimated" and "misdesigned".

29 • Reply • Share ›

Eric Schleicher Mark van Gulik • 4 years ago

+10

• Reply • Share ›

Aleksandr Strutynskyy Mark van Gulik • 4 years ago

you can see a frustrated java developer or inexperienced js developer right there. you should probably listen Douglas Crockford on javascript, specially on "misdesigned" part.

1 • Reply • Share ›

Okay Aleksandr Strutynskyy • 4 years ago

Sure, let's go ask the guy who designed it for an 'unbiased' opinion on JavaScript, "The World's Most Misunderstood Programming Language" - which is a Crockford article.

"It was originally called LiveScript, but that name wasn't confusing enough", "JavaScript's C-like syntax, including curly braces and the clunky for statement, makes it appear to be an ordinary procedural language. This is misleading because JavaScript has more in common with functional languages like Lisp or Scheme than with C or Java." So why try copy the name and syntax?

"The first versions of JavaScript were quite weak. They lacked exception handling, inner functions, and inheritance."

"JavaScript has its share of design errors, such as the overloading of + to mean both addition and concatenation with type coercion, and the error-prone with statement should be avoided. The reserved word policies are much too strict. Semicolon insertion was a huge mistake, as was the notation for literal regular expressions. These mistakes have led to

see more

3 • Reply • Share ›

Okay Okay • 4 years ago

*later helped design, haha ;)

• Reply • Share ›

David Okay • 3 years ago

Crockford didn't design JavaScript, Brendan Eich did.

The spirit of what you wrote it still OK, I think, but you'll want to brush up on your history if you want to convince anyone.

3 • Reply • Share ›

waht mmm Mark van Gulik • a year ago

and initialized

• Reply • Share ›

Jordan Boesch • 9 years ago

All truly great stuff that a JavaScript programmer should know! Reminds me of another good

article Nicholas Zakas wrote: http://www.nczonline.net/bl...
5 • Reply • Share ›

Ben Cherry Mod Jordan Boesch • 9 years ago
Yeah, that quiz, and the in-depth response, reminded me that there was a LOT of confusion about these fundamental mechanisms out there, and were part of the inspiration for putting together a detailed article about it. Glad you liked it!
4 • Reply • Share ›

Santosh K. Tripathi Jordan Boesch • 4 years ago
very good .... http://www.nczonline.net/blog/...
• Reply • Share ›

azoff • 9 years ago
Good work Ben, very well thought out and written. To be honest, I don't even deal with variable declarations that often, instead, I just add my variable declarations to my private signature implementation. For instance:

say my spec says "this function takes an array and returns the sum of all the element" and the public signature looks like this:

function addAll(arr)

Then I would implement it as such:

```
function addAll(arr, i, sum) {
sum = 0;
for (i in arr) {
sum += arr[i];
}
return sum;
}
```

I find that this technique helps avoid hoisting problems because all of your declarations are part of your signature. It also helps with memory management as garbage collection works exceptionally well with function arguments and you won't run into memory leak problems (ahem... IE variables). The beauty of it all is that it is also unobtrusive thanks to the fact that JavaScript does not enforce signatures. That being said, I highly solicit learning the way the JavaScript engine works because little tricks like this one are no substitute for a concrete understanding of the language parser.

Good work Ben!
8 • Reply • Share ›

Ben Cherry Mod azoff • 9 years ago
Well that's an interesting approach to local variables. Not sure I'm on board with using that pattern, I'd need more concrete information about how garbage collection works.

Regardless, I'd think it would be much cleaner and safer to do something like this:
```
var addAll = (function(i, sum) {
return function (arr) {
```

```
sum = 0;
for (i in arr) {
sum += arr[i];
}
};
}());
```

But that approach is probably not great for memory if the function is not used very often, because the execution context of the anonymous function has to be kept around so the inner function still has access to those variables in its closure. If it were executed a lot, though, it might actually be better since each time addAll() is invoked, you only need to put one variable in the new execution context, while the other "local" variables are continually reused from the old one.

Hmmm, needs more investigation.

1 ^ | v • Reply • Share ›

berndartmueller ➦ Ben Cherry • 9 years ago

and why do you ned the outer self executing function? The variable "i" is in the global window object, but "sum" isn't. Is this solution better than just using the inner function?

1 ^ | v • Reply • Share ›

Ben Cherry Mod ➦ berndartmueller • 9 years ago

That was just a cleaner way of implementing azoff's idea of never using local variables, just using un-intialized extra function parameters. My method preserved the .length property of the addAll function. I don't think this is worthwhile, just a curiosity. I'm sticking to normal locals :)

^ | v • Reply • Share ›

solostyle ➦ azoff • 5 years ago

Thank you, azoff! I wanted more info on how exactly you rig the private function to handle different public signatures, and I found this.
https://github.com/azoff/lo...
I can see how this could be confusing when used with a team of developers. I post this link here if anyone else wanted to see your implementation.

I've been reading a detailed explanation of execution context objects written by Mikowski and Powell in Single Page Web Applications (2014). I highly solicit (love that word) perusing chapter two of the book which discusses garbage collection and many other deep js workings.

^ | v • Reply • Share ›

skim • 9 years ago

I have a question regarding variable declaration for inner functions. Should I declare them in the outside function?

```
function outer() {
var inner,
i;
// put inner's variable declaration here?
```

inner = function() {
// put variable declaration here?
}
}
1 ^ | v • Reply • Share ›

skim • 9 years ago
Nevermind, it looks like I should declare variables for inner function on the outside function due to hoisting

var outer = function() {
var inner;
alert(typeof inside_var);
inner = function() {
var inside_var = 4;
}
alert(typeof inside_var);
}

// undefined
// undefined
^ | v • Reply • Share ›

Tri • 9 years ago
I don't understand why it's 1 in the second example from the top. Can you explain?
^ | v • Reply • Share ›

skim ➦ Tri • 9 years ago
function a() {} is hoisted to the top of function b() and since 'a' cannot be changed after defined as a function, a = 10 is not valid. therefore, a = 1
^ | v • Reply • Share ›

Ben Cherry Mod ➦ skim • 9 years ago
Actually, that's not quite correct. The `function a(){}` does scope `a` a to the function `b`, but that `a = 10` does work to change it. Thus, after `b` finishes, its local `a` has the value `10`. However, because that `a` was local (due to the function declaration), the global `a` is unchanged, thus `1`. Hope that clears it up!
10 ^ | v • Reply • Share ›

skim ➦ Ben Cherry • 9 years ago
Oops. It looks like I'm rusty in my JavaScript. Thanks for the correction.
^ | v • Reply • Share ›

Clare Sudbery ➦ Ben Cherry • 7 years ago
I'm still a bit confused... you say "a function declaration takes priority over a variable declaration. This does not mean that an assignment to that name will not work, just that the declaration portion will be ignored."

So in the context of that second example, within the scope of b(), 'a' refers to a function rather than a variable... but it can still be assigned the value 10? So what impact is the function declaration having in this case?

Reply • Share ›

Demuynck_ruben Clare Sudbery • 7 years ago

Due to the way scoping apparently works in javascript, you can - if i understand correctly, have 2 variables with the same name, in a different scope. a in the global scope is 1, a in the function scope is 10. The clue is that they are different variables, because they are declared in a different scope.

Reply • Share ›

Eason Clare Sudbery • 5 years ago

" if a name has already been defined, it is never overridden by another property of the same name. "

Reply • Share ›

MK Safi Ben Cherry • 6 years ago

So, `function a() {}` inside "b" acts exactly as if you put `var a;` at the top of "b"...Well, not necessarily the top of "b", but anywhere in "b", because of hoisting!

2 • Reply • Share ›

rtoal • 9 years ago

Excellent post. The quote from the ES3 standard is puzzling, though -- making it appear that function scope is only created for variables introduced in a VariableStatement inside a FunctionDeclaration, while all other VariableStatements introduce globally scoped variables. Browsers give function scope to variables defined in function expressions, though, no? For example:

```
var a = 2;
var f = function (x) {
var a = 5;
alert(a);
};
f();
alert(a);
```

alerts 5 then 2. Is the ES3 remark incorrect?

Reply • Share ›

Ben Cherry Mod rtoal • 9 years ago

Yeah, that looks like a mistake in the standard. Good catch. A FunctionExpression will create a new scope, just like a FunctionDeclaration or a Program.

Reply • Share ›

Null • 8 years ago

Thank you for posting this - an amazing article. Imagine the feelings of terror running through my brain as I think about all the JavaScript code written with the assumption of block-level scoping hanging out in my client's tree. Zut alors!

1 • Reply • Share ›

greggman • 8 years ago

This is a good post but I feel like in the end it's missing the point.

The reason JavaScript is like this is because it makes it massively more powerful than C, C++, Java, C# etc when it comes to asynchronous usage and callbacks. Because stuff is scoped by function it's very easy for a callback to access variables that were in scope at the time the callback was setup. This is in contrast to C, C++, Java or C# where if you want a callback to have access to some data you have to pass that data into the callback in some form, usually by having to go to the trouble of creating a class to pass the data in or subclassing some class and overriding onXXX type functions.

None of that is needed because of the scoping rules of JavaScript. Let's say I want to load some images and append them too some element in my document. I don't want to insert the image tags until the images are loaded because otherwise they'll show up as broken images or if I have some fancy CSS borders there'll be a bunch of empty borders until the images are loaded. In JavaScript here's some code

```
function LoadAndInsertImages(imageUrls, elementToInsertIn) {
for (var ii = 0; ii < imageUrls.length; ++ii) {
var img = new Image();
```

see more

3 • Reply • Share ›

Ben Cherry Mod ➦ greggman • 8 years ago

Yes! Closure scope is the best feature of JavaScript. However, the hoisting behavior still confuses lots of JS programmers, even those who get closure scope. I wrote this to explain those mechanisms, not to knock the language for being awesome :)

1 • Reply • Share ›

Clare Sudbery ➦ greggman • 7 years ago

Me again. Apologies, but I'm new to JS, and I'm still getting my head around it. I know this is a bit of a tangent to your original article, but this example highlights some struggles I'm currently having with the way execution and declaration seem to get mixed up in JavaScript.
When LoadAndInsertImages is executed, there is an elementToInsertIn object being passed into it. But at this point the onload function is only declared - it is not executed. So what happens to the elementToInsertIn object in the meantime? Is it stored somewhere along with the function definition, ready to be used when onload is finally executed as a callback?
I'm also really confused by the fact that onload is a function itself, but when it executes, all it does is create and return yet another function. What then happens to that new function, and when is it executed?

• Reply • Share ›

Charles Wicksteed ➦ Clare Sudbery • 7 years ago

4 img.onload = function(img2) {

```
5 return function() {
6 elementToInsertIn.appendChild(img2);
7 }
8 }(img);
```

I have taken the liberty of changing the name of the formal parameter from img to img2, for clarity.
When the "onload = ..." line is executed, the first function, starting on line 4, is executed. The "(img)" on line 8 makes it do this.
The first function returns the inner function (which is, as you state, not executed at this time) which is assigned to the "onload" property. The inner function runs after the image is loaded.
You are right that the elementToInsertIn object is stored somewhere along with the function definition. This is called a closure -- JavaScript is a much more powerful language than beginners are led to believe.
Coming back to the fact that the outer function is executed at the time that line 4 runs:
consider the difference between "setTimeout(f, 100)" and "setTimeout(f(), 100)". The first passes the "f" function to setTimeout, the second causes the funtion "f" to run, and its return value is passed to setTimeout. That is, adding "()" or "(img)" to the end causes the function to be run then and there.

• Reply • Share ›

shane Charles Wicksteed • 5 years ago
Is there any reason this couldn't have just been coded as:

```
img.onload = function() {
elementToInsertIn.appendChild(img);
}
```

• Reply • Share ›

Daria Ezhova shane • 5 years ago
In your example when it comes to executing onload function (when any image is loaded) it will be executed with the current 'img' value in the cycle, whatever it will be that time. You have to 'close' img value for each cycle step, i.e. save it in a closure.

• Reply • Share ›

Jjpcondor greggman • 7 years ago
smart remarks

• Reply • Share ›

David greggman • 3 years ago
Closure has _nothing_ to do with function scoping or hoisting, neither of which adds any value to the language (vs. block scope). In this regard JavaScript would have been better off emulating other languages.

• Reply • Share ›

whootboy • 8 years ago

...okay I didn't knew the C-Behaviour, so I never understood why scoping is so "tricky" to some people. I knew it only that way.

1 • Reply • Share ›

Kabindra Bakey • 8 years ago

Wow, this was a real eye opener for me. Thanks for this article,

• Reply • Share ›

Ankur Oberoi • 8 years ago

awesome awesome

• Reply • Share ›

Farrukh Momin • 8 years ago

speechless

• Reply • Share ›

David Rivers • 8 years ago

I saw the term "hoisting" in another blog, scratched my head, found this, and now I'm no longer scratching my head :) Thanks!

• Reply • Share ›

Sean • 8 years ago

If you don't teach, you should. Very clear. Wish my college professors had been that clear. :-\

18 • Reply • Share ›

Alexandre Morgaut • 8 years ago

I also use the "one var" rule of JSLint but I go a step further...
In your
example/*jslint onevar: true [...] */ function foo(a, b, c) {   var x = 1,   bar,   baz = "someth
the variables are assigned at the declaration level which doesn't match as you said to the real JS behavior

I prefer this convention
```
/*jslint onevar: true [...] */
function foo(a, b, c) {
    var x,
        bar,
        baz;  x = 1;    baz = "something"; }
```
I try to start with var declarations and function declaration before anything else ;-)

• Reply • Share ›

Ben Cherry Mod → Alexandre Morgaut • 8 years ago

That's a fine convention too. I think it's perhaps a bit overly verbose, but you're right that it's clearer in regards to the actual interpreter behavior.

• Reply • Share ›

Pete Schaffner • 8 years ago

This article finally drove these concepts home for me. Well done and thank you!

This article finally drove these concepts home for me. Well done and thank you!

• Reply • Share ›

mrrena • 8 years ago

Excellent article. I link to it from my blog entry http://mrrena.blogspot.com/...

• Reply • Share ›

Jeffrey Morin • 8 years ago

I don't quite get this statement. "In general, if a name has already been defined, it is never overridden by another property of the same name." I thought in JavaScript overwriting other declarations was simple and if you didn't have a grasp on variables and scoping that it can become a real issue. Can anyone elaborate on this a little?

• Reply • Share ›

Esben Søvang Maaløe Jeffrey Morin • 7 years ago

Yep - me too. I don't get the point of that passage!

• Reply • Share ›

Load more comments

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Disqus' Privacy PolicyPrivacy PolicyPrivacy**

the author

Ben is a 25 year-old software engineer. He lives and works in San Francisco. Many people think he invented the term "hoisting" in JavaScript, but this is untrue.

- **Twitter(http://twitter.com/bcherry)** - @**bcherry(http://twitter.com/bcherry)**
- **GitHub(http://github.com/bcherry)** - My Code
- **LinkedIn(http://www.linkedin.com/in/bcherryprogrammer)** - Professional Profile
- **Facebook(http://www.facebook.com/bcherry)** - That Other Social Network
- **Presentations(http://www.bcherry.net/talks/)** - Slides From My Talks

Follow @bcherry 7,486 followers

categories

- **javascript(/tag/javascript)** (21)

- social gaming(/tag/social%20gaming) (1)
- css(/tag/css) (1)
- jquery(/tag/jquery) (2)
- performance(/tag/performance) (5)
- tools(/tag/tools) (2)
- html5(/tag/html5) (3)
- adequatelygood(/tag/adequatelygood) (1)
- timers(/tag/timers) (2)
- module pattern(/tag/module%20pattern) (3)
- talks(/tag/talks) (1)
- slide(/tag/slide) (1)
- python(/tag/python) (1)
- debugging(/tag/debugging) (1)
- testing(/tag/testing) (2)
- hashbang(/tag/hashbang) (1)