JavaScript function order: why does it matter?

Ask Question

Original Question:

79

JSHint complains when my JavaScript calls a function that is defined further down the page than the call to it. However, my page is for a game, and no functions are called until the whole thing has downloaded. So why does the order functions appear in my code matter?

EDIT: I think I may have found the answer.

67

http://www.adequatelygood.com/2010/2/JavaScript-Scoping-and-Hoisting

I am groaning inside. Looks like I need to spend ANOTHER day re-ordering six thousand lines of code. The learning curve with javascript is not steep at all, but it is very looooooong.

javascript function order jslint jshint

share improve this question

edited Sep 30 '11 at 13:34
Incognito
13.9k 9 58 108

asked Sep 30 '11 at 10:45
Chris Tolworthy
746 2 11 22

+1 for the excellent reference in the update. And I hope that convinces you that you don't really need to re-order your code. :) – awm Sep 30 '11 at 12:54

add a comment

3 Answers

active oldest votes

Edit: For an overview which also covers some ES6 declarations (`let` , `const`): https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Scope_Cheatsheet

223

This weird behavior depends on

1. How you define the functions and
2. When you call them.

+600

Here's some examples.

```
bar(); //This won't throw an error
function bar() {}

foo(); //This will throw an error
var foo = function() {}
```

```
----

bar();
function bar() {
    foo(); //This will throw an error
}
var foo = function() {}

----

bar();
function bar() {
    foo(); //This _won't_ throw an error
}
function foo() {}

---

function bar() {
    foo(); //no error
}
var foo = function() {}
bar();
```

This is because of something called **hoisting**!

There are two ways to define functions: Function *declaration* and function *expression*. The difference is annoying and minute, so let's just say this slightly wrong thing: If you're writing it like `function name() {}`, it's a **declaration**, and when you write it like `var name = function() {}` (or an anonymous function assigned to a return, things like that), it's a function **expression**.

First, let's look at how variables are handled:

```
var foo = 42;

//the interpreter turns it into this:
var foo;
foo = 42;
```

Now, how function *declarations* are handled:

```
var foo = 42;
function bar() {}

//turns into
var foo; //Insanity! It's now at the top
function bar() {}
foo = 42;
```

The `var` statements "throws" the *creation* of `foo` to the very top, but doesn't assign the value to it yet. The function declaration comes next in line, and finally a value is assigned to `foo`.

And what about this?

```
bar();
var foo = 42;
```

```
function bar() {}
//=>
var foo;
function bar() {}
bar();
foo = 42;
```

Only the *declaration* of `foo` is moved to the top. The assignment comes only after the call to `bar` is made, where it was before all the hoisting occurred.

And finally, for conciseness:

```
bar();
function bar() {}
//turns to
function bar() {}
bar();
```

Now, what about function *expressions*?

```
var foo = function() {}
foo();
//=>
var foo;
foo = function() {}
foo();
```

Just like regular variables, first `foo` is *declared* at the highest point of the scope, then it is assigned a value.

Let's see why the second example throws an error.

```
bar();
function bar() {
    foo();
}
var foo = function() {}
//=>
var foo;
function bar() {
    foo();
}
bar();
foo = function() {}
```

As we've seen before, only the creating of `foo` is hoisted, the assignment comes where it appeared in the "original" (un-hoisted) code. When `bar` is called, it is before `foo` is assigned a value, so `foo === undefined`. Now in the function-body of `bar`, it's as if you're doing `undefined()`, which throws an error.

tl;dr If you're not calling anything until everything loads, you should be fine.

share improve this answer

edited Oct 24 '14 at 6:38

answered Sep 30 '11 at 13:13

Zirak

21.6k 9 55 85

6 Many thanks! I do appreciate all the time you put in. (and incognito for his, er, incognito edits) – Chris Tolworthy Sep 30 '11 at 16:15

Sorry to dig this up, but are overloads like Array.prototype.someMethod = function(){} hoisted? I seem to be getting errors if these types of things are at the ends of my script. – Edge Sep 9 '14 at 4:58

Examples for the win! I grok them so much quicker than pure textual explanations. Thanks! – Joshua PinterMar 25 '16 at 20:57

add a comment

5

The main reason is probably that JSLint does only one pass on the file so it doesn't know you *will*define such a function.

If you used functions statement syntax

```
function foo(){ ... }
```

There is actually no difference at all where you declare the function (it always behaves as if the declaration is on the beginning).

On the other hand, if your function was set like a regular variable

```
var foo = function() { ... };
```

You have to guarantee you wont call it before the initialization (this can actually be a source of bugs).

Since reordering tons of code is complicated and can be a source of bugs in itself, I would suggest you search for a workaround. I'm pretty sure you can tell JSLint the name of global variables beforehand so it doesn't complain about undeclared stuff.

Put a comment on the beggining of the file

```
/*globals foo1 foo2 foo3*/
```

Or you can use a text box there for that. (I also think you can pass this in the arguments to the inner jslint function if you can meddle with it.)

share improve this answer

edited Jan 23 '15 at 14:15

mmoore

442 4 17

answered Sep 30 '11 at 13:14

hugomg

45.7k 11 102 186

Thanks. So the /* globals */ line will work? Good - anything to get JsHint to like me. I am still new to JavaScript and get inexplicable pauses when I refresh a page, yet no reported bugs. So I figured that the solution was to play by all the rules and then see if it still happens. – Chris Tolworthy Sep 30 '11 at 16:14

add a comment

There are way too many people pushing arbitrary rules about how JavaScript should be written.

3 Most rules are utter rubbish.

Function hoisting is a feature in JavaScript because it is a good idea.

When you have an internal function which is often the utility of inner functions, adding it to the beginning of the outer function is an acceptable style of writing code, but it does have the drawback that you have to read through the details to get to what the outer function does.

You should stick to one principle throughout your codebase either put private functions first or last in your module or function. JSHint is good for enforcing consistency, but you should ABSOLUTELY adjust the .jshintrc to fit your needs, NOT adjust your source code to other peoples wacky coding concepts.

One coding style that you might see in the wild you should avoid because it gives you no advantages and only possible refactoring pain:

```
function bigProcess() {
    var step1,step2;
    step1();
    step2();

    step1 = function() {...};
    step2 = function() {...};
}
```

This is exactly what function hoisting is there to avoid. Just learn the language and exploit its strengths.

share improve this answer

edited Apr 19 '15 at 10:00

answered Apr 18 '15 at 14:10

Henrik Vendelbo

71 2