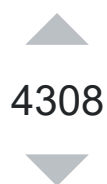


Make your voice heard. [Take the 2020 Developer Survey now.](#)

# What's the difference between using “let” and “var”?

Asked 10 years, 9 months ago   Active 10 days ago   Viewed 1.2m times



4308



1151



ECMAScript 6 introduced [the let statement](#).

I've heard it that it's described as a "local" variable, but I'm still not quite sure how it behaves differently than the `var` keyword.

What are the differences? When should `let` be used over `var` ?

javascript

scope

ecmascript-6

var

let

edited Mar 18 '19 at 21:21



ivanleoncz

3,288

3

32

39

asked Apr 17 '09 at 20:09



TM.

84.1k

28

114

125

102   ECMAScript is the standard and `let` is included in the [6th edition draft](#) and will most likely be in the final specification. – [Richard Ayotte](#) Mar 31 '12 at 15:08

5   See [kangax.github.io/es5-compat-table/es6](http://kangax.github.io/es5-compat-table/es6) for an up to date support matrix of ES6 features (including `let`). At the time of writing Firefox, Chrome and IE11 all support it (although I believe FF's implementation is not quite standard). – [Nico Burns](#) Jan 17 '14 at 12:37

22   For the longest time I did not know that vars in a for loop were scoped to the function it was wrapped in. I remember figuring this out for the first time and thought it was very stupid. I do see some power though knowing now how the two could be used for different reason and how in some cases you might actually want to use a `var` in a for loop and not have it scoped to the block. – [Eric Bishard](#) May 7 '15 at 13:54

As ES6 feature support improves, the question concerning ES6 adoption shifts focus from feature support to performance differences. As such, [here's a site I found benchmarking performance differences between ES6 and ES5](#). Keep in mind this will likely change over time as engines optimize for ES6 code. – [timolawl](#) May 4 '16 at 1:02 ✎

This is a very good reading [wesbos.com/javascript-scoping](http://wesbos.com/javascript-scoping) – [onmyway133](#) Jun 15 '17 at 9:32 ✎

34 Answers



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



1 2 next

## Scoping rules

5828 Main difference is scoping rules. Variables declared by `var` keyword are scoped to the immediate function body (hence the function scope) while `let` variables are scoped to the immediate *enclosing* block denoted by `{ }` (hence the block scope).



```
function run() {  
  var foo = "Foo";  
  let bar = "Bar";  
  
  console.log(foo, bar);  
  
  {  
    let baz = "Bazz";  
    console.log(baz);  
  }  
  
  console.log(baz); // ReferenceError  
}  
  
run();
```

The reason why `let` keyword was introduced to the language was function scope is confusing and was one of the main sources of bugs in JavaScript.

Take a look at this example from [another stackoverflow question](#):

```
var funcs = [];  
// Let's create 3 functions  
for (var i = 0; i < 3; i++) {  
  // and store them in funcs  
  funcs[i] = function() {  
    // each should log its value.  
    console.log("My value: " + i);  
  };  
}  
for (var j = 0; j < 3; j++) {  
  // and now let's run each one to see  
  funcs[j]()  
}
```

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



My value: 3 was output to console each time `funcs[j]()`; was invoked since anonymous functions were bound to the same variable.

People had to create immediately invoked functions to capture correct value from the loops but that was also hairy.

## Hoisting

While variables declared with `var` keyword are "hoisted" to the top of the block which means they are accessible in their enclosing scope even before they are declared:

```
function run() {  
  console.log(foo); // undefined  
  var foo = "Foo";  
  console.log(foo); // Foo  
}  
  
run();
```

`let` variables are not initialized until their definition is evaluated. Accessing them before the initialization results in a `ReferenceError`. Variable said to be in "temporal dead zone" from the start of the block until the initialization is processed.

```
function checkHoisting() {  
  console.log(foo); // ReferenceError  
  let foo = "Foo";  
  console.log(foo); // Foo  
}  
  
checkHoisting();
```

## Creating global object property

At the top level, `let`, unlike `var`, does not create a property on the global object:

```
var foo = "Foo"; // globally scoped  
let bar = "Bar"; // globally scoped  
  
console.log(window.foo); // Foo  
console.log(window.bar); // undefined
```

In strict mode, `var` will let you re-declare the same variable in the same scope while `let` raises a `SyntaxError`.

```
'use strict';
var foo = "foo1";
var foo = "foo2"; // No problem, 'foo' is replaced.

let bar = "bar1";
let bar = "bar2"; // SyntaxError: Identifier 'bar' has already been declared
```

edited Jan 13 at 15:53

community wiki  
30 revs, 20 users 30%  
ThinkingStiff

- 
- 16 Remember you can create block whenever you want. `function() { code;{ let inBlock = 5; } code; };` – [average Joe](#) Dec 14 '12 at 10:14
- 
- 165 So is the purpose of `let` statements only to free up memory when not needed in a certain block? – [NoBugs](#) Jun 7 '13 at 5:18
- 
- 205 @NoBugs, Yes, and it is encouraged that variables are existent only where they are needed. – [batman](#) Jun 7 '13 at 15:02
- 
- 64 `let` block expression `let (variable declaration) statement` is non-standard and will be removed in future, [bugzilla.mozilla.org/show\\_bug.cgi?id=1023609](https://bugzilla.mozilla.org/show_bug.cgi?id=1023609). – [Gajus](#) Dec 17 '14 at 14:51
- 
- 13 So, I just cannot think of any case where using `var` is of any use. Could someone give me an example of a situation where it's preferable to use `var`? – [Luis Sieira](#) Nov 8 '15 at 13:12
- 

`let` can also be used to avoid problems with closures. It binds fresh value rather than keeping an old reference as shown in examples below.

597

```
for(var i=1; i<6; i++) {
  $("#div" + i).click(function () { console.log(i); });
}
```

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



```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js">
</script>
<p>Clicking on each number will log to console:</p>
<div id="div1">1</div>
<div id="div2">2</div>
<div id="div3">3</div>
<div id="div4">4</div>
<div id="div5">5</div>
```

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

Code above demonstrates a classic JavaScript closure problem. Reference to the `i` variable is being stored in the click handler closure, rather than the actual value of `i`.

Every single click handler will refer to the same object because there's only one counter object which holds 6 so you get six on each click.

A general workaround is to wrap this in an anonymous function and pass `i` as an argument. Such issues can also be avoided now by using `let` instead `var` as shown in the code below.

(Tested in Chrome and Firefox 50)

```
for(let i=1; i<6; i++) {
  $("#div" + i).click(function () { console.log(i); });
}
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js">
</script>
<p>Clicking on each number will log to console:</p>
<div id="div1">1</div>
<div id="div2">2</div>
<div id="div3">3</div>
<div id="div4">4</div>
<div id="div5">5</div>
```

[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](#).



edited Oct 29 '19 at 8:47

answered May 27 '15 at 10:16



Nash Bridges

2,260 10 17



Gurpreet Singh

17.4k 5 36 54

- 50 That is actually cool. I would expect "i" to be defined outside the loop body contains within brackets and to NOT form a "closure" around "i". Of course your example proves otherwise. I think it is a bit confusing from the syntax point of view but this scenario is so common it makes sense to support it in that way. Many thanks for bringing this up. – [Karol Kolenda](#) Jul 27 '15 at 12:49
- 9 IE 11 supports `let`, but it alerts "6" for all the buttons. Do you have any source saying how `let` is supposed to behave? – [Jim Hunziker](#) Oct 22 '15 at 13:29
- 10 Looks like your answer is the correct behavior: [developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/...](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/...) – [Jim Hunziker](#) Oct 22 '15 at 13:32
- 11 Indeed this is a common pitfall in Javascript and now I can see why `let` would be really useful. Setting event listeners in a loop no longer requires an immediately invoked function expression for locally scoping `i` at each iteration. – [Adrian Moisa](#) Feb 21 '16 at 8:12
- 19 The use of "let" just defers this problem. So each iteration creates a private independent block scope, but the "i" variable can still be corrupted by subsequent changes within the block, (granted the iterator variable is not *usually* changed within the block, but other declared let variables within the block may well be) and any function declared within the block can, when invoked, corrupt the value of "i" for other functions declared within the block because they *do* share the same private block scope hence the same reference to "i". – [gary](#) Sep 7 '16 at 23:10

## What's the difference between `let` and `var`?

175

- A variable defined using a `var` statement is known throughout [the function](#) it is defined in, from the start of the function. (\*)
- A variable defined using a `let` statement is only known in [the block](#) it is defined in, from the moment it is defined onward. (\*\*)

To understand the difference, consider the following code:

```
// i IS NOT known here
// j IS NOT known here
// k IS known here, but undefined
// l IS NOT known here

function loop(arr) {
  // i IS known here, but undefined
  // j IS NOT known here
  // k IS known here, but has a value only the second time loop is called
  // l IS NOT known here
```

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



```
// k IS known here, but has a value only the second time loop is called
// l IS NOT known here
};

// i IS known here, and has a value
// j IS NOT known here
// k IS known here, but has a value only the second time loop is called
// l IS NOT known here

for( let j = 0; j < arr.length; j++ ) {
    // i IS known here, and has a value
    // j IS known here, and has a value
    // k IS known here, but has a value only the second time loop is called
    // l IS NOT known here
};

// i IS known here, and has a value
// j IS NOT known here
// k IS known here, but has a value only the second time loop is called
// l IS NOT known here
}

loop([1,2,3,4]);

for( var k = 0; k < arr.length; k++ ) {
    // i IS NOT known here
    // j IS NOT known here
    // k IS known here, and has a value
    // l IS NOT known here
};

for( let l = 0; l < arr.length; l++ ) {
    // i IS NOT known here
    // j IS NOT known here
    // k IS known here, and has a value
    // l IS known here, and has a value
};

loop([1,2,3,4]);

// i IS NOT known here
// j IS NOT known here
// k IS known here, and has a value
// l IS NOT known here
```

Also, consider that block scoped variables are not known before they are declared because they are not hoisted. You're also not allowed to redeclare the same block scoped variable within the same block. This makes block scoped variables less error prone than globally or functionally scoped variables, which are hoisted and which do not produce any errors in case of multiple declarations.

## Is it safe to use `let` today?

Some people would argue that in the future we'll ONLY use `let` statements and that `var` statements will become obsolete. JavaScript guru [Kyle Simpson](#) wrote [a very elaborate article on why he believes that won't be the case](#).

Today, however, that is definitely not the case. In fact, we need actually to ask ourselves whether it's safe to use the `let` statement. The answer to that question depends on your environment:

- If you're writing server-side JavaScript code ([Node.js](#)), you can safely use the `let` statement.
- If you're writing client-side JavaScript code and use a browser based transpiler (like [Traceur](#) or [babel-standalone](#)), you can safely use the `let` statement, however your code is likely to be anything but optimal with respect to performance.
- If you're writing client-side JavaScript code and use a Node based transpiler (like the [traceur shell script](#) or [Babel](#)), you can safely use the `let` statement. And because your browser will only know about the transpiled code, performance drawbacks should be limited.
- If you're writing client-side JavaScript code and don't use a transpiler, you need to consider browser support.

There are still some browsers that don't support `let` at all :



IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Blackberry Browser
			4-18		10-12.1				
			<sup>2</sup> 19-40	3.1-9.1	<sup>2</sup> 15-27	3.2-9.3			
		<sup>1</sup> 2-43	<sup>3</sup> 41-48	<sup>4</sup> 10-10.1	<sup>3</sup> 28-35	<sup>4</sup> 10-10.3			
6-10	12-17	44-65	49-72	11-12	36-57	11-11.4		2.1-4.4.4	7
<sup>5</sup> 11	18	66	73	12.1	58	12.1	all	67	10
		67-68	74-76	TP		12.2			

<sup>1</sup> Supports a non-standard version that can only be used in script elements with a type attribute of `application/javascript;version=1.7`. As other browsers do not support these types of script tags this makes support useless for cross-browser support.

<sup>2</sup> Requires the 'Experimental JavaScript features' flag to be enabled

<sup>3</sup> Only supported in strict mode

<sup>4</sup> `let` bindings in for loops are incorrectly treated as function-scoped instead of block scoped.

<sup>5</sup> `let` variables are not bound separately to each iteration of for loops

## How to keep track of browser support

For an up-to-date overview of which browsers support the `let` statement at the time of your reading this answer, see [this Can I Use page](#).

(\*) Globally and functionally scoped variables can be initialized and used before they are declared because JavaScript variables are [hoisted](#). This means that declarations are always much to the top of the scope.

(\*\*) Block scoped variables are not hoisted

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





32.9k 14 165 141

- 14 regarding answer v4: `let` is known everywhere in the function-block! It starts as `undefined` (due to hoisting) until you assign a value! ps: `let` is also hoisted (to the top of it's containing block), but will give a `ReferenceError` when referenced in the block before first assignment. (ps2: I'm a pro-semicolon kinda guy but you really don't need a semicolon after a block ). That being said, thanks for adding the reality-check regarding support! – [GitaarLAB](#) May 21 '16 at 4:41

@GitaarLAB : According to the [Mozilla Developer Network](#) : "In ECMAScript 2015, `let` bindings are not subject to Variable Hoisting, which means that `let` declarations do not move to the top of the current execution context." - Anyway, I made a few improvements to my answer that should clarify the difference in hoisting behavior between `let` and `var` ! – [John Slegers](#) Feb 26 '18 at 23:37

- 1 Your answer improved a lot (I thoroughly checked). Note that same link you referenced in your comment also says: "The (`let`) variable is in a "temporal dead zone" from the *start of the block* until the initialization is processed." That means that the 'identifier' (the text-string 'reserved' to point to 'something') *is already* reserved in the relevant scope, otherwise it would become part of the root/host/window scope. To me personally, 'hoisting' means nothing more than reserving/linking declared 'identifiers' to their relevant scope; excluding their initialization/assignment/modifyability! – [GitaarLAB](#) Mar 1 '18 at 18:16

And..+1. That Kyle Simpson article you linked is an *excellent* read, thank you for that! It is also clear about the "temporal dead zone" aka "TDZ". One interesting thing I'd like to add: I've read on MDN that `let` and `const` were *recommended to only use when you actually need their additional functionality*, because enforcing/checking these extra features (like write-only `const`) result in 'more work' (and additional scope-nodes in the scope-tree) for the (current)engine(s) to enforce/check/verify/setup. – [GitaarLAB](#) Mar 1 '18 at 18:17

- 1 Note that MDN says that IE DOES interpret `let` correctly. Which is it? [developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/...](#) – [Katinka Hesselink](#) Feb 6 '19 at 12:42

Here's an [explanation of the `let` keyword](#) with some examples.

145 `let` works very much like `var` . The main difference is that the scope of a `var` variable is the entire enclosing function

[This table](#) on Wikipedia shows which browsers support Javascript 1.7.



Note that only Mozilla and Chrome browsers support it. IE, Safari, and potentially others don't.

edited Jun 24 '19 at 2:52



Jack Bashford

34.8k 7 29 54

answered Apr 17 '09 at 20:11



Ben S

62.8k 27 159 207



50 While it's technically correct to say IE does not support it, it's more correct to say that it's a mozilla only extension. – [olliej](#) Apr 17 '09 at 22:56

54 @olliej, actually Mozilla is just ahead of the game. See page 19 of [ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf](http://ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf) – [Tyler Crompton](#) Jun 18 '12 at 20:16

@TylerCrompton that's just the set of words that have been reserved for years. When mozilla added let it was purely a mozilla extension, with no related spec. ES6 should define behaviour for let statements, but that came after mozilla introduced the syntax. Remember moz also has E4X, which is entirely dead and moz only. – [olliej](#) Jul 11 '12 at 18:49

9 IE11 added support for let [msdn.microsoft.com/en-us/library/ie/dn342892%28v=vs.85%29.aspx](http://msdn.microsoft.com/en-us/library/ie/dn342892%28v=vs.85%29.aspx) – [eloyesp](#) Dec 24 '13 at 12:59

The accepted answer is missing a point:

110

```
{
  let a = 123;
};
```

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

edited Jul 14 '16 at 14:13



[William](#)

393 7 20

answered Jun 2 '15 at 20:59



[Lcf.vs](#)

1,376 1 9 14

18 The accepted answer does NOT explain this point in its example. The accepted answer only demonstrated it in a for loop initializer, dramatically narrowing the scope of application of the limitations of let . Upvoted. – [Jon Davis](#) Sep 22 '15 at 6:55 ✎

35 @stimp77 It explicitly states "let is scoped to the nearest enclosing block"; does every way that manifests need to be included? – [Dave Newton](#) Mar 31 '16 at 21:32 ✎

6 there were a lot of examples and none of them properly demonstrated the matter .. I might've upvoted both the accepted answer and this one? – [Jon Davis](#) Mar 31 '16 at 21:38 ✎

5 This contribution demonstrates that a "block" can simply be a set of lines enclosed in brackets; i.e. it doesn't need to be associated with any sort of control flow, loop, etc. – [webelo](#) Nov 22 '17 at 14:37

let

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





Variables declared using the `let` keyword are block-scoped, which means that they are available only in the [block](#) in which they were declared.



## At the top level (outside of a function)

At the top level, variables declared using `let` don't create properties on the global object.

```
var globalVariable = 42;
let blockScopedVariable = 43;

console.log(globalVariable); // 42
console.log(blockScopedVariable); // 43

console.log(this.globalVariable); // 42
console.log(this.blockScopedVariable); // undefined
```

## Inside a function

Inside a function (but outside of a block), `let` has the same scope as `var`.

```
(() => {
  var functionScopedVariable = 42;
  let blockScopedVariable = 43;

  console.log(functionScopedVariable); // 42
  console.log(blockScopedVariable); // 43
})();

console.log(functionScopedVariable); // ReferenceError: functionScopedVariable is not
defined
console.log(blockScopedVariable); // ReferenceError: blockScopedVariable is not defined
```

## Inside a block

Variables declared using `let` inside a block can't be accessed outside that block.

```
{
  var globalVariable = 42;
  let blockScopedVariable = 43;
}
```

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



```
console.log(globalVariable); // 42
console.log(blockScopedVariable); // ReferenceError: blockScopedVariable is not defined
```

## Inside a loop

Variables declared with `let` in loops can be referenced only inside that loop.

```
for (var i = 0; i < 3; i++) {
  var j = i * 2;
}
console.log(i); // 3
console.log(j); // 4

for (let k = 0; k < 3; k++) {
  let l = k * 2;
}
console.log(typeof k); // undefined
console.log(typeof l); // undefined
// Trying to do console.log(k) or console.log(l) here would throw a ReferenceError.
```

## Loops with closures

If you use `let` instead of `var` in a loop, with each iteration you get a new variable. That means that you can safely use a closure inside a loop.

```
// Logs 3 thrice, not what we meant.
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 0);
}

// Logs 0, 1 and 2, as expected.
for (let j = 0; j < 3; j++) {
  setTimeout(() => console.log(j), 0);
}
```

## Temporal dead zone

Because of [the temporal dead zone](#), variables declared using `let` can't be accessed before they are declared. Attempting to do so

..

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



```
console.log(noTDZ); // undefined
var noTDZ = 43;
console.log(hasTDZ); // ReferenceError: hasTDZ is not defined
let hasTDZ = 42;
```

## No re-declaring

You can't declare the same variable multiple times using `let`. You also can't declare a variable using `let` with the same identifier as another variable which was declared using `var`.

```
var a;
var a; // Works fine.

let b;
let b; // SyntaxError: Identifier 'b' has already been declared

var c;
let c; // SyntaxError: Identifier 'c' has already been declared
```

`const`

`const` is quite similar to `let` —it's block-scoped and has TDZ. There are, however, two things which are different.

## No re-assigning

Variable declared using `const` can't be re-assigned.

```
const a = 42;
a = 43; // TypeError: Assignment to constant variable.
```

Note that it doesn't mean that the value is immutable. Its properties still can be changed.

```
const obj = {};
obj.a = 42;
console.log(obj.a); // 42
```

## Initializer is required

You always must specify a value when declaring a variable using `const` .

```
const a; // SyntaxError: Missing initializer in const declaration
```

edited Oct 25 '18 at 20:54



ketchupisred

611 3 16

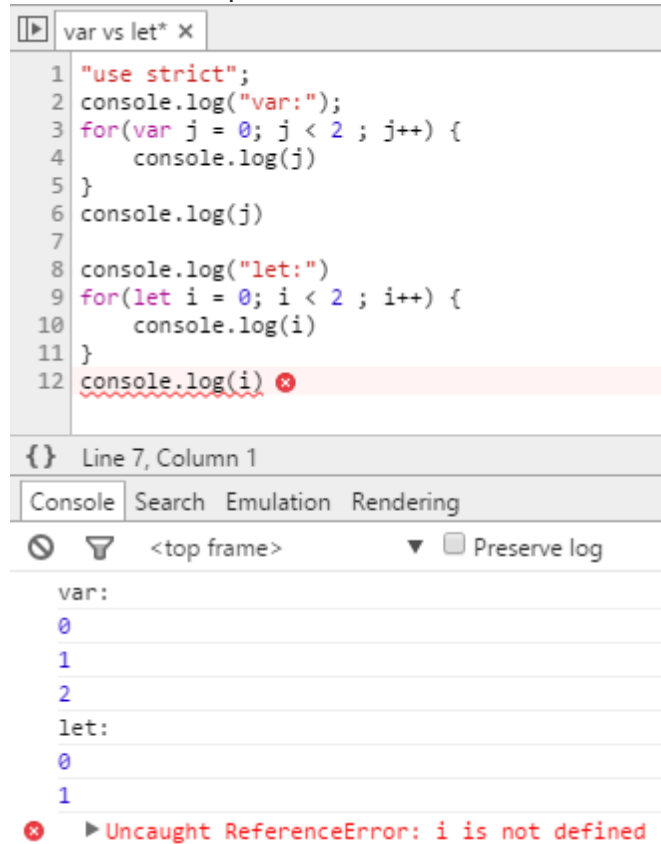
answered Nov 23 '16 at 22:52



Michał Perłakowski

55k 20 124 136

Here is an example for the difference between the two (support just started for chrome):



```
var vs let* X
1 "use strict";
2 console.log("var:");
3 for(var j = 0; j < 2 ; j++) {
4   console.log(j)
5 }
6 console.log(j)
7
8 console.log("let:")
9 for(let i = 0; i < 2 ; i++) {
10   console.log(i)
11 }
12 console.log(i) X
```

{ } Line 7, Column 1

Console Search Emulation Rendering

<top frame> Preserve log

var:

0

1

2

let:

0

1

Uncaught ReferenceError: i is not defined

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



```
"use strict";
console.log("var:");
for (var j = 0; j < 2; j++) {
  console.log(j);
}

console.log(j);

console.log("let:");
for (let i = 0; i < 2; i++) {
  console.log(i);
}

console.log(i);
```

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

edited Jun 5 '19 at 12:03

answered Mar 6 '15 at 10:41



vlio20

7,144 14 78 151

2 What tool am I looking at here? – Barton Mar 24 '15 at 21:43

20 Chrome devtools – vlio20 Mar 24 '15 at 21:44

As a developer of desktop applets for Cinnamon, I haven't been exposed to such shiny tools. – Barton Oct 26 '17 at 4:20



47



There are some subtle differences — `let` scoping behaves more like variable scoping does in more or less any other languages.

e.g. It scopes to the enclosing block, They don't exist before they're declared, etc.

However it's worth noting that `let` is only a part of newer Javascript implementations and has varying degrees of [browser support](#).

edited Jun 24 '19 at 2:53



Jack Bashford

answered Apr 17 '09 at 21:38




olliej

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





- 11 It's also worth noting that ECMAScript is the standard and `let` is included in the [9th Edition draft](#) and will most likely be in the final specification. – [Richard Ayotte](#) Mar 31 '12 at 15:09
- 
- 23 That's the difference 3 years makes :D – [olliej](#) Apr 13 '12 at 3:28
- 
- 4 Just stubled across this question and in 2012 it is still the case that only Mozilla browsers support `let` . Safari, IE, and Chome all don't. – [pseudosavant](#) Jul 13 '12 at 17:38
- 
- 2 The idea of accidentally creating partial block scope on accident is a good point, beware, `let` does not hoist, to use a variable defined by a `let` defined at the top of your block. If you have an `if` statement that is more than just a few lines of code, you may forget that you cannot use that variable until after it is defined. GREAT POINT!!! – [Eric Bishard](#) May 7 '15 at 14:01 
- 
- 2 @EricB: yes and no: "In ECMAScript 2015, `let` **will hoist** the variable to the top of the block. However, referencing the variable in the block before the variable declaration results in a *ReferenceError* (my note: instead of good old `undefined` ). The variable is in a 'temporal dead zone' from the start of the block until the declaration is processed." Same goes for "switch statements because there is only one underlying block". Source: [developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/...](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/...) – [GitaarLAB](#) May 21 '16 at 4:15
- 

The main difference is the **scope** difference, while **let** can be only available inside the **scope** it's declared, like in for loop, **var** can be accessed outside the loop for example. From the documentation in [MDN](#) (examples also from MDN):

26

**let** allows you to declare variables that are limited in scope to the block, statement, or expression on which it is used. This is unlike the **var** keyword, which defines a variable globally, or locally to an entire function regardless of block scope.

Variables declared by **let** have as their scope the block in which they are defined, as well as in any contained sub-blocks. In this way, **let** works very much like **var**. The main difference is that the scope of a **var** variable is the entire enclosing function:

```
function varTest() {
  var x = 1;
  if (true) {
    var x = 2; // same variable!
    console.log(x); // 2
  }
  console.log(x); // 2
}

function letTest() {
  let x = 1;
  if (true) {
    let x = 2; // different variable
    console.log(x); // 2
  }
}
```

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



At the top level of programs and functions, **let**, unlike **var**, does not create a property on the global object. For example:

```
var x = 'global';
let y = 'global';
console.log(this.x); // "global"
console.log(this.y); // undefined
```

When used inside a block, **let** limits the variable's scope to that block. Note the difference between **var** whose scope is inside the function where it is declared.

```
var a = 1;
var b = 2;

if (a === 1) {
  var a = 11; // the scope is global
  let b = 22; // the scope is inside the if-block

  console.log(a); // 11
  console.log(b); // 22
}

console.log(a); // 11
console.log(b); // 2
```

Also don't forget it's ECMA6 feature, so it's not fully supported yet, so it's better always transpiles it to ECMA5 using Babel etc... for more info about visit [babel website](https://babeljs.io/)

edited Jan 18 '19 at 7:03

answered Mar 22 '17 at 14:39



Alireza

67.4k

18

208

138

## • Variable Not Hoisting

~~let will not hoist to the entire scope of the block they appear in. By contrast, var could hoist as below.~~

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





```

    var cc = 23;
  }

  {
    console.log(bb); // ReferenceError: bb is not defined
    let bb = 23;
  }

```

Actually, Per @Bergi, [Both var and let are hoisted.](#)

## • Garbage Collection

Block scope of `let` is useful relates to closures and garbage collection to reclaim memory. Consider,

```

function process(data) {
  //...
}

var hugeData = { .. };

process(hugeData);

var btn = document.getElementById("mybutton");
btn.addEventListener( "click", function click(evt){
  //....
});

```

The `click` handler callback does not need the `hugeData` variable at all. Theoretically, after `process(..)` runs, the huge data structure `hugeData` could be garbage collected. However, it's possible that some JS engine will still have to keep this huge structure, since the `click` function has a closure over the entire scope.

However, the block scope can make this huge data structure to garbage collected.

```

function process(data) {
  //...
}

{ // anything declared inside this block can be garbage collected
  let hugeData = { .. };
  process(hugeData);
}

```



```
//....  
});
```

- **let loops**

`let` in the loop can **re-binds it** to each iteration of the loop, making sure to re-assign it the value from the end of the previous loop iteration. Consider,

```
// print '5' 5 times  
for (var i = 0; i < 5; ++i) {  
  setTimeout(function () {  
    console.log(i);  
  }, 1000);  
}
```

However, replace `var` with `let`

```
// print 1, 2, 3, 4, 5. now  
for (let i = 0; i < 5; ++i) {  
  setTimeout(function () {  
    console.log(i);  
  }, 1000);  
}
```

Because `let` create a new lexical environment with those names for a) the initialiser expression b) each iteration (previously to evaluating the increment expression), more details are [here](#).

edited May 23 '17 at 12:34




1 1

answered Jan 17 '16 at 15:11

undefined [zangw](#)

**29.5k** 8 104 128

4 Yip they are hoisted, but behave as if not hoisted because of the (drum roll) Temporal Dead Zone - a very dramatic name for an identifier not being accessible until it's declared:-) – [Drenai](#) Dec 31 '16 at 15:42 

So let is hoisted, but unavailable? How is that different than 'not hoisted'? – [N-ate](#) Nov 21 '17 at 21:46

Hopefully Brian or Bergi come back to answer this. Is the declaration of let hoisted, but not the assignment? Thanks! – [N-ate](#) Nov 22 '17 at 16:18

1 @N-ate, Here is [one post](#) of Bergi, maybe you can find answer in it. – [zangw](#) Nov 23 '17 at 3:46

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



19 Here's an example to add on to what others have already written. Suppose you want to make an array of functions, `adderFunctions`, where each function takes a single Number argument and returns the sum of the argument and the function's index in the array. Trying to generate `adderFunctions` with a loop using the `var` keyword won't work the way someone might naïvely expect:

```
// An array of adder functions.
var adderFunctions = [];

for (var i = 0; i < 1000; i++) {
  // We want the function at index i to add the index to its argument.
  adderFunctions[i] = function(x) {
    // What is i bound to here?
    return x + i;
  };
}

var add12 = adderFunctions[12];

// Uh oh. The function is bound to i in the outer scope, which is currently 1000.
console.log(add12(8) === 20); // => false
console.log(add12(8) === 1008); // => true
console.log(i); // => 1000

// It gets worse.
i = -8;
console.log(add12(8) === 0); // => true
```

The process above doesn't generate the desired array of functions because `i`'s scope extends beyond the iteration of the `for` block in which each function was created. Instead, at the end of the loop, the `i` in each function's closure refers to `i`'s value at the end of the loop (1000) for every anonymous function in `adderFunctions`. This isn't what we wanted at all: we now have an array of 1000 different functions in memory with exactly the same behavior. And if we subsequently update the value of `i`, the mutation will affect all the `adderFunctions`.

However, we can try again using the `let` keyword:

```
// Let's try this again.
// NOTE: We're using another ES6 keyword, const, for values that won't
// be reassigned. const and let have similar scoping behavior.
const adderFunctions = [];
```

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



```
// using the "function(x) { ...}" syntax from the previous example
// here would not change the behavior shown.
adderFunctions[i] = x => x + i;
}

const add12 = adderFunctions[12];

// Yay! The behavior is as expected.
console.log(add12(8) === 20); // => true

// i's scope doesn't extend outside the for loop.
console.log(i); // => ReferenceError: i is not defined
```

This time, `i` is rebound on each iteration of the `for` loop. Each function now keeps the value of `i` at the time of the function's creation, and `adderFunctions` behaves as expected.

Now, imagine mixing the two behaviors and you'll probably see why it's not recommended to mix the newer `let` and `const` with the older `var` in the same script. Doing so can result in some spectacularly confusing code.

```
const doubleAdderFunctions = [];

for (var i = 0; i < 1000; i++) {
  const j = i;
  doubleAdderFunctions[i] = x => x + i + j;
}

const add18 = doubleAdderFunctions[9];
const add24 = doubleAdderFunctions[12];

// It's not fun debugging situations like this, especially when the
// code is more complex than in this example.
console.log(add18(24) === 42); // => false
console.log(add24(18) === 42); // => false
console.log(add18(24) === add24(18)); // => false
console.log(add18(24) === 2018); // => false
console.log(add24(18) === 2018); // => false
console.log(add18(24) === 1033); // => true
console.log(add24(18) === 1030); // => true
```

Don't let this happen to you. Use a linter.

**NOTE:** This is a teaching example intended to demonstrate the `var` / `let` behavior in loops and with function closures that would also

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



edited Oct 9 '17 at 22:24

answered Aug 18 '14 at 0:58



abroz

301 2 7

2 @aborz: Also very cool anonymous function syntax in the second example. It's just what I'm used to in C#. I've learned something today. – Barton Feb 20 '15 at 8:59

Correction: Technically, Arrow function syntax described here => [developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/...](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/...) – Barton Mar 16 '15 at 6:58

3 Actually, you don't need `let value = i;`. The `for` statement creates a lexical block. – Toothbrush Oct 22 '15 at 22:38



The difference is in the [scope](#) of the variables declared with each.

16



In practice, there are a number of useful consequences of the difference in scope:



1. `let` variables are only visible in their *nearest enclosing* block ( `{ ... }` ).
2. `let` variables are only usable in lines of code that occur *after* the variable is declared (even though [they are hoisted](#)!).
3. `let` variables may not be redeclared by a subsequent `var` or `let` .
4. Global `let` variables are not added to the global `window` object.
5. `let` variables are *easy to use* with closures (they do not cause [race conditions](#)).

The restrictions imposed by `let` reduce the visibility of the variables and increase the likelihood that unexpected name collisions will be found early. This makes it easier to track and reason about variables, including their [reachability](#) (helping with reclaiming unused memory).

Consequently, `let` variables are less likely to cause problems when used in large programs or when independently-developed frameworks are combined in new and unexpected ways.

`var` may still be useful if you are sure you want the single-binding effect when using a closure in a loop (#5) or for declaring externally-visible global variables in your code (#4). Use of `var` for exports may be supplanted if [export](#) migrates out of transpiler space and into the core language.

## Examples

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



**1. No use outside nearest enclosing block:** This block of code will throw a reference error because the second use of `x` occurs outside of the block where it is declared with `let` :

```
{  
  let x = 1;  
}  
console.log(`x is ${x}`); // ReferenceError during parsing: "x is not defined".
```

In contrast, the same example with `var` works.

**2. No use before declaration:**

This block of code will throw a `ReferenceError` before the code can be run because `x` is used before it is declared:

```
{  
  x = x + 1; // ReferenceError during parsing: "x is not defined".  
  let x;  
  console.log(`x is ${x}`); // Never runs.  
}
```

In contrast, the same example with `var` parses and runs without throwing any exceptions.

**3. No redeclaration:** The following code demonstrates that a variable declared with `let` may not be redeclared later:

```
let x = 1;  
let x = 2; // SyntaxError: Identifier 'x' has already been declared
```

**4. Globals not attached to `window` :**

```
var button = "I cause accidents because my name is too common.";  
let link = "Though my name is common, I am harder to access from other JS files.";  
console.log(link); // OK  
console.log(window.link); // undefined (GOOD!)  
console.log(window.button); // OK
```

**5. Easy use with closures:** Variables declared with `var` do not work well with closures inside loops. Here is a simple loop that outputs the sequence of values that the variable `i` has at different points in time:



Specifically, this outputs:

```
i is 0
i is 1
i is 2
i is 3
i is 4
```

In JavaScript we often use variables at a significantly later time than when they are created. When we demonstrate this by delaying the output with a closure passed to `setTimeout`:

```
for (let i = 0; i < 5; i++) {
  setTimeout(_ => console.log(`i is ${i}`), 125/*ms*/);
}
```

... the output remains unchanged as long as we stick with `let`. In contrast, if we had used `var i` instead:

```
for (var i = 0; i < 5; i++) {
  setTimeout(_ => console.log(`i is ${i}`), 125/*ms*/);
}
```

... the loop unexpectedly outputs "i is 5" five times:

```
i is 5
i is 5
i is 5
i is 5
i is 5
```

edited May 22 '17 at 1:21

answered May 22 '17 at 1:09



[mormegil](#)

1,300 1 16 22

- 5 #5 is not caused by a race condition. By using `var` instead of `let`, the code is equivalent to: `var i = 0; while (i < 5) { doSomethingLater(); i++; }` `i` is outside the closure, and by the time that `doSomethingLater()` is executed, `i` has already been incremented 5 times, hence the output is `i is 5` five times. By using `let`, the variable `i` is within the closure, so each async call gets its own copy of `i` instead of using the 'global' one

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



of the semantics of `for`. A more accurate transformation, though more complicated, is the classical `for (var i = 0; i < 5; i++) { (function(j) { setTimeout(_ => console.log( i is ${{j}} ), 125/*ms*/); })(i); } which introduces a "function-activation record" to save each value of i with the name of j inside the function. – mormegil Jul 25 '17 at 7:13`

May the following two functions show the difference:

14

```
function varTest() {
  var x = 31;
  if (true) {
    var x = 71; // Same variable!
    console.log(x); // 71
  }
  console.log(x); // 71
}

function letTest() {
  let x = 31;
  if (true) {
    let x = 71; // Different variable
    console.log(x); // 71
  }
  console.log(x); // 31
}
```

edited Nov 26 '16 at 16:18



Peter Mortensen

24.8k 20 89 118

answered Dec 17 '15 at 3:22



Abdennour TOUMI

47.9k 23 168 179

`let` is interesting, because it allows us to do something like this:

13

```
((() => {
  var count = 0;

  for (let i = 0; i < 2; ++i) {
    for (let i = 0; i < 2; ++i) {
      for (let i = 0; i < 2; ++i) {
```

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



```
    }
  })();
```

Which results in counting [0, 7].

Whereas

```
((() => {
  var count = 0;

  for (var i = 0; i < 2; ++i) {
    for (var i = 0; i < 2; ++i) {
      for (var i = 0; i < 2; ++i) {
        console.log(count++);
      }
    }
  }
})());
```

Only counts [0, 1].

edited Nov 26 '16 at 16:34



Peter Mortensen

24.8k 20 89 118

answered Jul 8 '16 at 0:21



Dmitry

3,914 3 26 39

2 this is the first time i've ever seen anyone act like variable shadowing was desirable. no, the purpose of let is not to enable shadowing – [John Haugeland](#) Nov 24 '16 at 0:37

purpose? it's a construct, you can use it however you please, one of the interesting ways is like this. – [Dmitry](#) Nov 24 '16 at 0:39

## Function VS block scope:

13

The main difference between `var` and `let` is that variables declared with `var` are **function scoped**. Whereas functions declared with `let` are **block scoped**. For example:

```
function testVar () {
```

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



```
    console.log(foo);
  }

  testVar();
  // Logs 'foo'

  function testLet () {
    if(true) {
      let bar = 'bar';
    }

    console.log(bar);
  }

  testLet();
  // reference error
  // bar is scoped to the block of the if statement
```

### variables with var :

When the first function `testVar` gets called the variable `foo`, declared with `var`, is still accessible outside the `if` statement. This variable `foo` would be available **everywhere** within the scope of the `testVar` **function**.

### variables with let :

When the second function `testLet` gets called the variable `bar`, declared with `let`, is only accessible inside the `if` statement. Because variables declared with `let` are **block scoped** (where a block is the code between curly brackets e.g `if{}`, `for{}`, `function{}`).

### let variables don't get hoisted:

Another difference between `var` and `let` is variables with declared with `let` **don't get hoisted**. An example is the best way to illustrate this behavior:

variables with `let` **don't** get hoisted:

```
console.log(letVar);

let letVar = 10;
// referenceError, the variable doesn't get hoisted
```

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



```
console.log(varVar);

var varVar = 10;
// Logs undefined, the variable gets hoisted
```

## Global `let` doesn't get attached to `window` :

A variable declared with `let` in the global scope (which is code that is not in a function) doesn't get added as a property on the global `window` object. For example (this code is in global scope):

```
var bar = 5;
let foo = 10;

console.log(bar); // Logs 5
console.log(foo); // Logs 10

console.log(window.bar);
// Logs 5, variable added to window object

console.log(window.foo);
// Logs undefined, variable not added to window object
```

### When should `let` be used over `var` ?

Use `let` over `var` whenever you can because it is simply scoped more specific. This reduces potential naming conflicts which can occur when dealing with a large number of variables. `var` can be used when you want a global variable explicitly to be on the `window` object (always consider carefully if this is really necessary).

edited Sep 10 '18 at 7:39



Suraj Rao

26.2k 8 73 84

answered Sep 9 '18 at 13:08



Willem van der Veen



10.6k 5 68 66



It also appears that, at least in Visual Studio 2015, TypeScript 1.5, "var" allows multiple declarations of the same variable name in a block and "let" doesn't

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



```
var x = 1;  
var x = 2;
```

This will:

```
let x = 1;  
let x = 2;
```

edited Nov 28 '16 at 9:31



John Slegers


32.9k 14 165 141

answered Aug 11 '15 at 0:35



RDoc

117 2 7




var is global scope (hoist-able) variable.

9

let and const is block scope.



test.js



```
{  
  let l = 'let';  
  const c = 'const';  
  var v = 'var';  
  v2 = 'var 2';  
}  
  
console.log(v, this.v);  
console.log(v2, this.v2);  
console.log(l); // ReferenceError: l is not defined  
console.log(c); // ReferenceError: c is not defined
```

Run code snippet

[Expand snippet](#)

answered Oct 28 '17 at 12:42

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



## When Using `let`

8

The `let` keyword attaches the variable declaration to the scope of whatever block (commonly a `{ .. }` pair) it's contained in. In other words, `let` implicitly hijacks any block's scope for its variable declaration.

`let` variables cannot be accessed in the `window` object because they cannot be globally accessed.



```
function a(){
  // this is the Max Scope for let variable
  let x = 12;
}
console.log(x);
}
a(); // Uncaught ReferenceError: x is not defined
```

## When Using `var`

`var` and variables in ES5 has scopes in functions meaning the variables are valid within the function and not outside the function itself.

`var` variables can be accessed in the `window` object because they cannot be globally accessed.

```
function a(){ // this is the Max Scope for var variable
{
  var x = 12;
}
console.log(x);
}
a(); // 12
```

## If you want to know more continue reading below

one of the most famous interview questions on scope also can suffice the exact use of `let` and `var` as below;

## When using `let`

```
for (let i = 0; i < 10 ; i++) {
  setTimeout(
```

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



```
    100 * i);
}
```

This is because when using `let`, for every loop iteration the variable is scoped and has its own copy.

### When using `var`

```
for (var i = 0; i < 10 ; i++) {
  setTimeout(
    function a() {
      console.log(i); //print 10 times 10
    },
    100 * i);
}
```

This is because when using `var`, for every loop iteration the variable is scoped and has shared copy.

edited May 22 '18 at 13:22

answered May 22 '18 at 13:12



Ankur Soni

3,834 3 23 49



7

If I read the specs right then `let` **thankfully** can also be leveraged to avoid [self invoking functions](#) used to simulate private only members - *a popular design pattern that decreases code readability, complicates debugging, that adds no real code protection or other benefit - except maybe satisfying someone's desire for semantics, so stop using it. /rant*



```
var SomeConstructor;

{
  let privateScope = {};

  SomeConstructor = function SomeConstructor () {
    this.someProperty = "foo";
    privateScope.hiddenProperty = "bar";
  }

  SomeConstructor.prototype.showPublic = function () {
    console.log(this.someProperty); // foo
  }
}
```

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





```
    }  
  }  
  
  var myInstance = new SomeConstructor();  
  
  myInstance.showPublic();  
  myInstance.showPrivate();  
  
  console.log(privateScope.hiddenProperty); // error
```

See '[Emulating private interfaces](#)'

edited Jan 12 '19 at 5:29

answered Oct 14 '16 at 5:01



[Daniel Sokolowski](#)

9,185 2 52 46



Some hacks with let :

4

1.

```
let statistics = [16, 170, 10];  
let [age, height, grade] = statistics;  
  
console.log(height)
```

2.

```
let x = 120,  
    y = 12;  
[x, y] = [y, x];  
console.log(`x: ${x} y: ${y}`);
```

3.

```
let node = {  
  type: "Identifier",
```

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



```
let { type, name, value } = node;

console.log(type);    // "Identifier"
console.log(name);    // "foo"
console.log(value);   // undefined

let node = {
  type: "Identifier"
};

let { type: localType, name: localName = "bar" } = node;

console.log(localType);    // "Identifier"
console.log(localName);    // "bar"
```

## Getter and setter with `let` :

```
let jar = {
  numberOfCookies: 10,
  get cookies() {
    return this.numberOfCookies;
  },
  set cookies(value) {
    this.numberOfCookies = value;
  }
};

console.log(jar.cookies)
jar.cookies = 7;

console.log(jar.cookies)
```

edited Nov 26 '16 at 16:44



Peter Mortensen

24.8k 20 89 118

answered Jul 21 '16 at 17:42



zloctb

6,938 3 53 65

please what do this mean `let { type, name, value } = node;` ? you create a new object with 3 properties type/name/value and initialise them with the properties values from node ? – [Alainlb](#) Jun 15 '17 at 7:55

In example 3 you are re-declaring node which cause exception. These all examples also work perfectly with `var` too. – [Rehan Haider](#) Jan 9 '19 at 10:57



let vs var. It's all about **scope**.

4



**var variables are global** and can be accessed basically everywhere, while **let variables are not global** and only exist until a closing parenthesis kills them.



See my example below, and note how the lion (let) variable acts differently in the two console.logs; it becomes out of scope in the 2nd console.log.

```
var cat = "cat";
let dog = "dog";

var animals = () => {
  var giraffe = "giraffe";
  let lion = "lion";

  console.log(cat); //will print 'cat'.
  console.log(dog); //will print 'dog', because dog was declared outside this
function (like var cat).

  console.log(giraffe); //will print 'giraffe'.
  console.log(lion); //will print 'lion', as lion is within scope.
}

console.log(giraffe); //will print 'giraffe', as giraffe is a global variable (var).
console.log(lion); //will print UNDEFINED, as lion is a 'let' variable and is now out of
scope.
```

answered Apr 18 '19 at 0:49



daCoda

1,760 3 19 23



let is a part of es6. These functions will explain the difference in easy way.

3



```
function varTest() {
  var x = 1;
  if (true) {
    var x = 2; // same variable!
    console.log(x); // 2
  }
}
```

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



```
function letTest() {  
  let x = 1;  
  if (true) {  
    let x = 2; // different variable  
    console.log(x); // 2  
  }  
  console.log(x); // 1  
}
```

answered Dec 17 '17 at 10:47



vipul jain

74 4

2

I think the terms and most of the examples are a bit overwhelming, The main issue i had personally with the difference is understanding what a "Block" is. At some point i realized, a block would be any curly brackets except for `if` statement. an opening bracket `{` of a function or loop will define a new block, anything defined with `let` within it, will not be available after the closing bracket `}` of the same thing (function or loop); With that in mind, it was easier to understand:

```
let msg = "Hello World";  
  
function doWork() { // msg will be available since it was defined above this opening bracket!  
  let friends = 0;  
  console.log(msg);  
  
  // with VAR though:  
  for (var iCount2 = 0; iCount2 < 5; iCount2++) {} // iCount2 will be available after this closing bracket!  
  console.log(iCount2);  
  
  for (let iCount1 = 0; iCount1 < 5; iCount1++) {} // iCount1 will not be available behind this closing bracket, it will return undefined  
  console.log(iCount1);  
  
} // friends will no be available after this closing bracket!  
doWork();  
console.log(friends);
```

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



answered Apr 28 '19 at 2:21



Dementic

13.1k 15 58 87

The below shows how 'let' and 'var' are different in the scope:

2

```
let gfoo = 123;
if (true) {
  let gfoo = 456;
}
console.log(gfoo); // 123

var hfoo = 123;
if (true) {
  var hfoo = 456;
}
console.log(hfoo); // 456
```

The `gfoo`, defined by `let` initially is in the **global scope**, and when we declare `gfoo` again inside the `if` clause its **scope changed** and when a new value is assigned to the variable inside that scope it **does not affect** the global scope.

Whereas `hfoo`, defined by `var` is initially in the **global scope**, but again when we declare it inside the `if` clause, it considers the global scope `hfoo`, although `var` has been used again to declare it. And when we re-assign its value we see that the global scope `hfoo` is also affected. This is the primary difference.

answered Sep 7 '19 at 11:25



Piklu Dey

150 9

Previously there were only two scopes in JavaScript, i.e. functional and global. With ' `let` ' keyword JavaScript has now introduced block-level variables.

1

To have a complete understanding of the 'let' keyword, [ES6: 'let' keyword to declare variable in JavaScript](#) will help.

edited Nov 26 '16 at 16:27

answered Feb 8 '16 at 15:03

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



Now I think there is better scoping of variables to a block of statements using `let` :

```
1 function printnums()
  {
    // i is not accessible here
    for(let i = 0; i < 10; i++)
    {
      console.log(i);
    }
    // i is not accessible here

    // j is accessible here
    for(var j = 0; j < 10; j++)
    {
      console.log(j);
    }
    // j is accessible here
  }
}
```

I think people will start using `let` here after so that they will have similar scoping in JavaScript like other languages, Java, C#, etc.

People with not a clear understanding about scoping in JavaScript used to make the mistake earlier.

Hoisting is not supported using `let` .

With this approach errors present in JavaScript are getting removed.

Refer to [ES6 In Depth: let and const](#) to understand it better.

edited Nov 26 '16 at 16:33



Peter Mortensen

24.8k 20 89 118

answered Jul 1 '16 at 8:22



swaraj patil

151 1 5

For in depth understanding on it refer link - [davidwalsh.name/for-and-against-let](#) – swaraj patil Jul 1 '16 at 8:27

1

`const` is a signal that the identifier won't be reassigned.



`let` is a signal that the variable may be reassigned, such as a counter in a loop, or a value swap in an algorithm. It also signals that the variable will be used only in the block it's defined in, which is not always the entire containing function.

`var` is now the weakest signal available when you define a variable in JavaScript. The variable may or may not be reassigned, and the variable may or may not be used for an entire function, or just for the purpose of a block or loop.

<https://medium.com/javascript-scene/javascript-es6-var-let-or-const-ba58b8dcde75#esmkpbg9b>

edited Apr 30 '17 at 17:11



FelixSFD

4,912 9 35 99

answered Dec 27 '16 at 9:44



anandharshan

3,143 4 21 23

As mentioned above:

1

The difference is scoping. `var` is scoped to the nearest **function block** and `let` is scoped to the **nearest enclosing block**, which can be smaller than a function block. Both are global if outside any block. Lets see an example:

### Example1:



In my both examples I have a function `myfunc`. `myfunc` contains a variable `myvar` equals to 10. In my first example I check if `myvar` equals to 10 ( `myvar==10` ). If yes, I again declare a variable `myvar` (now I have two `myvar` variables) using `var` keyword and assign it a new value (20). In next line I print its value on my console. After the conditional block I again print the value of `myvar` on my console. If you look at the output of `myfunc`, `myvar` has value equals to 20.



**Example 1:**

```
function myfunc() {
  var myvar=10;

  if(myvar==10){
    var myvar=20;
    console.log("conditional block myvar=", myvar);
  }

  console.log("function myvar=", myvar);
}
```

**Output**

```
conditional block myvar= 20
function myvar= 20
```

**Example 2:**

```
function myfunc() {
  var myvar=10;

  if(myvar==10){
    let myvar=20;
    console.log("conditional block myvar=", myvar);
  }

  console.log("function myvar=", myvar);
}
```

**Output**

```
conditional block myvar= 20
function myvar= 10
```

**Example2:** In my second example instead of using `var` keyword in my conditional block I declare `myvar` using `let` keyword . Now when I call `myfunc` I get two different outputs: `myvar=20` and `myvar=10` .

So the difference is very simple i.e its scope.

edited Aug 13 '18 at 14:02

answered Aug 7 '18 at 10:25



N Randhawa

4,669 2 29 41

- 3 Please don't post pictures of code, it's considered bad practice on SO as it will not be searchable for future users (as well as accessibility concerns). As well, this answer adds nothing that other answers haven't already addressed. – inostia Aug 24 '18 at 17:29



1

I want to link these keywords to the Execution Context, because the Execution Context is important in all of this. The Execution Context has two phases: a Creation Phase and Execution Phase. In addition, each Execution Context has a Variable Environment and Outer Environment (its Lexical Environment).



During the Creation Phase of an Execution Context, `var`, `let` and `const` will still store its variable in memory with an undefined value in the Variable Environment of the given Execution Context. The difference is in the Execution Phase. If you use reference a variable defined with `var` before it is assigned a value, it will just be undefined. No exception will be raised.



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





Creation Phase of the Execution Context, but the Engine will not allow you to use it:

```
function a(){  
  b;  
  let b;  
}  
a();  
> Uncaught ReferenceError: b is not defined
```

With a variable defined with var, if the Engine cannot find the variable in the current Execution Context's Variable Environment, then it will go up the scope chain (the Outer Environment) and check the Outer Environment's Variable Environment for the variable. If it cannot find it there, it will continue searching the Scope Chain. This is not the case with let and const.

The second feature of let is it introduces block scope. Blocks are defined by curly braces. Examples include function blocks, if blocks, for blocks, etc. When you declare a variable with let inside of a block, the variable is only available inside of the block. In fact, each time the block is run, such as within a for loop, it will create a new variable in memory.

ES6 also introduces the const keyword for declaring variables. const is also block scoped. The difference between let and const is that const variables need to be declared using an initializer, or it will generate an error.

And, finally, when it comes to the Execution Context, variables defined with var will be attached to the 'this' object. In the global Execution Context, that will be the window object in browsers. This is not the case for let or const.

answered Feb 13 '19 at 16:07



Donato

4,115 4 21 52



1



```
1  const name = 'Max';
2  let age = 33;
3  var hasHobbies = true;
```

● Attempting to override 'name' which is a constant.

```
● 7  name = 'Maximilian';
8  age = 34;
9  hasHobbies = false;
10
11  const summarizeUser = (userName, userAge, userHasHobby) => {
12    return (
13      'Name is ' +
14      userName +
15      ', age is ' +
16      userAge +
17      ' and the user has hobbies: ' +
18      userHasHobby
19    );
20  }
21
22  console.log(summarizeUser(name, age, hasHobbies));
```

Take a look at this image, I created one very simple example for demonstration of `const` and `let` variables. As you can see, when you try to change `const` variable, you will get the error (**Attempting to override 'name' which is constant**), but take a look at `let` variable...

First we declare `let age = 33`, and later assign some other value `age = 34;`, which is ok, we don't have any errors when we try to change `let` variable

edited Feb 16 '19 at 17:26

answered Feb 16 '19 at 17:17



[Mile Mijatović](#)

1,587 14 30

1 2 next

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



answer activity.

