

var functionName = function() {} vs function functionName() {}

Asked 10 years, 10 months ago Active 24 days ago Viewed 969k times

I've recently started maintaining someone else's JavaScript code. I'm fixing bugs, adding features and also trying to tidy up the code and make it more consistent.

6592

The previous developer uses two ways of declaring functions and I can't work out if there is a reason behind it or not.

The two ways are:



2434

```
var functionOne = function() {  
    // Some code  
};
```

```
function functionTwo() {  
    // Some code  
}
```

What are the reasons for using these two different methods and what are the pros and cons of each? Is there anything that can be done with one method that can't be done with the other?

javascript

function

syntax

idioms

edited Apr 15 at 17:45

asked Dec 3 '08 at 11:31



Richard Garside

42.5k 9 66 78

193 permadi.com/tutorial/jsFunc/index.html is very good page about javascript functions – uzay95 Apr 9 '10 at 11:51

64 Related is this *excellent* article on [Named Function Expressions](#). – Phrogz Apr 21 '11 at 21:30

12 @CMS references this article: kangax.github.io/nfe/#expr-vs-decl – Upperstage Aug 3 '11 at 14:18

00 There are two things you need to be aware of: #1 In JavaScript, declarations are hoisted. Meaning that var a = 1; var b = 2; becomes var a;

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

declaration, it gets put at the top of the scope. #2 functionTwo lets you access the name property and that helps a lot when trying to debug something. – [xaviern02](#) Aug 21 '12 at 16:20

- 61 Oh and btw, the correct syntax is with a ";" after the assignation and without after the declaration. E.g. `function f(){} vs var f = function() {};` . – [xaviern02](#) Aug 21 '12 at 16:27

38 Answers

1 2 next

4874



The difference is that `functionOne` is a function expression and so only defined when that line is reached, whereas `functionTwo` is a function declaration and is defined as soon as its surrounding function or script is executed (due to [hoisting](#)).

For example, a function expression:

```
// TypeError: functionOne is not a function
functionOne();

var functionOne = function() {
  console.log("Hello!");
};
```

Run code snippet

Hide results

[Full page](#)

```
Error: {
  "message": "Uncaught TypeError: functionOne is not a function",
  "filename": "https://stacksnippets.net/js",
  "lineno": 13,
  "colno": 1
}
```

13:48:26.047

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

And, a function declaration:

```
// Outputs: "Hello!"
functionTwo();

function functionTwo() {
  console.log("Hello!");
}
```

Run code snippet

[Expand snippet](#)

This also means you can't conditionally define functions using function declarations:

```
if (test) {
  // Error or misbehavior
  function functionThree() { doSomething(); }
}
```

The above actually defines `functionThree` irrespective of `test`'s value — unless `use strict` is in effect, in which case it simply raises an error.

edited May 15 '18 at 9:03



Sudhanshu Vishnoi
651 1 7 22

answered Dec 3 '08 at 11:37



Greg
267k 48 343 322

610 @Greg: By the way, the difference isn't only that they are parsed at different times. Essentially, your `functionOne` is merely a variable that has an anonymous function assigned to it, whereas `functionTwo` is actually a named function. Call `.toString()` on both to see the difference. This is significant in some cases where you want to get the name of a function programmatically. — [Jason Bunting](#) Aug 5 '11 at 21:18

5 @Jason Bunting.. not sure what you're getting at here, `.toString()` seems to return essentially the same value (the function definition) for both: [cl.ly/2a2C2Y1r0J451o0q0B1B](#) — [Jon z](#) Sep 9 '11 at 16:29

124 There are both different. The first one is a `function expression` the second one is a `function declaration`. You can read more on the topic here: [javascriptweblog.wordpress.com/2010/07/06/...](#) — [Michal Kuklis](#) Nov 14 '11 at 6:03

125 @Greg The part of your answer concerning parse time vs. run time is not correct. In JavaScript, function declarations are not defined during parse time, but during run time. The process goes like this: Source code is parsed -> JavaScript program is evaluated -> The global execution context is

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

101 The terminology for this phenomenon is known as hoisting. – Colin Pear Jan 3 '13 at 7:22

1901 First I want to correct Greg: `function abc(){}` is scoped too — the name `abc` is defined in the scope where this definition is encountered. Example:

```
function xyz(){
  function abc(){};
  // abc is defined here...
}
// ...but not here
```

Secondly, it is possible to combine both styles:

```
var xyz = function abc(){};
```

`xyz` is going to be defined as usual, `abc` is undefined in all browsers but Internet Explorer — do not rely on it being defined. But it will be defined inside its body:

```
var xyz = function abc(){
  // xyz is visible here
  // abc is visible here
}
// xyz is visible here
// abc is undefined here
```

If you want to alias functions on all browsers, use this kind of declaration:

```
function abc(){};
var xyz = abc;
```

In this case, both `xyz` and `abc` are aliases of the same object:

```
console.log(xyz === abc); // prints "true"
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```
function abc(){};
console.log(abc.name); // prints "abc"
```

its name is automatically assigned. But when you define it like

```
var abc = function(){};
console.log(abc.name); // prints ""
```

its name is empty — we created an anonymous function and assigned it to some variable.

Another good reason to use the combined style is to use a short internal name to refer to itself, while providing a long non-conflicting name for external users:

```
// Assume really.long.external.scoped is {}
really.long.external.scoped.name = function shortcut(n){
  // Let it call itself recursively:
  shortcut(n - 1);
  // ...
  // Let it pass itself as a callback:
  someFunction(shortcut);
  // ...
}
```

In the example above we can do the same with an external name, but it'll be too unwieldy (and slower).

(Another way to refer to itself is to use `arguments.callee`, which is still relatively long, and not supported in the strict mode.)

Deep down, JavaScript treats both statements differently. This is a function declaration:

```
function abc(){}
```

abc here is defined everywhere in the current scope:

```
// We can call it here
abc(); // Works

// Yet, it is defined down there.
function abc(){}
// ...
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```
// We can call it again  
abc(); // Works
```

Also, it hoisted through a `return` statement:

```
// We can call it here  
abc(); // Works  
return;  
function abc(){}
```

This is a function expression:

```
var xyz = function(){};
```

xyz here is defined from the point of assignment:

```
// We can't call it here  
xyz(); // UNDEFINED!!!  
  
// Now it is defined  
xyz = function(){}  
  
// We can call it here  
xyz(); // works
```

Function declaration vs. function expression is the real reason why there is a difference demonstrated by Greg.

Fun fact:

```
var xyz = function abc(){};  
console.log(xyz.name); // Prints "abc"
```

Personally, I prefer the "function expression" declaration because this way I can control the visibility. When I define the function like

```
var abc = function(){};
```

I know that I defined the function locally. When I define the function like

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

I know that I defined it globally providing that I didn't define `abc` anywhere in the chain of scopes. This style of definition is resilient even when used inside `eval()`. While the definition

```
function abc(){};
```

depends on the context and may leave you guessing where it is actually defined, especially in the case of `eval()` — the answer is: It depends on the browser.

edited Oct 10 '16 at 20:38



Merlin

3,943 2 22 45

answered Dec 3 '08 at 17:43



Eugene Lazutkin

39.8k 8 44 55

66 I refer to RoBorg but he is nowhere to be found. Simple: `RoBorg === Greg`. That's how history can be rewritten in the age of internet. ;-) — [Eugene Lazutkin](#) Jul 26 '09 at 2:52

10 `var xyz = function abc(){}; console.log(xyz === abc);` All browsers I've tested (Safari 4, Firefox 3.5.5, Opera 10.10) gives me "Undefined variable: abc". — [NVI](#) Dec 3 '09 at 17:43

7 Overall I think this post does a good job of explaining the differences and the advantages of utilizing the function declaration. I'll agree to disagree as far as the benefits of utilizing function expression assignments to a variable especially since the "benefit" seems to be an advocacy of declaring a global entity... and everyone knows that you shouldn't clutter the global namespace, right? ;-) — [natlee75](#) Oct 8 '13 at 16:30

80 imo a huge reason to use named function is because debuggers can use the name to help you make sense of your call stack or stack trace. it sucks when you look at the call stack and see "anonymous function" 10 levels deep... — [goat](#) Jan 26 '14 at 18:25

3 `var abc = function(){}; console.log(abc.name);` does not produce "" any more, but "abc" instead. — [Qwerty](#) May 4 '18 at 12:49



604



+200

Here's the rundown on the standard forms that create functions: *(Originally written for another question, but adapted after being moved into the canonical question.)*

Terms:

- **ES5:** [ECMAScript 5th edition](#), 2009
- **ES2015:** [ECMAScript 2015](#) (also known as "ES6")

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

- Function Declaration
- "Anonymous" function Expression (*which despite the term, sometimes create functions with names*)
- Named function Expression
- Accessor Function Initializer (ES5+)
- Arrow Function Expression (ES2015+) (*which, like anonymous function expressions, don't involve an explicit name, and yet can create functions with names*)
- Method Declaration in Object Initializer (ES2015+)
- Constructor and Method Declarations in class (ES2015+)

Function Declaration

The first form is a *function declaration*, which looks like this:

```
function x() {  
    console.log('x');  
}
```

A function declaration is a *declaration*; it's not a statement or expression. As such, you don't follow it with a `;` (although doing so is harmless).

A function declaration is processed when execution enters the context in which it appears, **before** any step-by-step code is executed. The function it creates is given a proper name (`x` in the example above), and that name is put in the scope in which the declaration appears.

Because it's processed before any step-by-step code in the same context, you can do things like this:

```
x(); // Works even though it's above the declaration  
function x() {  
    console.log('x');  
}
```

Until ES2015, the spec didn't cover what a JavaScript engine should do if you put a function declaration inside a control structure like `try`, `if`, `switch`, `while`, etc., like this:

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.


```

    }
    // <===== BE DRAGONS
}

```

And since they're processed *before* step-by-step code is run, it's tricky to know what to do when they're in a control structure.

Although doing this wasn't *specified* until ES2015, it was an *allowable extension* to support function declarations in blocks. Unfortunately (and inevitably), different engines did different things.

As of ES2015, the specification says what to do. In fact, it gives three separate things to do:

1. If in loose mode *not* on a web browser, the JavaScript engine is supposed to do one thing
2. If in loose mode on a web browser, the JavaScript engine is supposed to do something else
3. If in *strict* mode (browser or not), the JavaScript engine is supposed to do yet another thing

The rules for the loose modes are tricky, but in *strict* mode, function declarations in blocks are easy: They're local to the block (they have *block scope*, which is also new in ES2015), and they're hoisted to the top of the block. So:

```

"use strict";
if (someCondition) {
    foo();                // Works just fine
    function foo() {
    }
}
console.log(typeof foo); // "undefined" (`foo` is not in scope here
                        // because it's not in the same block)

```

"Anonymous" function Expression

The second common form is called an *anonymous function expression*:

```

var y = function () {
    console.log('y');
};

```

Like all expressions, it's evaluated when it's reached in the step-by-step execution of the code.

In ES5, the function this creates has no name (it's anonymous). In ES2015, the function is assigned a name if possible by inferring it from context. In the example above, the name would be `foo`. Something similar is done when the function is the value of a property:

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

Named function Expression

The third form is a *named function expression* ("NFE"):

```
var z = function w() {  
    console.log('zw')  
};
```

The function this creates has a proper name (`w` in this case). Like all expressions, this is evaluated when it's reached in the step-by-step execution of the code. The name of the function is *not* added to the scope in which the expression appears; the name *is* in scope within the function itself:

```
var z = function w() {  
    console.log(typeof w); // "function"  
};  
console.log(typeof w);    // "undefined"
```

Note that NFEs have frequently been a source of bugs for JavaScript implementations. IE8 and earlier, for instance, handle NFEs [completely incorrectly](#), creating two different functions at two different times. Early versions of Safari had issues as well. The good news is that current versions of browsers (IE9 and up, current Safari) don't have those issues any more. (But as of this writing, sadly, IE8 remains in widespread use, and so using NFEs with code for the web in general is still problematic.)

Accessor Function Initializer (ES5+)

Sometimes functions can sneak in largely unnoticed; that's the case with *accessor functions*. Here's an example:

```
var obj = {  
    value: 0,  
    get f() {  
        return this.value;  
    },  
    set f(v) {  
        this.value = v;  
    }  
};  
console.log(obj.f);           // 0  
console.log(typeof obj.f);    // "number"
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

You can also create accessor functions with `Object.defineProperty`, `Object.defineProperties`, and the lesser-known second argument to `Object.create`.

Arrow Function Expression (ES2015+)

ES2015 brings us the *arrow function*. Here's one example:

```
var a = [1, 2, 3];
var b = a.map(n => n * 2);
console.log(b.join(", ")); // 2, 4, 6
```

See that `n => n * 2` thing hiding in the `map()` call? That's a function.

A couple of things about arrow functions:

1. They don't have their own `this`. Instead, they *close over* the `this` of the context where they're defined. (They also close over arguments and, where relevant, `super`.) This means that the `this` within them is the same as the `this` where they're created, and cannot be changed.
2. As you'll have noticed with the above, you don't use the keyword `function`; instead, you use `=>`.

The `n => n * 2` example above is one form of them. If you have multiple arguments to pass the function, you use parens:

```
var a = [1, 2, 3];
var b = a.map((n, i) => n * i);
console.log(b.join(", ")); // 0, 2, 6
```

(Remember that `Array#map` passes the entry as the first argument, and the index as the second.)

In both cases, the body of the function is just an expression; the function's return value will automatically be the result of that expression (you don't use an explicit `return`).

If you're doing more than just a single expression, use `{}` and an explicit `return` (if you need to return a value), as normal:

```
var a = [
  {first: "Joe", last: "Bloggs"},
  {first: "Albert", last: "Bloggs"},
  {first: "Mary", last: "Albright"}
];
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```
if (rv === 0) {  
    rv = a.first.localeCompare(b.first);  
}  
return rv;  
});  
console.log(JSON.stringify(a));
```

The version without `{ ... }` is called an arrow function with an *expression body* or *concise body*. (Also: A *concise* arrow function.) The one with `{ ... }` defining the body is an arrow function with a *function body*. (Also: A *verbose* arrow function.)

Method Declaration in Object Initializer (ES2015+)

ES2015 allows a shorter form of declaring a property that references a function called a *method definition*; it looks like this:

```
var o = {  
    foo() {  
    }  
};
```

the almost-equivalent in ES5 and earlier would be:

```
var o = {  
    foo: function foo() {  
    }  
};
```

the difference (other than verbosity) is that a method can use `super`, but a function cannot. So for instance, if you had an object that defined (say) `valueOf` using method syntax, it could use `super.valueOf()` to get the value `Object.prototype.valueOf` would have returned (before presumably doing something else with it), whereas the ES5 version would have to do `Object.prototype.valueOf.call(this)` instead.

That also means that the method has a reference to the object it was defined on, so if that object is temporary (for instance, you're passing it into `Object.assign` as one of the source objects), method syntax *could* mean that the object is retained in memory when otherwise it could have been garbage collected (if the JavaScript engine doesn't detect that situation and handle it if none of the methods uses `super`).

Constructor and Method Declarations in `class` (ES2015+)

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  getFullName() {
    return this.firstName + " " + this.lastName;
  }
}
```

There are two function declarations above: One for the constructor, which gets the name `Person`, and one for `getFullName`, which is a function assigned to `Person.prototype`.

edited Jan 11 at 13:58

answered Mar 4 '14 at 13:35



[T.J. Crowder](#)

742k 136 1344
1408

3 then the name `w` is simply ignored? – [BiAiB](#) Mar 4 '14 at 13:37

8 @PellePenna: Function names are useful for lots of things. The two biggies in my view are recursion, and the name of the function being shown in call stacks, exception traces, and such. – [T.J. Crowder](#) Mar 4 '14 at 13:42

4 @ChaimEliyah - "Accepting doesn't mean it's the best answer, it just means that it worked for the person who asked." [source](#) – [ScrapCode](#) Feb 10 '16 at 10:19

6 @A.R.: Quite true. Amusingly, though, right above that it says "The best answers show up first so that they are always easy to find." Since the accepted answer shows up first even over higher-voted answers, the tour might be somewhat self-contradicting. ;-) Also a bit inaccurate, if we determine "best" by votes (which isn't reliable, it's just what we've got), "best" answers only show up first if you're using the "Votes" tab -- otherwise, the answers that are first are the active ones, or the oldest ones. – [T.J. Crowder](#) Feb 10 '16 at 10:32

1 @T.J.Crowder : Agreed. 'arranged by date' is sometimes annoying. – [ScrapCode](#) Feb 10 '16 at 10:46



138

Speaking about the global context, both, the `var` statement and a `FunctionDeclaration` at the end will create a *non-deletable* property on the global object, but the value of both *can be overwritten*.

The subtle difference between the two ways is that when the [Variable Instantiation](#) process runs (before the actual code execution) all

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```
alert(typeof foo); // 'function', it's already available
alert(typeof bar); // 'undefined'
function foo () {}
var bar = function () {};
alert(typeof bar); // 'function'
```

The assignment of the `bar` `FunctionExpression` takes place until runtime.

A global property created by a `FunctionDeclaration` can be overwritten without any problems just like a variable value, e.g.:

```
function test () {}
test = null;
```

Another obvious difference between your two examples is that the first function doesn't have a name, but the second has it, which can be really useful when debugging (i.e. inspecting a call stack).

About your edited first example (`foo = function() { alert('hello!'); };`), it is an undeclared assignment, I would highly encourage you to always use the `var` keyword.

With an assignment, without the `var` statement, if the referenced identifier is not found in the scope chain, it will become a *deleteable* property of the global object.

Also, undeclared assignments throw a `ReferenceError` on ECMAScript 5 under [Strict Mode](#).

A must read:

- [Named function expressions demystified](#)

Note: This answer has been merged from [another question](#), in which the major doubt and misconception from the OP was that identifiers declared with a `FunctionDeclaration`, couldn't be overwritten which is not the case.

edited May 23 '17 at 12:10



answered Aug 8 '10 at 19:32



I did not know that functions could be overwritten in JavaScript! Also, that parse order is the big selling point for me. I guess I need to watch how I create functions. – [Xeoncross](#) Aug 8 '10 at 19:43

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

Nov 29 '11 at 15:25

@Mr_Chimp I'm pretty sure it is, thewaybackmachine is saying that it got a 302 at crawl time and the redirection was to the link you provided. – John
Jan 30 '12 at 15:18



119



The two code snippets you've posted there will, for almost all purposes, behave the same way.

However, the difference in behaviour is that with the first variant (`var functionOne = function() {}`), that function can only be called after that point in the code.

With the second variant (`function functionTwo()`), the function is available to code that runs above where the function is declared.

This is because with the first variant, the function is assigned to the variable `foo` at run time. In the second, the function is assigned to that identifier, `foo`, at parse time.

More technical information

JavaScript has three ways of defining functions.

1. Your first snippet shows a **function expression**. This involves using the *"function" operator* to create a function - the result of that operator can be stored in any variable or object property. The function expression is powerful that way. The function expression is often called an "anonymous function", because it does not have to have a name,
2. Your second example is a **function declaration**. This uses the *"function" statement* to create a function. The function is made available at parse time and can be called anywhere in that scope. You can still store it in a variable or object property later.
3. The third way of defining a function is the **"Function()" constructor**, which is not shown in your original post. It's not recommended to use this as it works the same way as `eval()`, which has its problems.

edited Dec 28 '15 at 19:47



Peter Mortensen

14.5k 19 89 118

answered Apr 20 '10 at 4:54



thomasrutter

94k 22 131 153



A better explanation to [Greg's answer](#)

22

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

Why no error? We were always taught that expressions are executed from top to bottom(??)

Because:

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. [ben cherry](#).

This means that code like this:

functionOne();			var functionOne;
var functionOne = function () {	is actually		functionOne();
};	interpreted -->		
	like		functionOne = function () {
			};

Notice that the assignment portion of the declarations were not hoisted. Only the name is hoisted.

But in the case with function declarations, the entire function body will be hoisted as well:

functionTwo();			function functionTwo() {
function functionTwo() {	is actually		};
}	interpreted -->		
	like		functionTwo();

edited May 23 '17 at 12:26



Community ♦

1 1

answered Aug 9 '14 at 2:45



[suhailvs](#)

9,904 4 54 71

Hi suhail thanks for clear info about function topic. Now my question is which one will be the first declaration in declaration hierarchy whether variable declaration (functionOne) or function declaration (functionTwo) ? – [Sharathi RB](#) Feb 2 '16 at 12:09



Other commenters have already covered the semantic difference of the two variants above. I wanted to note a stylistic difference: Only the "assignment" variation can set a property of another object.

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).


```
(function(){
  var exports = {};

  function privateUtil() {
    ...
  }

  exports.publicUtil = function() {
    ...
  };

  return exports;
})();
```

With this pattern, your public functions will all use assignment, while your private functions use declaration.

(Note also that assignment should require a semicolon after the statement, while declaration prohibits it.)

edited Oct 19 '14 at 22:24



ROMANIA_engineer

37.6k 20 165 154

answered Mar 3 '11 at 19:19



Sean McMillan

7,713 4 48 62

- 4 yuiblog.com/blog/2007/06/12/module-pattern is the primordial reference for the module pattern, as far as I can tell. (Though that article uses the `var foo = function(){...}` syntax even for private variables. – Sean McMillan Jun 3 '11 at 12:32

This isn't entirely true in some older versions of IE, actually. (`function window.onload() {}` was a thing.) – Ry- ♦ Apr 21 '13 at 19:42

An illustration of when to prefer the first method to the second one is when you need to avoid overriding a function's previous definitions.

With

```
if (condition){
  function myfunction(){
    // Some code
  }
}
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

While

```

if (condition){
    var myfunction = function (){
        // Some code
    }
}

```

does the correct job of defining `myfunction` only when `condition` is met.

edited Jun 29 '17 at 14:08



Alireza

62.5k 15 200 135

answered Mar 29 '13 at 13:26



Mbengue Assane

2,405 1 13 14

- 1 this example is good and is close to perfection, but could be improved. the better example would be to defined `var myFunc = null;` outside of a loop, or outside of an if/elseif/else block. Then you can conditionally assign different functions to the same variable. In JS, it is a better convention to assign a missing value to null, then to undefined. Therefore, you should declare `myFunction` as null first, then assign it later, conditionally. – [Alexander Mills](#) May 26 '15 at 20:31

An important reason is to add one and only one variable as the "Root" of your namespace...

60

```

var MyNamespace = {}
MyNamespace.foo= function() {

}

```

or

```

var MyNamespace = {
    foo: function() {
    },
    ...
}

```

There are many techniques for namespacing. It's become more important with the plethora of JavaScript modules available.

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

edited May 23 '17 at 11:55

answered Aug 8 '10 at 19:44



Community ♦

1 1



Rob

4,897 1 18 26

3 It appears this answer was merged into this question from another question, and the wording *might* seem to be a tiny bit unrelated to *this* question. Would you consider editing the answer so it seems more directed specifically at this question? (to reiterate; this isn't your fault at all... just a side-effect of a merged question). You can also delete it, and I think you would keep your reputation. Or you can leave it; since it's old, it may not make a big difference. – Andrew Barber May 29 '13 at 21:13



54



Hoisting is the JavaScript interpreter's action of moving all variable and function declarations to the top of the current scope.

However, only the actual declarations are hoisted. by leaving assignments where they are.

- variable's/Function's declared inside the page are global can access anywhere in that page.
- variable's/Functions declared inside the function are having local scope. means they are available/accessed inside the function body (scope), they are not available outside the function body.

Variable

Javascript is called loosely typed language. Which means Javascript variables can hold value of any [Data-Type](#). Javascript automatically takes care of changing the variable-type based on the value/literal provided during runtime.

global_Page = 10;		var global_Page;	«
undefined			
« Integer literal, Number Type.	-----	global_Page = 10;	«
Number			
global_Page = 'Yash';	Interpreted	global_Page = 'Yash';	«
String			
« String literal, String Type.	« AS «	global_Page = true;	«
Boolean			
var global_Page = true;		global_Page = function	
() { « function			
« Boolean Type	-----	var	
local_functionblock; « undefined			
global_Page = function () {			
local_functionblock = 777; « Number			
var local_functionblock = 777;			
// Assigning function as a data			
		};	

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

Function

```
function Identifier_opt ( FormalParameterList_opt ) {
    FunctionBody | sequence of statements

    « return; Default undefined
    « return 'some data';
}
```

- functions declared inside the page are hoisted to top of the page having global access.
- functions declared inside the function-block are hoisted to top of the block.
- Default return value of function is 'undefined', [Variable](#) declaration default value also 'undefined'

Scope with respect to function-block **global**.
 Scope with respect to page **undefined** | **not** available.

Function Declaration

<pre>function globalAccess() { } globalAccess(); Re-Defined / overridden. localAccess(); function globalAccess() { localAccess(); } accessed with in globalAccess() only. function localAccess() { } } ReferenceError as the function is not defined</pre>	<p>-----</p> <p> </p> <p>« Hoisted As «</p> <p> </p> <p>-----</p>	<pre>function globalAccess() { } function globalAccess() { « function localAccess() { } localAccess(); « function } globalAccess(); localAccess(); «</pre>
--	---	--

Function Expression

<pre>10; (10); var a; a = 10; (function invoke() {</pre>	<pre>« literal « Expression « Expression var « Expression Function</pre>	<pre>(10).toString() -> '10' a.toString() -> '10'</pre>
--	--	---

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```
var f;
  f = function (){          « Expression var Function
    console.log('var Function');          f ()  -> 'var
Function'
  };

```

Function assigned to variable Example:

```
(function selfExecuting(){
  console.log('IIFE - Immediately-Invoked Function Expression');
})();

var anonymous = function (){
  console.log('anonymous function Expression');
};

var namedExpression = function for_InternalUSE(fact){
  if(fact === 1){
    return 1;
  }

  var localExpression = function(){
    console.log('Local to the parent Function Scope');
  };
  globalExpression = function(){
    console.log('creates a new global variable, then assigned this function.');
```

javascript interpreted as

```
var anonymous;
var namedExpression;
var globalExpression;

anonymous = function () {

```

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

```

namedExpression = function for_InternalUSE(fact){
    var localExpression;

    if(fact === 1){
        return 1;
    }
    localExpression = function() {
        console.log('Local to the parent Function Scope');
    };
    globalExpression = function() {
        console.log('creates a new global variable, then assigned this function.');
```

```

    };

    return fact * for_InternalUSE( fact - 1);    // DEFAULT UNDEFINED.
};

namedExpression(10);
globalExpression();
```

You can check function declaration, expression test over different browser's using [jsperf Test Runner](#)

ES5 Constructor Function Classes: Function objects created using Function.prototype.bind

JavaScript treats functions as first-class objects, so being an object, you can assign properties to a function.

```

function Shape(id) { // Function Declaration
    this.id = id;
};

// Adding a prototyped method to a function.
Shape.prototype.getID = function () {
    return this.id;
};
Shape.prototype.setID = function ( id ) {
    this.id = id;
};

var expFn = Shape; // Function Expression

var funObj = new Shape( ); // Function Object
funObj.hasOwnProperty('prototype'); // false
funObj.setID( 10 );
console.log( funObj.getID() ); // 10
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

ES6 introduced **Arrow function**: An arrow function expression has a shorter syntax, they are best suited for non-method functions, and they cannot be used as constructors.

[ArrowFunction : ArrowParameters => ConciseBody](#) .

```
const fn = (item) => { return item & 1 ? 'Odd' : 'Even'; };  
console.log( fn(2) ); // Even  
console.log( fn(3) ); // Odd
```

edited Sep 28 '17 at 6:35

answered Jan 25 '16 at 14:46



Yash

5,399 1 34 49

3 ahm, your answer... isn't it ambiguous? well written though so +1 for spending and writing too much info. – [Danish](#) Jan 31 '16 at 17:56

I'm adding my own answer just because everyone else has covered the hoisting part thoroughly.

38

I've wondered about which way is better for a long while now, and thanks to <http://jsperf.com> now I know :)

Testing in Chrome 42.0.2311.135 on OS X 10.10.3		
Test		Ops/sec
function declaration	<pre>function myFunc(a, b) { return (a + b); }; myFunc(2, 4);</pre>	1,663,624,539 ±1.12% fastest
function expression	<pre>var myFunc = function(a, b) { return (a + b); }; myFunc(2, 4);</pre>	57,878,148 ±1.30% 97% slower

Function declarations are faster, and that's what really matters in web dev right? ;)

answered May 1 '15 at 15:06



Leon Gaban

11.6k 49 206 384

8 I'd say that maintainability is the most important aspect of most code. Performance is important, but in most cases IO is likely to be a bigger bottleneck than the way you define your functions. However there are some problems where you need every bit of performance you can get and this is useful in those cases. Also good to have an answer here that answers clearly a well defined part of the question. – [Richard Garside](#) May 2 '15 at 15:22

3 Well, I found it to be other way around with Firefox. jsperf.com/sandytest – [Sandeep Nayak](#) Nov 17 '15 at 14:15

2 Microbenchmarks always fail. Looking to jsperf.com is a waste of time. What you really need is to look to JS engine source code, official documentation, or at least sniff dev blogs or mail lists. – [gavenkoa](#) Jan 10 '16 at 15:13

Just an update, since I've gone full functional programming style in JavaScript now, I never use declarations, only function expressions so I can chain and call my functions by their variable names. Check out RamdaJS... – [Leon Gaban](#) Dec 28 '16 at 16:56

1 @SandeepNayak I just ran your own test in Firefox 50.0.0 / Windows 7 0.0.0, and it actually is the same way as Leon's. So if your test is correct, I would conclude that jsperf's tests are not indicative, and it all depends on your browser and / or OS version, or in the particular state of the current machine in that particular moment. – [Sandeep Nayak](#) Jan 10 '16 at 15:13

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

32

A function declaration and a function expression assigned to a variable behave the same once the binding is established.

There is a difference however at *how* and *when* the function object is actually associated with its variable. This difference is due to the mechanism called *variable hoisting* in JavaScript.

Basically, all function declarations and variable declarations are hoisted to the top of the *function* in which the declaration occurs (this is why we say that JavaScript has *function scope*).

- When a function declaration is hoisted, the function body "follows" so when the function body is evaluated, the variable will immediately be bound to a function object.
- When a variable declaration is hoisted, the initialization does *not* follow, but is "left behind". The variable is initialized to `undefined` at the start of the function body, and will be *assigned* a value at its original location in the code. (Actually, it will be assigned a value at *every* location where a declaration of a variable with the same name occurs.)

The order of hoisting is also important: function declarations take precedence over variable declarations with the same name, and the last function declaration takes precedence over previous function declarations with the same name.

Some examples...

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

Variable `foo` is hoisted to the top of the function, initialized to `undefined`, so that `!foo` is `true`, so `foo` is assigned `10`. The `foo` outside of `bar`'s scope plays no role and is untouched.

```
function f() {
  return a;
  function a() {return 1;}
  var a = 4;
  function a() {return 2;}
}
f()() // 2

function f() {
  return a;
}
```

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

```
function a() {return 2;}  
f()() // 2
```

Function declarations take precedence over variable declarations, and the last function declaration "sticks".

```
function f() {  
  var a = 4;  
  function a() {return 1};  
  function a() {return 2};  
  return a; }  
f() // 4
```

In this example `a` is initialized with the function object resulting from evaluating the second function declaration, and then is assigned `4`.

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

Here the function declaration is hoisted first, declaring and initializing variable `a`. Next, this variable is assigned `10`. In other words: the assignment does not assign to outer variable `a`.

answered Feb 6 '13 at 16:29



[eljense](#)

12.4k

5

50

62

-
- 3 You have a bit weird way to place the closing braces. Are you a Python coder? It looks like you try to make Javascript look like Python. I am afraid it is confusing for other people. If I had to maintain your JavaScript code I would let your code through an automatic prettyprinter first. – [nalply](#) May 14 '13 at 12:46
-
- 1 Excellent post. A 'self-executing function' or 'immediately invoked function expression' should be easy enough to see and his style preference should not detract from his post - which is accurate and summarizes 'hoisting' perfectly. +1 – [Ricalsin](#) Dec 17 '13 at 18:13
-

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

31

```
function abc(){}  
▼
```

The second example is a function expression:

```
var abc = function() {};
```

The main difference is how they are hoisted (lifted and declared). In the first example, the whole function declaration is hoisted. In the second example only the var 'abc' is hoisted, its value (the function) will be undefined, and the function itself remains at the position that it is declared.

To put it simply:

```
//this will work  
abc(param);  
function abc(){}  
  
//this would fail  
abc(param);  
var abc = function() {}
```

To study more about this topic I strongly recommend you this [link](#)

edited May 9 '15 at 9:37

answered Jun 5 '14 at 8:28



[sla55er](#)

551 1 6 16

1 Your example is kind of the same as the top answer – [GôTô](#) Jun 5 '14 at 8:34

The main reason for posting this answer was to provide the link at the bottom. This was the piece that was missing for me to fully understand the above question. – [sla55er](#) Jun 8 '14 at 5:44

1 It's very cool that you wanted to share the link. But links to additional information, in SO, should just be a comment on either the question or your favorite answer. It's very much sub-optimal to clutter a long, complicated page like this with repeated information just to add a single useful link at the end of it. No, you won't get rep points for providing the link, but you'll be helping the community. – [XML](#) Jan 13 '15 at 0:26

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

28

- More resistant to mistakes like conditional initialization (you are still able to override if wanted to).
- The code becomes more readable by allocating local functions separately of scope functionality. Usually in the scope the functionality goes first, followed by declarations of local functions.
- In a debugger you will clearly see the function name on the call stack instead of an "anonymous/evaluated" function.

I suspect more PROS for named functions are follow. And what is listed as an advantage of named functions is a disadvantage for anonymous ones.

Historically, anonymous functions appeared from the inability of JavaScript as a language to list members with named functions:

```
{
  member: function() { /* How do I make "this.member" a named function? */
}
}
```

edited Dec 28 '15 at 19:44



Peter Mortensen

14.5k 19 89 118

answered Jan 23 '10 at 20:32



Sasha Firsov

552 7 6

2 There are the test to confirm: blog.firsov.net/2010/01/... JS performance test - scope and named functions - Analytics – Sasha Firsov Feb 4 '10 at 0:45

▲

I use the variable approach in my code for a very specific reason, the theory of which has been covered in an abstract way above, but an example might help some people like me, with limited JavaScript expertise.

24

▼

I have code that I need to run with 160 independently-designed brandings. Most of the code is in shared files, but branding-specific stuff is in a separate file, one for each branding.

Some brandings require specific functions, and some do not. Sometimes I have to add new functions to do new branding-specific things. I am happy to change the shared coded, but I don't want to have to change all 160 sets of branding files.

By using the variable syntax, I can declare the variable (a function pointer essentially) in the shared code and either assign a trivial stub function, or set to null.

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

From people's comments above, I gather it may be possible to redefine a static function too, but I think the variable solution is nice and clear.

edited Dec 28 '15 at 20:26



Peter Mortensen

14.5k 19 89 118

answered Nov 29 '12 at 11:28



Herc

427 3 9

23

In computer science terms, we talk about anonymous functions and named functions. I think the most important difference is that an anonymous function is not bound to an name, hence the name anonymous function. In JavaScript it is a first class object dynamically declared at runtime.

For more information on anonymous functions and lambda calculus, Wikipedia is a good start (http://en.wikipedia.org/wiki/Anonymous_function).

edited Oct 19 '14 at 22:25



ROMANIA_engineer

37.6k 20 165 154

answered Dec 18 '08 at 19:30



Kafka

392 1 4

22

[Greg's Answer](#) is good enough, but I still would like to add something to it that I learned just now watching [Douglas Crockford's](#) videos.

Function expression:

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

Function statement:

```
function foo() {};
```

The function statement is just a shorthand for `var` statement with a `function` value.

So

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

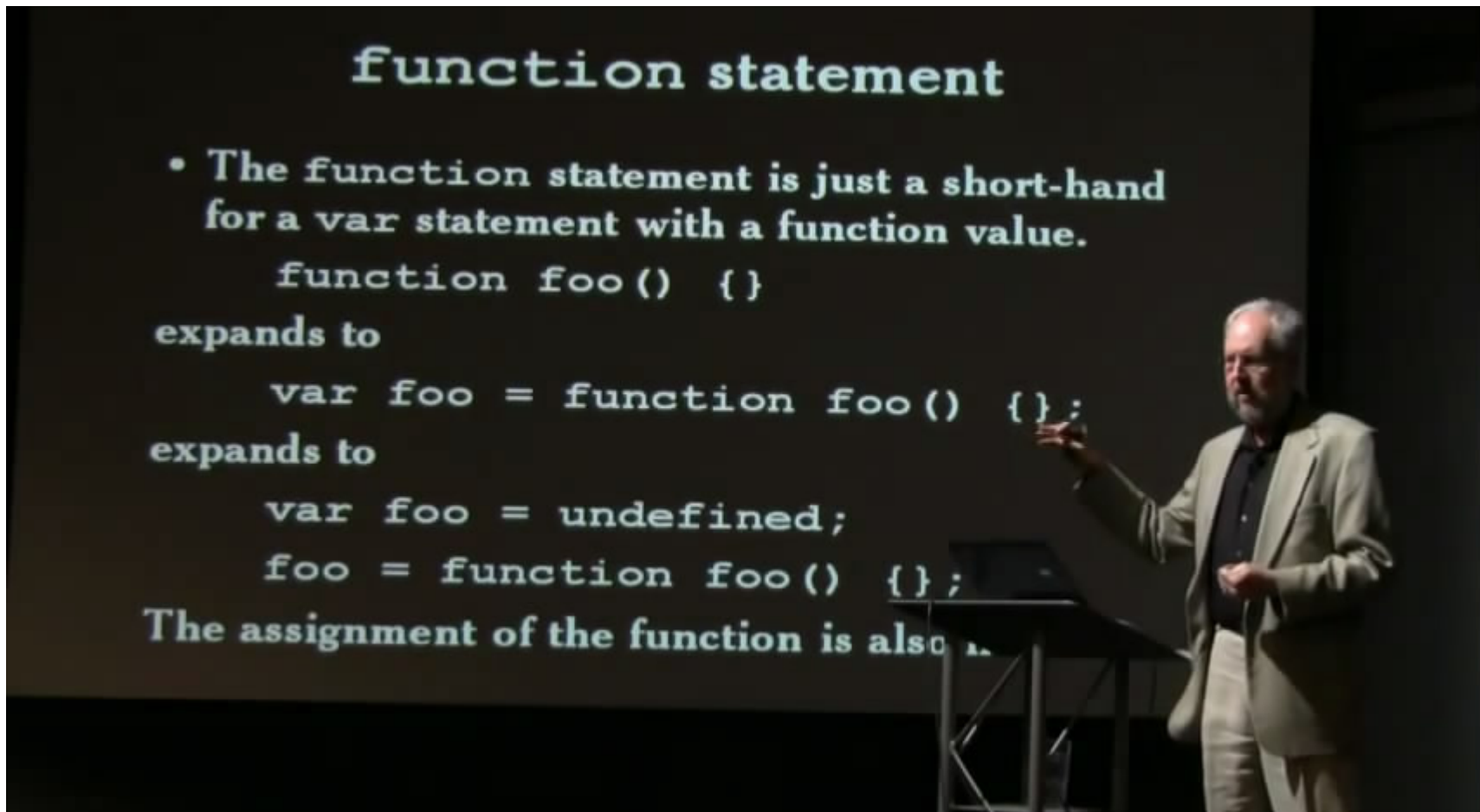
expands to

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

Which expands further to:

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

And they are both hoisted to the top of the code.

A man with a beard, wearing a light-colored blazer over a dark shirt, stands on a stage next to a podium. He is gesturing with his right hand towards a large screen behind him. The screen displays a presentation slide titled "function statement" in a green, monospaced font. The slide contains a bulleted point explaining that a function statement is a shorthand for a var statement with a function value, followed by code examples showing the expansion of "function foo() {}" into "var foo = function foo() {};" and then "var foo = undefined; foo = function foo() {};". The text "The assignment of the function is also" is partially visible at the bottom of the slide.

function statement

- The function statement is just a short-hand for a var statement with a function value.

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

expands to

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

expands to

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

The assignment of the function is also

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.



Community ♦

1 1



Rohan

5,153 14 49 107

- 7 Sorry but this is incorrect - I don't know what Crockford is trying to say in that slide. Both function & variable declarations are always hoisted to top of their scope. The difference is that variable assignments (whether you are assigning it with a string, boolean or function) are not hoisted to the top whereas function bodies (using function declaration) are. – [Thomas Heymann](#) Oct 5 '15 at 16:25 ✎

Have a look at these code examples: gist.github.com/cyberthom/36603fbc20de8e04fd09 – [Thomas Heymann](#) Oct 5 '15 at 16:51 ✎



18

[@EugeneLazutkin](#) gives an example where he [names an assigned function to be able to use shortcut\(\)](#) as an internal reference to itself. [John Resig](#) gives another example - *copying a recursive function assigned to another object* in his [Learning Advanced Javascript](#) tutorial. While assigning functions to properties isn't strictly the question here, I recommend actively trying the tutorial out - run the code by clicking the button in the upper right corner, and double click the code to edit to your liking.



Examples from the tutorial: recursive calls in `yell()` :

[Tests fail when the original ninja object is removed.](#) (page 13)

```
var ninja = {
  yell: function(n){
    return n > 0 ? ninja.yell(n-1) + "a" : "hiy";
  }
};
assert( ninja.yell(4) == "hiyaaaa", "A single object isn't too bad, either." );

var samurai = { yell: ninja.yell };
var ninja = null;

try {
  samurai.yell(4);
} catch(e){
  assert( false, "Uh, this isn't good! Where'd ninja.yell go?" );
}
```

[If you name the function that will be called recursively, the tests will pass.](#) (page 14)

```
var ninja = {
  ...
}
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```

};
assert( ninja.yell(4) == "hiyaaaa", "Works as we would expect it to!" );

var samurai = { yell: ninja.yell };
var ninja = {};
assert( samurai.yell(4) == "hiyaaaa", "The method correctly calls itself." );

```

edited May 23 '17 at 12:34



Community ♦

1 1

answered Aug 4 '12 at 15:24



Joel Purra

18.1k 7 51 56

Just calling `this.yell` works too :) – Ry- ♦ Apr 21 '13 at 20:54



Another difference that is not mentioned in the other answers is that if you use the anonymous function

16



```

var functionOne = function() {
    // Some code
};

```

and use that as a constructor as in

```
var one = new functionOne();
```

then `one.constructor.name` will not be defined. `Function.name` is non-standard but is supported by Firefox, Chrome, other Webkit-derived browsers and IE 9+.

With

```

function functionTwo() {
    // Some code
}
two = new functionTwo();

```

it is possible to retrieve the name of the constructor as a string with `two.constructor.name`.

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).



The name in the first case will not be defined because its an **anonymous** function assigned to a variable. I think the word anonymous was invented for things that don't have their name defined :) – [Om Shankar](#) Jan 17 '13 at 6:13

In this example the two=new becomes a global function because no var – [Waqas Tahir](#) Aug 29 '15 at 5:04

If you would use those functions to create objects, you would get:

14

```
var objectOne = new functionOne();
console.log(objectOne.__proto__); // prints "Object {}" because constructor is an
anonymous function

var objectTwo = new functionTwo();
console.log(objectTwo.__proto__); // prints "functionTwo {}" because constructor is a
named function
```

answered Oct 25 '13 at 16:38



[Pawel Furmaniak](#)
2,834 3 24 33

I can't seem to reproduce this. `console.log(objectOne.__proto__);` prints "functionOne {}" in my console. Any ideas of why this may be the case? – [Mike](#) Mar 22 '15 at 10:06

I can't seem to reproduce it as well. – [daremkd](#) Jan 21 '16 at 15:57

- 1 This is a capability of your debugger (to display the "class" of the logged object), and most ones are able to derive a name even for anonymous function expressions these days. Btw, you should make clear that there is no functional difference between the two instances. – [Bergi](#) Jan 21 '16 at 16:26

The first one (function doSomething(x)) should be part of an object notation.

14

The second one (`var doSomething = function(x){ alert(x);}`) is simply creating an anonymous function and assigning it to a variable, `doSomething` . So `doSomething()` will call the function.

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

A function declaration defines a named function variable without requiring variable assignment. Function declarations occur as standalone constructs and cannot be nested within non-function blocks.

```
function foo() {  
    return 3;  
}
```

ECMA 5 (13.0) defines the syntax as

function Identifier (FormalParameterList_{opt}) { FunctionBody }

In above condition the function name is visible within its scope and the scope of its parent (otherwise it would be unreachable).

And in a function expression

A function expression defines a function as a part of a larger expression syntax (typically a variable assignment). Functions defined via functions expressions can be named or anonymous. Function expressions should not start with “function”.

```
// Anonymous function expression  
var a = function() {  
    return 3;  
}
```

```
// Named function expression  
var a = function foo() {  
    return 3;  
}
```

```
// Self-invoking function expression  
(function foo() {  
    alert("hello!");  
})();
```

ECMA 5 (13.0) defines the syntax as

function Identifier_{opt} (FormalParameterList_{opt}) { FunctionBody }

edited Dec 28 '15 at 20:29



Peter Mortensen

answered Jan 5 '13 at 18:37



NullPointer

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

14

I'm listing out the differences below:

1. A function declaration can be placed anywhere in the code. Even if it is invoked before the definition appears in code, it gets executed as function declaration is committed to memory or in a way it is hoisted up, before any other code in the page starts execution.

Take a look at the function below:

```
function outerFunction() {  
    function foo() {  
        return 1;  
    }  
    return foo();  
    function foo() {  
        return 2;  
    }  
}  
alert(outerFunction()); // Displays 2
```

This is because, during execution, it looks like:-

```
function foo() { // The first function declaration is moved to top  
    return 1;  
}  
function foo() { // The second function declaration is moved to top  
    return 2;  
}  
function outerFunction() {  
    return foo();  
}  
alert(outerFunction()); //So executing from top to bottom,  
                        //the last foo() returns 2 which gets displayed
```

A function expression, if not defined before calling it, will result in an error. Also, here the function definition itself is not moved to the top or committed to memory like in the function declarations. But the variable to which we assign the function gets hoisted up and **undefined** gets assigned to it.

Same function using function expressions:

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

```

    }
    return foo();
    var foo = function() {
        return 2;
    }
}
alert(outerFunction()); // Displays 1

```

This is because during execution, it looks like:

```

function outerFunction() {
    var foo = undefined;
    var foo = undefined;

    foo = function() {
        return 1;
    };
    return foo ();
    foo = function() { // This function expression is not reachable
        return 2;
    };
}
alert(outerFunction()); // Displays 1

```

2. It is not safe to write function declarations in non-function blocks like **if** because they won't be accessible.

```

if (test) {
    function x() { doSomething(); }
}

```

3. Named function expression like the one below, may not work in Internet Explorer browsers prior to version 9.

```

var today = function today() {return new Date();}

```

edited Dec 28 '15 at 20:35



Peter Mortensen

14.5k 19 89 118

answered Sep 9 '15 at 10:30



varna

851 8 12

-
- 1 @Arjun What's the problem if a question was asked years earlier ? An answer doesn't only benefit the OP but potentially all SO users, no matter when the question was asked. And what's wrong with answering questions that already have an accepted answer ? – SantiBailors Oct 5 '15 at 12:58

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

the answer doesn't make sense and doesn't belong here, it would be downvoted and removed automatically. You don't need to bother with it!!!! –
[Sudhansu Choudhary](#) Oct 18 '15 at 21:37

13 There are four noteworthy comparisons between the two different declarations of functions as listed below.

1. Availability (scope) of the function

The following works because `function add()` is scoped to the nearest block:

```
try {
  console.log("Success: ", add(1, 1));
} catch(e) {
  console.log("ERROR: " + e);
}

function add(a, b){
  return a + b;
}
```

[Run code snippet](#)[Expand snippet](#)

The following does not work because the variable is called before a function value is assigned to the variable `add`.

```
try {
  console.log("Success: ", add(1, 1));
} catch(e) {
  console.log("ERROR: " + e);
}

var add=function(a, b){
  return a + b;
}
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

The above code is identical in functionality to the code below. Note that explicitly assigning `add = undefined` is superfluous because simply doing `var add;` is the exact same as `var add=undefined`.

```
var add = undefined;

try {
  console.log("Success: ", add(1, 1));
} catch(e) {
  console.log("ERROR: " + e);
}

add = function(a, b){
  return a + b;
}
```

[Run code snippet](#)[Expand snippet](#)

The following does not work because the `var add=` superseeds the `function add()`.

```
try {
  console.log("Success: ", add(1, 1));
} catch(e) {
  console.log("ERROR: " + e);
}

var add=function add(a, b){
  return a + b;
}
```

[Run code snippet](#)[Expand snippet](#)

2. (function).name

The name of a function `function thefuncname(){} is thefuncname when it is declared this way.`

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

[Run code snippet](#)[Expand snippet](#)

```
var a = function foobar(){};

console.log(a.name);
```

[Run code snippet](#)[Expand snippet](#)

Otherwise, if a function is declared as `function(){} ,` the `function.name` is the first variable used to store the function.

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

console.log(a.name);
console.log(b.name);
```

[Run code snippet](#)[Expand snippet](#)

If there are no variables set to the function, then the functions name is the empty string (`""`).

```
console.log((function(){}).name === "");
```

[Run code snippet](#)[Expand snippet](#)

Lastly, while the variable the function is assigned to initially sets the name, successive variables set to the function do not change the name.

```
var a = function(){};
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```
console.log(a.name);  
console.log(b.name);  
console.log(c.name);
```

[Run code snippet](#)[Expand snippet](#)

3. Performance

In Google's V8 and Firefox's Spidermonkey there might be a few microsecond JIST compilation difference, but ultimately the result is the exact same. To prove this, let's examine the efficiency of JSPerf at microbenchmarks by comparing the speed of two blank code snippets. The [JSPerf tests are found here](#). And, the [jsben.ch tests are found here](#). As you can see, there is a noticeable difference when there should be none. If you are really a performance freak like me, then it might be more worth your while trying to reduce the number of variables and functions in the scope and especially eliminating polymorphism (such as using the same variable to store two different types).

4. Variable Mutability

When you use the `var` keyword to declare a variable, you can then reassign a different value to the variable like so.

```
(function(){  
  "use strict";  
  var foobar = function(){}; // initial value  
  try {  
    foobar = "Hello World!"; // new value  
    console.log("[no error]");  
  } catch(error) {  
    console.log("ERROR: " + error.message);  
  }  
  console.log(foobar, window.foobar);  
})();
```

[Run code snippet](#)[Expand snippet](#)

However, when we use the `const`-statement, the variable reference becomes immutable. This means that we cannot assign a new value to the variable. Please note, however, that this does not make the contents of the variable immutable: if you do `const arr = []`, then you can still do `arr[10] = "example"`. Only doing something like `arr = "new value"` or `arr = []` would throw an error as seen below.

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).


```
(function(){
  "use strict";
  const foobar = function(){}; // initial value
  try {
    foobar = "Hello World!"; // new value
    console.log("[no error]");
  } catch(error) {
    console.log("ERROR: " + error.message);
  }
  console.log(foobar, window.foobar);
})();
```

[Run code snippet](#)[Expand snippet](#)

Interestingly, if we declare the variable as `function funcName(){} ,` then the immutability of the variable is the same as declaring it with `var .`

```
(function(){
  "use strict";
  function foobar(){}; // initial value
  try {
    foobar = "Hello World!"; // new value
    console.log("[no error]");
  } catch(error) {
    console.log("ERROR: " + error.message);
  }
  console.log(foobar, window.foobar);
})();
```

[Run code snippet](#)[Expand snippet](#)

What Is The "Nearest Block"

The "nearest block" is the nearest "function," (including asynchronous functions, generator functions, and asynchronous generator functions). However, interestingly, a `function functionName() {}` behaves like a `var functionName = function() {}` when in a non-closure block to items outside said closure. Observe

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```
try {  
  // typeof will simply return "undefined" if the variable does not exist  
  if (typeof add !== "undefined") {  
    add(1, 1); // just to prove it  
    console.log("Not a block");  
  } else if (add === undefined) { // this throws an exception if add doesn't exist  
    console.log('Behaves like var add=function(a,b){return a+b}');  
  }  
} catch(e) {  
  console.log("Is a block");  
}  
var add=function(a, b){return a + b}
```

[Run code snippet](#)[Expand snippet](#)

- Normal function add(){}

```
try {  
  // typeof will simply return "undefined" if the variable does not exist  
  if (typeof add !== "undefined") {  
    add(1, 1); // just to prove it  
    console.log("Not a block");  
  } else if (add === undefined) { // this throws an exception if add doesn't exist  
    console.log('Behaves like var add=function(a,b){return a+b}')  
  }  
} catch(e) {  
  console.log("Is a block");  
}  
function add(a, b){  
  return a + b;  
}
```

[Run code snippet](#)[Expand snippet](#)

- Function

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```

    add(1, 1); // just to prove it
    console.log("Not a block");
  } else if(add===undefined){ // this throws an exception if add doesn't exist
    console.log('Behaves like var add=function(a,b){return a+b}');
  }
} catch(e) {
  console.log("Is a block");
}
(function () {
  function add(a, b){
    return a + b;
  }
})();

```

Run code snippet

[Expand snippet](#)

- Statement (such as `if`, `else`, `for`, `while`, `try / catch / finally`, `switch`, `do / while`, `with`)

```

try {
  // typeof will simply return "undefined" if the variable does not exist
  if (typeof add !== "undefined") {
    add(1, 1); // just to prove it
    console.log("Not a block");
  } else if(add===undefined){ // this throws an exception if add doesn't exist
    console.log('Behaves like var add=function(a,b){return a+b}');
  }
} catch(e) {
  console.log("Is a block");
}
{
  function add(a, b){
    return a + b;
  }
}

```

Run code snippet

[Expand snippet](#)

- Arrow Function with `var add=function()`

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```
try {
  // typeof will simply return "undefined" if the variable does not exist
  if (typeof add !== "undefined") {
    add(1, 1); // just to prove it
    console.log("Not a block");
  } else if (add === undefined) { // this throws an exception if add doesn't exist
    console.log('Behaves like var add=function(a,b){return a+b}');
  }
} catch(e) {
  console.log("Is a block");
}
(() => {
  var add=function(a, b){
    return a + b;
  }
})();
```

[Run code snippet](#)[Expand snippet](#)

- Arrow Function With `function add()`

```
try {
  // typeof will simply return "undefined" if the variable does not exist
  if (typeof add !== "undefined") {
    add(1, 1); // just to prove it
    console.log("Not a block");
  } else if (add === undefined) { // this throws an exception if add doesn't exist
    console.log('Behaves like var add=function(a,b){return a+b}');
  }
} catch(e) {
  console.log("Is a block");
}
(() => {
  function add(a, b){
    return a + b;
  }
})();
```

[Run code snippet](#)[Expand snippet](#)

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).



▲ In light of the "named functions show up in stack traces" argument, modern JavaScript engines are actually quite capable of representing anonymous functions.

11



As of this writing, V8, SpiderMonkey, Chakra and Nitro always refer to named functions by their names. They almost always refer to an anonymous function by its identifier if it has one.

SpiderMonkey can figure out the name of an anonymous function returned from another function. The rest can't.

If you really, really wanted your iterator and success callbacks to show up in the trace, you could name those too...

```
[].forEach(function iterator() {});
```

But for the most part it's not worth stressing over.

Harness ([Fiddle](#))

```
'use strict';

var a = function () {
  throw new Error();
},
    b = function b() {
      throw new Error();
    },
    c = function d() {
      throw new Error();
    },
    e = {
      f: a,
      g: b,
      h: c,
      i: function () {
        throw new Error();
      },
    },
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```

    k: function l() {
        throw new Error();
    }
},
m = (function () {
    return function () {
        throw new Error();
    };
})();
n = (function () {
    return function n() {
        throw new Error();
    };
})();
o = (function () {
    return function p() {
        throw new Error();
    };
})();

console.log([a, b, c].concat(Object.keys(e).reduce(function (values, key) {
    return values.concat(e[key]);
}, [])).concat([m, n, o]).reduce(function (logs, func) {

    try {
        func();
    } catch (error) {
        return logs.concat('func.name: ' + func.name + '\n' +
            'Trace:\n' +
            error.stack);
        // Need to manually log the error object in Nitro.
    }

}, []).join('\n\n'));

```

V8

```

func.name:
Trace:
Error
    at a (http://localhost:8000/test.js:4:11)
    at http://localhost:8000/test.js:47:9
    at Array.reduce (native)
    at http://localhost:8000/test.js:44:27

```

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

Error

at b (http://localhost:8000/test.js:7:15)
at http://localhost:8000/test.js:47:9
at **Array**.reduce (**native**)
at http://localhost:8000/test.js:44:27

func.name: d

Trace:**Error**

at d (http://localhost:8000/test.js:10:15)
at http://localhost:8000/test.js:47:9
at **Array**.reduce (**native**)
at http://localhost:8000/test.js:44:27

func.name:

Trace:**Error**

at a (http://localhost:8000/test.js:4:11)
at http://localhost:8000/test.js:47:9
at **Array**.reduce (**native**)
at http://localhost:8000/test.js:44:27

func.name: b

Trace:**Error**

at b (http://localhost:8000/test.js:7:15)
at http://localhost:8000/test.js:47:9
at **Array**.reduce (**native**)
at http://localhost:8000/test.js:44:27

func.name: d

Trace:**Error**

at d (http://localhost:8000/test.js:10:15)
at http://localhost:8000/test.js:47:9
at **Array**.reduce (**native**)
at http://localhost:8000/test.js:44:27

func.name:

Trace:**Error**

at e.i (http://localhost:8000/test.js:17:19)
at http://localhost:8000/test.js:47:9
at **Array**.reduce (**native**)
at http://localhost:8000/test.js:44:27

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

```

at j (http://localhost:8000/test.js:20:19)
at http://localhost:8000/test.js:47:9
at Array.reduce (native)
at http://localhost:8000/test.js:44:27

```

func.name: l

Trace:

Error

```

at l (http://localhost:8000/test.js:23:19)
at http://localhost:8000/test.js:47:9
at Array.reduce (native)
at http://localhost:8000/test.js:44:27

```

func.name:

Trace:

Error

```

at http://localhost:8000/test.js:28:19
at http://localhost:8000/test.js:47:9
at Array.reduce (native)
at http://localhost:8000/test.js:44:27

```

func.name: n

Trace:

Error

```

at n (http://localhost:8000/test.js:33:19)
at http://localhost:8000/test.js:47:9
at Array.reduce (native)
at http://localhost:8000/test.js:44:27

```

func.name: p

Trace:

Error

```

at p (http://localhost:8000/test.js:38:19)
at http://localhost:8000/test.js:47:9
at Array.reduce (native)
at http://localhost:8000/test.js:44:27 test.js:42

```

SpiderMonkey

func.name:

Trace:

```

a@http://localhost:8000/test.js:4:5
@http://localhost:8000/test.js:47:9
@http://localhost:8000/test.js:54:1

```

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

Trace:

b@http://localhost:8000/test.js:7:9
@http://localhost:8000/test.js:47:9
@http://localhost:8000/test.js:54:1

func.name: d

Trace:

d@http://localhost:8000/test.js:10:9
@http://localhost:8000/test.js:47:9
@http://localhost:8000/test.js:54:1

func.name:

Trace:

a@http://localhost:8000/test.js:4:5
@http://localhost:8000/test.js:47:9
@http://localhost:8000/test.js:54:1

func.name: b

Trace:

b@http://localhost:8000/test.js:7:9
@http://localhost:8000/test.js:47:9
@http://localhost:8000/test.js:54:1

func.name: d

Trace:

d@http://localhost:8000/test.js:10:9
@http://localhost:8000/test.js:47:9
@http://localhost:8000/test.js:54:1

func.name:

Trace:

e.i@http://localhost:8000/test.js:17:13
@http://localhost:8000/test.js:47:9
@http://localhost:8000/test.js:54:1

func.name: j

Trace:

j@http://localhost:8000/test.js:20:13
@http://localhost:8000/test.js:47:9
@http://localhost:8000/test.js:54:1

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

Trace:

```
l@http://localhost:8000/test.js:23:13
@http://localhost:8000/test.js:47:9
@http://localhost:8000/test.js:54:1
```

func.name:

Trace:

```
m</>@http://localhost:8000/test.js:28:13
@http://localhost:8000/test.js:47:9
@http://localhost:8000/test.js:54:1
```

func.name: n

Trace:

```
n@http://localhost:8000/test.js:33:13
@http://localhost:8000/test.js:47:9
@http://localhost:8000/test.js:54:1
```

func.name: p

Trace:

```
p@http://localhost:8000/test.js:38:13
@http://localhost:8000/test.js:47:9
@http://localhost:8000/test.js:54:1
```

Chakra

func.name: **undefined**

Trace:**Error**

```
at a (http://localhost:8000/test.js:4:5)
at Anonymous function (http://localhost:8000/test.js:47:9)
at Global code (http://localhost:8000/test.js:42:1)
```

func.name: **undefined**

Trace:**Error**

```
at b (http://localhost:8000/test.js:7:9)
at Anonymous function (http://localhost:8000/test.js:47:9)
at Global code (http://localhost:8000/test.js:42:1)
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```
at d (http://localhost:8000/test.js:10:9)
at Anonymous function (http://localhost:8000/test.js:47:9)
at Global code (http://localhost:8000/test.js:42:1)
```

func.name: undefined

Trace:

Error

```
at a (http://localhost:8000/test.js:4:5)
at Anonymous function (http://localhost:8000/test.js:47:9)
at Global code (http://localhost:8000/test.js:42:1)
```

func.name: undefined

Trace:

Error

```
at b (http://localhost:8000/test.js:7:9)
at Anonymous function (http://localhost:8000/test.js:47:9)
at Global code (http://localhost:8000/test.js:42:1)
```

func.name: undefined

Trace:

Error

```
at d (http://localhost:8000/test.js:10:9)
at Anonymous function (http://localhost:8000/test.js:47:9)
at Global code (http://localhost:8000/test.js:42:1)
```

func.name: undefined

Trace:

Error

```
at e.i (http://localhost:8000/test.js:17:13)
at Anonymous function (http://localhost:8000/test.js:47:9)
at Global code (http://localhost:8000/test.js:42:1)
```

func.name: undefined

Trace:

Error

```
at j (http://localhost:8000/test.js:20:13)
at Anonymous function (http://localhost:8000/test.js:47:9)
at Global code (http://localhost:8000/test.js:42:1)
```

func.name: undefined

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

at **Anonymous function** (http://localhost:8000/test.js:47:9)
 at **Global** code (http://localhost:8000/test.js:42:1)

func.name: **undefined**

Trace:

Error

at **Anonymous function** (http://localhost:8000/test.js:28:13)
 at **Anonymous function** (http://localhost:8000/test.js:47:9)
 at **Global** code (http://localhost:8000/test.js:42:1)

func.name: **undefined**

Trace:

Error

at n (http://localhost:8000/test.js:33:13)
 at **Anonymous function** (http://localhost:8000/test.js:47:9)
 at **Global** code (http://localhost:8000/test.js:42:1)

func.name: **undefined**

Trace:

Error

at p (http://localhost:8000/test.js:38:13)
 at **Anonymous function** (http://localhost:8000/test.js:47:9)
 at **Global** code (http://localhost:8000/test.js:42:1)

Nitro

func.name:

Trace:

a@http://localhost:8000/test.js:4:22
 http://localhost:8000/test.js:47:13
 reduce@[**native** code]
global code@http://localhost:8000/test.js:44:33

func.name: b

Trace:

b@http://localhost:8000/test.js:7:26
 http://localhost:8000/test.js:47:13
 reduce@[**native** code]
global code@http://localhost:8000/test.js:44:33

func.name: d

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

```
reduce@[native code]  
global code@http://localhost:8000/test.js:44:33
```

```
func.name:  
Trace:  
a@http://localhost:8000/test.js:4:22  
http://localhost:8000/test.js:47:13  
reduce@[native code]  
global code@http://localhost:8000/test.js:44:33
```

```
func.name: b  
Trace:  
b@http://localhost:8000/test.js:7:26  
http://localhost:8000/test.js:47:13  
reduce@[native code]  
global code@http://localhost:8000/test.js:44:33
```

```
func.name: d  
Trace:  
d@http://localhost:8000/test.js:10:26  
http://localhost:8000/test.js:47:13  
reduce@[native code]  
global code@http://localhost:8000/test.js:44:33
```

```
func.name:  
Trace:  
i@http://localhost:8000/test.js:17:30  
http://localhost:8000/test.js:47:13  
reduce@[native code]  
global code@http://localhost:8000/test.js:44:33
```

```
func.name: j  
Trace:  
j@http://localhost:8000/test.js:20:30  
http://localhost:8000/test.js:47:13  
reduce@[native code]  
global code@http://localhost:8000/test.js:44:33
```

```
func.name: l  
Trace:  
l@http://localhost:8000/test.js:23:30  
http://localhost:8000/test.js:47:13  
reduce@[native code]  
global code@http://localhost:8000/test.js:44:33
```

```
func.name:
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```

reduce@[native code]
global code@http://localhost:8000/test.js:44:33

func.name: n
Trace:
n@http://localhost:8000/test.js:33:30
http://localhost:8000/test.js:47:13
reduce@[native code]
global code@http://localhost:8000/test.js:44:33

func.name: p
Trace:
p@http://localhost:8000/test.js:38:30
http://localhost:8000/test.js:47:13
reduce@[native code]
global code@http://localhost:8000/test.js:44:33

```

edited Jan 13 '15 at 3:24

answered Oct 12 '14 at 0:58



Jackson

5,771 2 35 58

▲ In JavaScript there are two ways to create functions:

10

1. Function declaration:

```

function fn(){
  console.log("Hello");
}
fn();

```

This is very basic, self-explanatory, used in many languages and standard across C family of languages. We declared a function defined it and executed it by calling it.

What you should be knowing is that functions are actually objects in JavaScript; internally we have created an object for above function and given it a name called fn or the reference to the object is stored in fn. Functions are objects in JavaScript; an instance of function is actually an object instance.

2. Function expression:

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

```
}
fn();
```

JavaScript has first-class functions, that is, create a function and assign it to a variable just like you create a string or number and assign it to a variable. Here, the `fn` variable is assigned to a function. The reason for this concept is functions are objects in JavaScript; `fn` is pointing to the object instance of the above function. We have initialized a function and assigned it to a variable. It's not executing the function and assigning the result.

Reference: [JavaScript function declaration syntax: var fn = function\(\) {} vs function fn\(\) {}](#).

edited Apr 9 '17 at 7:13



Peter Mortensen

14.5k 19 89 118

answered Aug 14 '16 at 9:13



Anoop Rai

259 4 4

1 what about the third option, `var fn = function fn() {...}` ? – [chharvey](#) Aug 20 '16 at 0:40

Hi Chharvey, not sure about ur question, I guess u r talking about function expression which I have already mentioned. However, if still there is some confusion just be more elaborative. – [Anoop Rai](#) Aug 23 '16 at 10:00

yes I was asking about a *named* function expression. it's similar to your option #2 except that the function has an identifier. usually this identifier is the same as the variable it's being assigned to, but that's not always the case. – [chharvey](#) Aug 23 '16 at 10:39

1 Yes Named function expression is similar to my option #2. Well having an identifier is not mandatory as it's not used. Whenever you will be executing the function expression you will use the variable holding the function object. Identifier serves no purpose. – [Anoop Rai](#) Aug 23 '16 at 11:42 ✎

9

Both are different ways of defining a function. The difference is how the browser interprets and loads them into an execution context.

The first case is of function expressions which loads only when the interpreter reaches that line of code. So if you do it like the following, you will get an error that the **functionOne is not a function**.

```
functionOne();
var functionOne = function() {
    // Some code
};
```

The reason is that on the first line no value is assigned to `functionOne`, and hence it is undefined. We are trying to call it as a function,

```
..
    ..
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

The second case is of function declarations that loads before any code is executed. So if you do like the following you won't get any error as the declaration loads before code execution.

```
functionOne();  
function functionOne() {  
    // Some code  
}
```

edited Dec 28 '15 at 20:42



Peter Mortensen

14.5k 19 89 118

answered Dec 28 '15 at 20:18



Nitin9791

679 8 14

About performance:

9

New versions of `v8` introduced several under-the-hood optimizations and so did `SpiderMonkey`.

There is almost no difference now between expression and declaration.

Function expression [appears to be faster](#) now.

Chrome 62.0.3202

Testing in Chrome 62.0.3202 / Windows 7 0.0.0		
Test		Ops/sec
Function declaration	<pre>function foo(x,y){ return (x+y); } foo(5,5);</pre>	473,749,508 ±0.33% fastest
Function expression	<pre>var foo = function(x,y){ return (x+y); } foo(5,5);</pre>	456,291,050 ±0.52% 4% slower
Function expression (global)	<pre>foo = function(x,y){ return (x+y); } foo(5,5);</pre>	15,990,025 ±2.41% 97% slower
Expression_new_function	<pre>var foo = new Function("x", "y", "return (x+y);"); foo(5,5);</pre>	672,063 ±0.28% 100% slower

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

Firefox 55

Testing in Firefox 55.0.0 / Windows 7 0.0.0		
Test		Ops/sec
Function declaration	<pre>function foo(x,y){ return (x+y); } foo(5,5);</pre>	955,216,006 ±0.34% 0.02% slower
Function expression	<pre>var foo = function(x,y){ return (x+y); } foo(5,5);</pre>	954,470,454 ±0.24% fastest
Function expression (global)	<pre>foo = function(x,y){ return (x+y); } foo(5,5);</pre>	40,101,461 ±9.95% 96% slower
Expression_new_function	<pre>var foo = new Function("x", "y", "return (x+y);"); foo(5,5);</pre>	53,660 ±5.50% 100% slower

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

Chrome Canary 63.0.3225

Testing in Chrome 63.0.3225 / Windows 7 0.0.0		
Test		Ops/sec
Function declaration	<pre>function foo(x,y){ return (x+y); } foo(5,5);</pre>	723,700,179 ±0.20% 0.12% slower
Function expression	<pre>var foo = function(x,y){ return (x+y); } foo(5,5);</pre>	724,528,937 ±0.20% fastest
Function expression (global)	<pre>foo = function(x,y){ return (x+y); } foo(5,5);</pre>	25,595,160 ±4.20% 97% slower
Expression_new_function	<pre>var foo = new Function("x", "y", "return (x+y);"); foo(5,5);</pre>	671,159 ±0.72% 100% slower

Anonymous function expressions [appear to have better performance](#) against Named function expression.

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

Firefox

Testing in Firefox 55.0.0 / Windows 7 0.0.0		
Test		Ops/sec
named function expression	<pre>var foo = function foo(x,y){ return (x+y); } foo(5,5);</pre>	959,026,268 ±0.18% fastest
anonymous function expression	<pre>var foo = function (x,y){ return (x+y); } foo(5,5);</pre>	959,166,819 ±0.22% fastest

Chrome Canary

Testing in Chrome 63.0.3225 / Windows 7 0.0.0		
Test		Ops/sec
named function expression	<pre>var foo = function foo(x,y){ return (x+y); } foo(5,5);</pre>	720,426,867 ±0.18% fastest
anonymous function expression	<pre>var foo = function (x,y){ return (x+y); } foo(5,5);</pre>	726,176,936 ±0.17% fastest


By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).


Chrome


Testing in Chrome 62.0.3202 / Windows 7 0.0.0		
Test		Ops/sec
named function expression	<pre>var foo = function foo(x,y){ return (x+y); } foo(5,5);</pre>	451,652,759 ±0.70% fastest
anonymous function expression	<pre>var foo = function (x,y){ return (x+y); } foo(5,5);</pre>	450,210,059 ±0.42% 0.03% slower

edited Sep 28 '17 at 5:13

answered Sep 28 '17 at 4:34

 **Panos Kal.**
8,978 8 49 67

- 1
- Yes, this difference is so insignificant that hopefully devs will concern themselves with which approach is more maintainable for their specific needs rather than which one *might* be faster (you'll get different jsperf results on each try depending on what the browser is doing -- the majority of javascript tasks needn't concern themselves with micro-optimizations to this degree). – squidbe Nov 26 '17 at 1:41 
- @squidbe There is no difference. Look here: jsperf.com/empty-tests-performance – Jack Giffin May 10 '18 at 21:10


8

They are pretty similar with some small differences, first one is a variable which assigned to an anonymous function (Function Declaration) and second one is the normal way to create a function in JavaScript(Anonymous function Declaration), both has usage, cons and pros:

1 Function Expression

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

```
var functionOne = function() {  
    // Some code  
};
```

A Function Expression defines a function as a part of a larger expression syntax (typically a variable assignment). Functions defined via Functions Expressions can be named or anonymous. Function Expressions must not start with “function” (hence the parentheses around the self invoking example below).

Assign a variable to a function, means no Hoisting, as we know functions in JavaScript can Hoist, means they can be called before they get declared, while variables need to be declared before getting access to them, so means in this case we can not access the function before where it's declared, also it could be a way that you write your functions, for the functions which return another function, this kind of declaration could make sense, also in ECMA6 & above you can assign this to an arrow function which can be used to call anonymous functions, also this way of declaring is a better way to create Constructor functions in JavaScript.

2. Function Declaration

```
function functionTwo() {  
    // Some code  
}
```

A Function Declaration defines a named function variable without requiring variable assignment. Function Declarations occur as standalone constructs and cannot be nested within non-function blocks. It's helpful to think of them as siblings of Variable Declarations. Just as Variable Declarations must start with “var”, Function Declarations must begin with “function”.

This is the normal way of calling a function in JavaScript, this function can be called before you even declare it as in JavaScript all functions get Hoisted, but if you have 'use strict' this won't Hoist as expected, it's a good way to call all normal functions which are not big in lines and neither are a constructor function.

Also, if you need more info about how hoisting works in JavaScript, visit the link below:

<https://developer.mozilla.org/en-US/docs/Glossary/Hoisting>

edited Jul 30 '17 at 4:30

answered May 9 '17 at 13:56



Alireza

62.5k


15

200

135

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

Karl Morrison Jun 13 '17 at 12:44 

One reason is because all built-in Constructor functions in JavaScript created like this function Number() { [native code] } and you shouldn't be confused with built-in ones, also referencing later on in this case is safer and you end up neater code but not using hoisting... – Alireza Jun 13 '17 at 12:56 

This is just two possible ways of declaring functions, and in the second way, you can use the function before declaration.

6

edited Dec 28 '15 at 20:32

answered Jun 24 '15 at 10:08



Peter Mortensen

14.5k 19 89 118



Tao

441 5 6

Please elaborate and provide working code snippets – Jack Giffin May 10 '18 at 21:06

`new Function()` can be used to pass the function's body in a string. And hence this can be used to create dynamic functions. Also passing the script without executing the script.

5

```
var func = new Function("x", "y", "return x*y;");
function secondFunction(){
    var result;
    result = func(10,20);
    console.log ( result );
}

secondFunction()
```

answered May 10 '16 at 7:05



SuperNova

7,978 3 41 30

While this is good and true, how exactly does this alone relate to the question being asked? – Jack Giffin May 10 '18 at 21:06

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

protected by [NullPointer](#) Jun 10 '13 at 5:05

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 [reputation](#) on this site (the [association bonus does not count](#)).

Would you like to answer one of these [unanswered questions](#) instead?

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).