Difference between variable declaration syntaxes in Javascript (including global variables)?

Asked 13 years, 10 months ago Modified 1 year, 8 months ago Viewed 272k times

305

Is there any difference between declaring a variable:

```
var a=0; //1
```

...this way:

```
a=0; //2
```

...or:

```
window.a=0; //3
```

in global scope?

javascript

Share
Improve this question
Follow

edited Nov 7, 2016 at 11:49
Steve Chambers
39.3k ● 29 ● 172 ● 220

asked Feb 1, 2011 at 11:53
Dan
57.7k ● 43 ● 122 ● 162

2 AFAIK var a = 0; does not work in IE when accessing the variable via another external js file which is declared in another js file – Aivan Monceller Feb 1, 2011 at 11:56

I donot know about window.a but the other 2 ways are the same in global scope. – programmer Feb 1, 2011 at 11:56

1 @AivanMonceller really? link please. – Raynos Feb 1, 2011 at 11:58

@Raynos, I experience it on my own website. IE6 to be specific. I could not get my var enum to appear which is on an external js file and i am referencing it as an inline javascript on an html file – Aivan Monceller Feb 1, 2011 at 12:00

@Ashwini In the global scope, window is the global object (in browsers). var a = 1; console.log(a); console.log(win – leebriggs Feb 1, 2011 at 12:08

5 Answers

Sorted by: Highest score (default)

581

Yes, there are a couple of differences, though in practical terms they're not usually big ones (except for your #2 — `a = 0;` — which A) I strongly recommend not doing, and B) is an error in strict mode).

There's a fourth way, and as of ES2015 (ES6) there's two more. I've added the fourth way at the end, but inserted the ES2015 ways after #1 (you'll see why), so we have:

```
var a = 0;                                  // 1
let a = 0;                                  // 1.1 (new with ES2015)
const a = 0;                                // 1.2 (new with ES2015)
a = 0;                                      // 2
window.a = 0; /*or*/ globalThis.a = 0; // 3
this.a = 0;                                 // 4
```

Those statements explained

1. `var a = 0;`

This creates a global variable which is also a property of the global object, which we access as `window` on browsers (or via the `globalThis` global added in ES2020, or via `this` at global scope). Unlike some other properties, the property cannot be removed via `delete`.

In specification terms, it creates an *identifier binding* on the Object Environment Record for the global environment. That makes it a property of the global object because the global object is where identifier bindings for the global environment's Object Environment Record are held. This is why the property is non-deletable: It's not just a simple property, it's an identifier binding, and identifiers can't be removed.

The binding (variable) is defined before the first line of code runs (see "When `var` happens" below).

The property this creates is enumerable (except on the very obsolete IE8 and earlier).

1.1 `let a = 0;`

This creates a global variable which is *not* a property of the global object. This is a new thing as of ES2015.

In specification terms, it creates an identifier binding on the *Declarative* Environment Record for the global environment rather than the *Object* Environment Record. The global environment is unique in having a split Environment Record, one for all the old stuff that goes on the global object (the *Object* Environment Record) and another for all the new stuff (`let`, `const`, and the functions created by `class`) that don't go on the global object, but go in the global environment's *Declarative* Environment Record instead.

The binding is *created* before any step-by-step code in its enclosing block is executed (in this case, before any global code runs), but it's not *accessible* in any way until the step-by-step execution reaches the `let` statement. Once execution reaches the `let` statement, the variable is accessible. (See "When `let` and `const` happen" below.) The time between the binding being *created* (on entry to the scope) and becoming *accessible* (code execution reaching the `let`) is called the *Temporal Dead Zone* [TMZ]. While the binding is in that state, any attempt to read from it or write to it is a runtime error.

(The specification's terminology for whether the binding is accessible is whether it's "initialized," but don't confuse that use of "initialized" with having an initializer on the `let` statement [`let a = 10;` vs. just `let a;`]; they're unrelated. The variable defined by `let a;` is initialized with `undefined` once the `let` is reached.)

1.2 `const a = 0;`

Creates a global constant, which is not a property of the global object.

A `const` binding is exactly like a `let` binding (including the TMZ and such) except it has a flag saying its value cannot be changed. One implication of that is you must provide an initializer (the `= value` part) to provide the initial (and never-changing) value for the `const`.

Using `const` does three things for you:

1. Makes it a runtime error if you try to assign to the constant (and most IDEs will flag it up for you more proactively than that).
2. Documents its unchanging nature for other programmers.
3. Lets the JavaScript engine optimize on the basis that the `const`'s value won't change (without having to track whether it's written to later or not — e.g., doesn't have to check if it's *effectively* constant).

It's important to understand that the `const`'s value never changing doesn't mean that an object the `const` refers to is immutable. It isn't. It just means that the value of the `const` can't be changed so it refers to a *different* object (or contains a primitive):

```
// This is fine:
const x1 = {a: 1};
```

```
console.log(x1.a); // 1
x1.a = 2;
//^^^^^^^--- No problem, just changing the object's state, not the value in the
`const` (the object reference)
console.log(x1.a); // 2

// This is not:
const x2 = {a: 1};
console.log(x2.a); // 1
x2 = {a: 2};
// ^------- Error here ("TypeError: Assignment to constant variable"),
// you can't change the value of a `const`
console.log(x2.a);
```

Run code snippet   Expand snippet

---

2 `a = 0;`

Don't do this. 😊 It's assigning to a completely undeclared identifier. In loose mode (the only mode before ES5), it creates a property on the global object *implicitly*. On my old blog, I call this *The Horror of Implicit Globals*. Thankfully, they fixed it with strict mode, added in ES5 and the default in new kinds of scopes (inside modules, inside `class` constructs, etc.). Strict mode makes assigning to an undeclared identifier the error it always should have been. It's one of several reasons to use strict mode.

Since it creates a normal property, you can `delete` it.

The property this creates is enumerable (except on the very obsolete IE8 and earlier).

---

3 `window.a = 0;` or `globalThis.a = 0;`

This creates a property on the global object explicitly, using the `window` global (on browsers) or the `globalThis` global that refers to the global object. As it's a normal property, you can delete it.

This property is enumerable (*even* on the very obsolete IE8 and earlier).

---

4 `this.a = 0;`

Exactly like #3, except we're referencing the global object through `this` instead of the globals `window` or `globalThis`. This works because `this` at global scope is the "global" `this` value. This is true even in strict mode. (Strict mode changes the `this` used when you call a function without supplying `this`, such as when you do `fn()`, but not what `this` is at global scope.) Note that it has to really be *global* scope. The top-level scope of modules is not global scope (it's *module* scope), and at module scope `this` is `undefined`.

Deleting properties

What do I mean by "deleting" or "removing" `a`? Exactly that: Removing the property (entirely) via the `delete` keyword:

```
window.a = 0;
console.log(`"a" in window? ${"a" in window}`); // "a" in window? true
delete window.a;
console.log(`"a" in window? ${"a" in window}`); // "a" in window? false
```

Run code snippet   Expand snippet

`delete` completely removes a property from an object. You can't do that with properties added to `window` indirectly via `var`, the `delete` is either silently ignored or throws an exception (depending on whether you're in strict mode).

(Minor note: The very obsolete IE8 and earlier, and the obsolete IE9-IE11 in their broken "compatibility" mode, wouldn't let you delete `window` properties even if you should have been allowed to.)

When `var` happens

Preface: `var` has no place in new code. Use `let` or `const` instead. But it's useful to understand `var` for the purposes of understanding old code you run across.

The variables defined via the `var` statement are created before *any* step-by-step code in the execution context is run, and so the variable (and its property on the global object) exists well *before* the `var` statement.

This can be confusing, so let's take a look. Here we have code trying to access `a` and `b`, followed by code in the middle creating them, and then code trying to access them again:

```
try {
    console.log(a);   // undefined
    console.log(b);   // ReferenceError: b is not defined
} catch (e) {
    console.error(e);
}
```

```
var a = "ayy";
b = "bee";            // Don't do this, but I didn't want to use `let` or
`const` in this example

try {
    console.log(a);   // "ayy"
    console.log(b);   // "bee"
} catch (e) {
    console.error(e);
}
```

Run code snippet   Expand snippet

As you can see, the identifier `a` is defined (with the value `undefined`) before the first line runs, but the identifier `b` isn't, so trying to read its value is a `ReferenceError`. The statement `var a = "ayy";` really deos *two* different things, at different times: On entry to the scope, it defines the identifier with the initial value `undefined` (the `var a` part), and later when it's reached in the execution of the code, it sets the value of `a` (the `a = "ayy"` part). Since `a` is defined before the first line of code runs, we can use it (and see its `undefined` value). This is known as "`var` hoisting" because the `var a` part is moved ("hoisted") to the top of the global scope or function scope where it appears, but the `a = "ayy"` part is left in its original location. (See *Poor misunderstood `var`* on my anemic old blog.)

When `let` and `const` happen

`let` and `const` are different from `var` in a couple of useful ways. The ways that are relevant to the question are A) that although the binding they define is created before any step-by-step code runs, it's not *accessible* until the `let` or `const` statement is reached; and B) as we've seen above, at global scope they don't create properties on the global object.

Re (A), while this using `var` runs:

```
console.log(a); // undefined
var a = 0;
console.log(a); // 0
```

Run code snippet   Expand snippet

This using `let` throws an error:

```
console.log(a); // ReferenceError: a is not defined
let a = 0;
console.log(a);
```

Run code snippet Expand snippet

The other two ways that `let` and `const` differ from `var`, which aren't really relevant to the question, are:

1. `var` always applies to the entire execution context (throughout global code, or throughout function code in the function where it appears; it jumps out of blocks), but `let` and `const` apply only within the *block* where they appear. That is, `var` has function (or global) scope, but `let` and `const` have *block* scope.
2. Repeating `var a` in the same context is harmless, but if you have `let a` (or `const a`), having another `let a` or a `const a` or a `var a` is a syntax error.

Here's an example demonstrating that `let` and `const` take effect immediately in their block before any code within that block runs, but aren't accessible until the `let` or `const` statement:

```
let a = 0;          // (Doesn't matter whether this is `let`, `const`, or `var
[or even `class` or `function`])
console.log(a);     // 0
if (true) {
    console.log(a); // ReferenceError: a is not defined
    let a = 1;
    console.log(a);
}
```

Run code snippet Expand snippet

Note that the second `console.log` fails, instead of accessing the `a` from outside the block, because within that block `a` refers to the `a` declared later in the block. But the `console.log` statement occurs within the Temporal Dead Zone for that inner `a`, so it causes an error.

Avoid cluttering global scope - use modules

Global scope is very, very cluttered. It has (at least):

- Lots of global variables created via the spec (like `undefined` and `NaN` which, oddly, are globals rather than keywords; miscellanous global functions)
- (On browsers) Variables for all DOM elements with an `id` and many with a `name` (provided the `id`/`name` value is a valid identifier; otherwise, they're just properties on `window` but not global variables)
- (On browsers) Variables for `window`-specific things, like `name`, `location`, `self`...
- Variables for all global-scope `var` statements
- Variables for all global-scope `let`, `const`, and `class` statements

All of those globals are ripe with opportunities for conflicts with your code, such as this classic example on browsers:

```
var name = 42;
console.log(name);          // 42 - seems okay, but...
console.log(typeof name);   // ...string?!?!!
```

Run code snippet   Expand snippet

Why is `name` a string? Because it's an accessor property on `window` for the *name* of the window object, which is always a string. (The equivalent with `let` would work as expected, since the declarative environment record is conceptually nested within the object environment record, so the `name` binding created with `let` shadows the `name` binding for the accessor property.)

Whenever possible, don't add to the mess. Use modules instead. Top-level scope in modules is *module scope*, not global scope, so only other code in your module sees those top-level declarations. You can share information between modules via `export` and `import`.

Before modules, we used "scoping" functions wrapped around our code:

Show code snippet

Modules make that obsolete.

Share   edited Mar 28, 2023 at 12:48   answered Feb 1, 2011 at 12:00

Improve this answer   T.J. Crowder

Follow   1.1m ● 199 ● 2k ● 1.9k

can i do `window['a']=0` to make it clear i'm using window as a map? is `window` special such that some browsers don't allow this and force me to use `window.a`? – Jayen Feb 27, 2015 at 1:25

One note on **#3** that's probably worth clarifying: `window.a = 0;` only works in browser environments, and only by convention. Binding the global object to a variable named `window` is not in the ES Spec and so will not work in, for example, V8 or Node.js, while `this.a = 0;` (when invoked in the global execution context) will work in any environment since the spec *does* specify that there must be *a* global object. If wrapping your code in an IIFE as in the **Off-topic** section, you can pass `this` as a parameter named `window` or `global` to get a direct reference to the global object. – Sherlock_HJ Apr 2, 2016 at 21:08

@Sherlock_HJ: I've added "on browsers;" that is earlier in the answer as well, but I added it in case people skip down to that. It is in the spec now; while it's only in passing, you won't find a browser that doesn't do it. I'm a bit surprised it's not in Annex B. – T.J. Crowder Apr 3, 2016 at 7:23

@T.J.Crowder, So, a global variable declared with `var a = 0;` automatically becomes a property of the global object. If I declare `var b = 0;` within a function declaration, will it also be a property of some underlying object? – ezpresso May 15, 2016 at 20:49

1 I like the sentence **It's not just a simple property, it's an identifier binding** – Gangadhar Jannu Jan 23, 2017 at 10:29

40

Keeping it simple :

```
a = 0
```

The code above gives a global scope variable

```
var a = 0;
```

This code will give a variable to be used in the current scope, and under it

```
window.a = 0;
```

This generally is same as the global variable.

Share

Improve this answer

Follow

edited Feb 1, 2011 at 12:22

answered Feb 1, 2011 at 12:15

Umair Jabbar
3,666 ● 5 ● 31 ● 42

Your statements *"The code above gives a global scope variable"* and *"This code will give a variable to be used in the current scope, and under it"*, taken together, suggest that you can't use the first line and access `a` *under* the current scope. You can. Also, your use of "global variable" is a bit off -- the two places you say "global variable" are no more global than the place you don't say it. – T.J. Crowder Feb 1, 2011 at 12:34

global itself means that you can access/read/write the variable anywhere, including the place where i mentioned current scope, that is so obvious. And if you suggest that window.a and 'a' wont be global in the script then you are 100% wrong. – Umair Jabbar Feb 1, 2011 at 12:52

3 @Umair: *"global itself means that you can access/read/write the variable anywhere"* Right. Again, you seem to be calling out the first and last as more "global" than the middle, which of course they aren't. – T.J. Crowder Feb 1, 2011 at 13:00

4 the middle one is considered to be used inside a function, all of them would be same if used under the main scope. using var inside a function was my assumption – Umair Jabbar Feb 1, 2011 at 13:18

4 @Umair: *"using var inside a function was my assumption"* Ah, okay. But that's not the question. The question very clearly says *"in global scope"*. If you're going to change the assumption (which is fair enough, to expand and explain a more general point), you'll need to be clear that's what you're doing in your answer. – T.J. Crowder Feb 1, 2011 at 13:26

10

```
<title>Index.html</title>
<script>
    var varDeclaration = true;
    noVarDeclaration = true;
    window.hungOnWindow = true;
    document.hungOnDocument = true;
</script>
<script src="external.js"></script>

/* external.js */

console.info(varDeclaration == true); // could be .log, alert etc
// returns false in IE8

console.info(noVarDeclaration == true); // could be .log, alert etc
// returns false in IE8

console.info(window.hungOnWindow == true); // could be .log, alert etc
// returns true in IE8

console.info(document.hungOnDocument == true); // could be .log, alert etc
// returns ??? in IE8 (untested!)  *I personally find this more clugy than
hanging off window obj
```

Is there a global object that all vars are hung off of by default? eg: 'globals.noVar declaration'

Share
Improve this answer
Follow

edited Mar 20, 2017 at 20:18
user719662

answered Nov 7, 2012 at 20:12
Cody
9,995 ● 4 ● 64 ● 47

Very nice exploration. The definite guide to using `window.*` declaration. This declaration looks most safe against copy-pasting your code, and clear also. – Dan Nov 20, 2012 at 19:33

8

Bassed on the excellent answer of **T.J. Crowder**: (**Off-topic: Avoid cluttering `window`**)

This is an example of his idea:

Html

```
<!DOCTYPE html>
<html>
  <head>
    <script type="text/javascript" src="init.js"></script>
    <script type="text/javascript">
      MYLIBRARY.init(["firstValue", 2, "thirdValue"]);
    </script>
    <script src="script.js"></script>
  </head>

  <body>
    <h1>Hello !</h1>
  </body>
</html>
```

init.js (Based on this answer)

```
var MYLIBRARY = MYLIBRARY || (function(){
    var _args = {}; // private

    return {
        init : function(Args) {
            _args = Args;
            // some other initialising
        },
        helloWorld : function(i) {
            return _args[i];
        }
    };
}());
```

script.js

```
// Here you can use the values defined in the html as if it were a global
variable
var a = "Hello World " + MYLIBRARY.helloWorld(2);

alert(a);
```

Here's the plnkr. Hope it help !

Share
Improve this answer
Follow

edited May 23, 2017 at 12:18
Community Bot
1 ● 1

answered Dec 7, 2015 at 14:29
robe007
3,889 ● 4 ● 37 ● 61

5

In global scope there is no semantic difference.

But you really should avoid `a=0` since your setting a value to an undeclared variable.

Also use closures to avoid editing global scope at all

```
(function() {
   // do stuff locally

   // Hoist something to global scope
   window.someGlobal = someLocal
}());
```

Always use closures and always hoist to global scope when its absolutely neccesary. You should be using asynchronous event handling for most of your communication anyway.

As @AvianMoncellor mentioned there is an IE bug with `var a = foo` only declaring a global for file scope. This is an issue with IE's notorious broken interpreter. This bug does sound familiar so it's probably true.

So stick to `window.globalName = someLocalpointer`

Share Improve this answer Follow

answered Feb 1, 2011 at 11:56
Raynos
169k ● 57 ● 356 ● 398

2 *"In global scope there is no semantic difference."* Actually, there's a huge semantic difference, the mechanisms by which the property gets defined are completely different -- but in practical terms it boils down to only a small *actual* difference (in that you can't `delete` a `var`). – T.J. Crowder Feb 1, 2011 at 12:08

@T.J. Crowder I didn't know that. I thought variable declaration was setting properties on the variable object. Didn't know those couldn't be deleted. – Raynos Feb 1, 2011 at 12:42

Yup. They're also defined earlier if you use `var`. They're just completely different mechanisms that have much the same practical result. :-) – T.J. Crowder Feb 1, 2011 at 12:44

@T.J. Crowder I forgot to mention that `var` jumps to the stop of scope. – Raynos Feb 1, 2011 at 12:58