What is the difference between "let" and "var"?

Asked 15 years, 8 months ago Modified 24 days ago

Viewed 2.4m times

6340

ECMAScript 6 introduced the `let` declaration keyword.

I've heard 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 instead of `var`?

javascript scope ecmascript-6 var let

Share Follow

edited Nov 26 at 21:37

isherwood

60.9k ● 16 ● 120 ● 168

asked Apr 17, 2009 at 20:09

TM.

111k ● 34 ● 124 ● 128

38 Answers

Sorted by: Highest score (default)

1 2 Next

8293

Scoping rules

The 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); // Foo Bar

  {
    var moo = "Mooo"
    let baz = "Bazz";
    console.log(moo, baz); // Mooo Bazz
  }

  console.log(moo); // Mooo
  console.log(baz); // ReferenceError
}

run();
```

Run code snippet   Expand snippet

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 Stack Overflow 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]();
}
```

Run code snippet   Expand snippet

`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 values from the loops but that was also hairy.

Hoisting

Variables declared with `var` keyword are hoisted and *initialized* which means they are accessible in their enclosing scope even before they are declared, however their value is `undefined` before the declaration statement is reached:

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

checkHoisting();
```

Run code snippet   Expand snippet

`let` variables are hoisted but *not initialized* until their definition is evaluated. Accessing them before the initialization results in a `ReferenceError`. The variable is said to be in the temporal dead zone from the start of the block until the declaration statement is processed.

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

checkHoisting();
```

Run code snippet   Expand snippet

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 but not part of

console.log(window.foo); // Foo
console.log(window.bar); // undefined
```

Run code snippet   Expand snippet

Redeclaration

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, 'foo1' is replaced
```

```
let bar = "bar1";
let bar = "bar2"; // SyntaxError: Identifier 'bar' h
```

Run code snippet   Expand snippet

Share Follow

edited Sep 18, 2023 at 16:43

community wiki
39 revs, 29 users 27%
ThinkingStiff

77 Remember you can create block whenever you want. function() { code;{ let inBlock = 5; } code; }; – average Joe Dec 14, 2012 at 10:14

261 So is the purpose of let statements only to free up memory when not needed in a certain block? – NoBugs Jun 7, 2013 at 5:18

307 @NoBugs, Yes, and it is encouraged that variables are existent only where they are needed. – batman Jun 7, 2013 at 15:02

75 `let` block expression `let (variable declaration) statement` is non-standard and will be removed in future, bugzilla.mozilla.org/show_bug.cgi?id=1023609. – Gajus Dec 17, 2014 at 14:51

11 Although a let enclosing block is deprecated you can do the same thing by creating an explicit block with braces. { let bar = foo; let foo = bar; `code` ; `more code` ; console.log(bar + foo); } and simply have your lets at the top of your block of code wrapped in braces. – Eric Bishard May 7, 2015 at 14:12

841

`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.

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

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3
</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
</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

Share Follow

edited Oct 29, 2019 at 8:47
Nash Bridges
2,370 ● 14 ● 19

answered May 27, 2015 at 10:16

Gurpreet Singh
21.2k ● 5 ● 46 ● 61

81 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, 2015 at 12:49

12 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, 2015 at 13:29

13 Looks like your answer is the correct behavior: developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/… – Jim Hunziker Oct 22, 2015 at 13:32

17 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 immediatelly invoked function expression for locally scoping `i` at each iteration. – Adrian Moisa Feb 21, 2016 at 8:12

26 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, 2016 at 23:10

336

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

- 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 secon
    // l IS NOT known here

    for( var i = 0; i < arr.length; i++ ) {
        // i IS known here, and has a value
        // j IS NOT known here
        // k IS known here, but has a value only the s
        // 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 secon
    // 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 s
        // 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 secon
    // 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
```

Here, we can see that our variable `j` is only known in the first for loop, but not before and after. Yet, our variable `i` is known in the entire function.

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**.

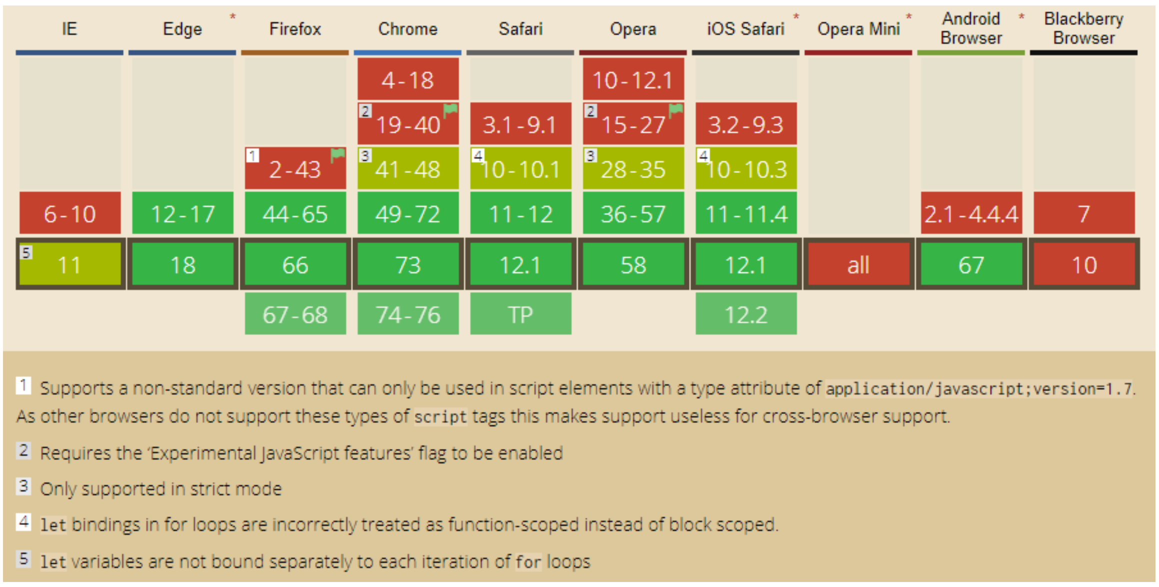
Today, however, that is definitely not the case. In fact, we need actually to ask ourselves whether it's safe to use the `let` statement. The answer to that question depends on your environment:

- 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				
			[2] 19-40	3.1-9.1	[2] 15-27	3.2-9.3			
		[1] 2-43	[3] 41-48	[4] 10-10.1	[3] 28-35	[4] 10-10.3			
6-10	12-17	44-65	49-72	11-12	36-57	11-11.4		2.1-4.4.4	7
[5] 11	18	66	73	12.1	58	12.1	all	67	10
		67-68	74-76	TP		12.2			

[1] 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.

[2] Requires the 'Experimental JavaScript features' flag to be enabled

[3] Only supported in strict mode

[4] `let` bindings in for loops are incorrectly treated as function-scoped instead of block scoped.

[5] `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 moved to the top of the scope.

(**) Block scoped variables are not hoisted

Share Follow

edited Dec 28, 2022 at 17:05

answered Feb 23, 2016 at 18:35

John Slegers
47k ● 23 ● 203 ● 172

23 regarding answer v4: `i` 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, 2016 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, 2018 at 23:37

2 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, 2018 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, 2018 at 18:17

2 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, 2019 at 12:42

187

Here's an explanation of the `let` keyword with some examples.

> `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.

Share Follow

edited Jun 24, 2019 at 2:52
Jack Bashford
44k ● 11 ● 55 ● 82

answered Apr 17, 2009 at 20:11

Ben S
69.3k ● 31 ● 173 ● 214

7 The key bit of text from the linked document seems to be, "let works very much like var. The main difference is that the scope of a var variable is the entire enclosing function". – Michael Burr Apr 17, 2009 at 20:25

60 @olliej, actually Mozilla is just ahead of the game. See page 19 of ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf – Tyler Crompton Jun 18, 2012 at 20:16

1 @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, 2012 at 18:49

11 IE11 added support for `let` msdn.microsoft.com/en-us/library/ie/dn342892%28v=vs.85%29.aspx – eloyesp Dec 24, 2013 at 12:59

2 Now `let` support all latest browser today except Opera, Blackberry & QQ Browsers. – Shapon Pal Jan 8, 2019 at 5:11

156

`let`

Block scope

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

Inside a block

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

```
{
  var globalVariable = 42;
  let blockScopedVariable = 43;
  console.log(globalVariable); // 42
  console.log(blockScopedVariable); // 43
}

console.log(globalVariable); // 42
console.log(blockScopedVariable); // ReferenceError: b
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
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 throws an error.

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

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

`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
```

If you want to have an immutable object, you should use `Object.freeze()`.

```
const obj = Object.freeze({a: 40});
obj.a = 42;
console.log(obj.a); // 40
console.log(obj.b); // undefined
```

Initializer is required

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

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

Share Follow

edited Mar 21, 2022 at 15:02

Abdul Rehman
1,794 ● 3 ● 23 ● 38

answered Nov 23, 2016 at 22:52

Michał Perłakowski
92.2k ● 30 ● 163 ● 185

147

The accepted answer is missing a point:

```
{
  let a = 123;
};

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

Share Follow

edited Jul 14, 2016 at 14:13

William
413 ● 9 ● 21

answered Jun 2, 2015 at 20:59

Lcf.vs
1,842 ● 1 ● 12 ● 15

21 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, 2015 at 6:55

45 @stimpy77 It explicitly states "let is scoped to the nearest enclosing block"; does every way that manifests need to be included? – Dave Newton Mar 31, 2016 at 21:32

9 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, 2016 at 21:38

6 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, 2017 at 14:37

112

In most basic terms,

```
for (let i = 0; i < 5; i++) {
  // i accessible ✔️
}
// i not accessible ❌
```

```
for (var i = 0; i < 5; i++) {
  // i accessible ✔️
}
// i accessible ✔️
```

⚡ Sandbox to play around ↓

Edit in CodeSandbox

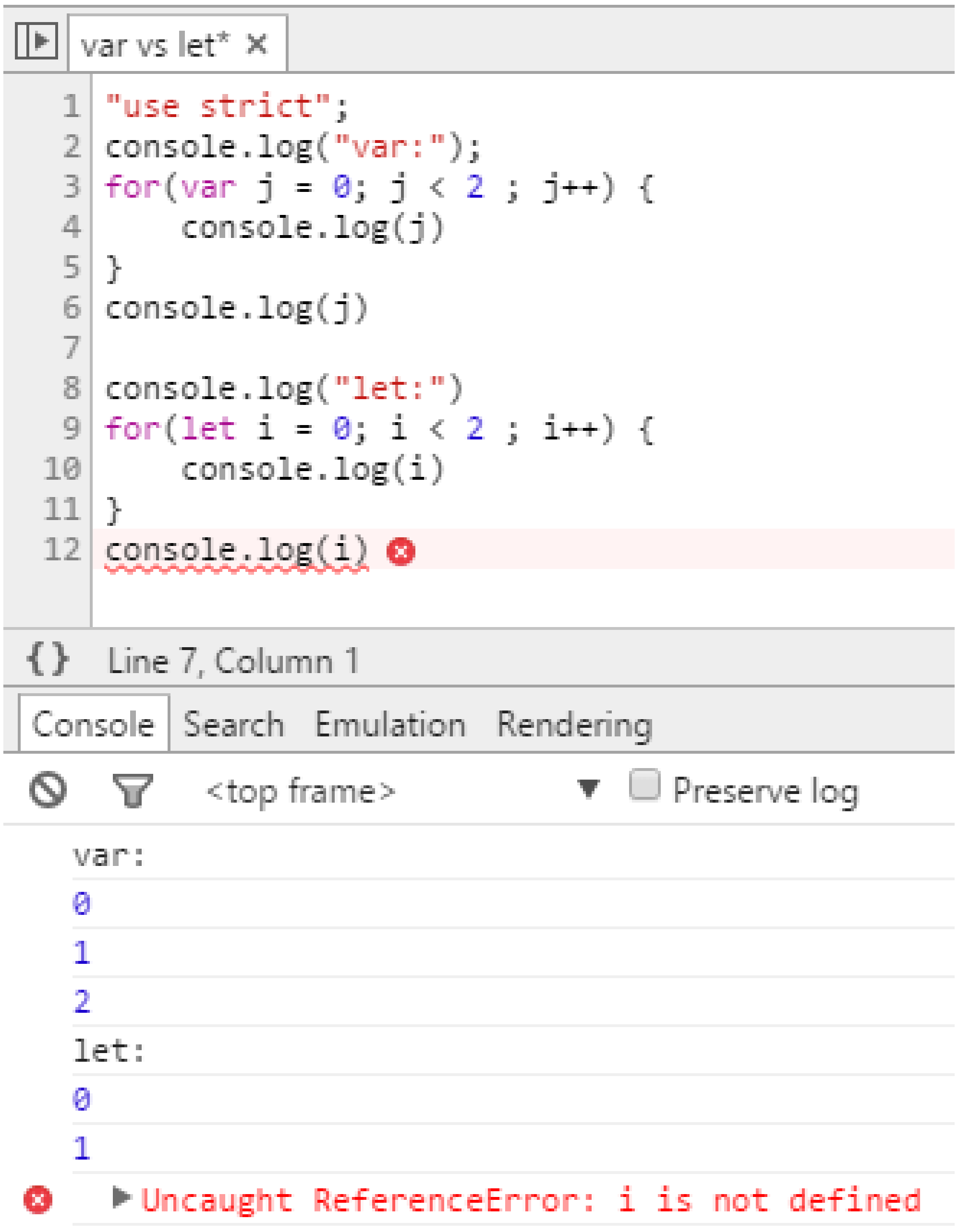
Share Follow

answered May 11, 2020 at 17:04

Hasan Sefa Ozalp
7,178 ● 5 ● 36 ● 47

71

Here is an example of the difference between the two:


var vs let*

```
"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)
```

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
```

As you can see, the `var j` variable still has a value outside the for loop scope (Block Scope), but the `let i` variable is undefined outside of the for loop scope.

```
"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

Share   Follow   edited Feb 6, 2023 at 10:02

answered Mar 6, 2015 at 10:41

vlio20
9,295 ● 18 ● 101 ● 186

70

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):

> **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
  }
  console.log(x);  // 1
}`
```

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

Share Follow

edited Sep 18, 2023 at 16:24

VLAZ
28.8k ● 9 ● 62 ● 82

answered Mar 22, 2017 at 14:39

Alireza
105k ● 27 ● 277 ● 173

I don't know if that last example is accurate. Because by calling it not from a function but a direct command line its still considered part of the same function. So, if you called it from

outside of a function, it shouldn't behave in the same way.
– ACopeLan Aug 28, 2020 at 18:20

66

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.

Share Follow

edited Jun 24, 2019 at 2:53
Jack Bashford
44k ● 11 ● 55 ● 82

answered Apr 17, 2009 at 21:38
olliej
36.7k ● 9 ● 60 ● 55

14 It's also worth noting that 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, 2012 at 15:09

8 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, 2012 at 17:38

4 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, 2015 at 14:01

1 This is one of the most important distinctions between let and var and it's not in the accepted answer haha. Especially considering the numerous bugs that can occur thanks to hoisting and scoping. I feel like there aren't many differences between let and var if you don't mention hoisting. – Jay Jun 21, 2015 at 16:12

4 @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/… – GitaarLAB May 21, 2016 at 4:15

35

- ~~**Variable Not Hoisting**~~

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

```
{
   console.log(cc); // undefined. Caused by hoisti
   var cc = 23;
}

{
   console.log(bb); // ReferenceError: bb is not d
```

```
    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 ga
    let hugeData = { .. };
    process(hugeData);
}

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

- `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 (previosly to evaluating the increment expression), more details are here.

Share Follow

edited May 23, 2017 at 12:34

Community Bot
1 ● 1

answered Jan 17, 2016 at 15:11

zangw
48.1k ● 23 ● 203 ● 241

6 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, 2016 at 15:42

35

The difference is in the scope of the variables declared with each.

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

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 pa
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 i
    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
```

4. Globals not attached to `window`:

```
var button = "I cause accidents because my name is too
let link = "Though my name is common, I am harder to a
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:

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

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
```

Share Follow

edited May 22, 2017 at 1:21

answered May 22, 2017 at 1:09

mormegil
1,880 ● 1 ● 20 ● 22

11 #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 that's created with `var`. – Daniel T. Jun 2, 2017 at 1:12

@DanielT.: I don't think the transformation of lifting the variable definition out of the loop initializer explains anything. That is simply the normal definition 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, 2017 at 7:13

28

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
  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 scop
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 v
// be reassigned. const and let have similar scoping b
const adderFunctions = [];

for (let i = 0; i < 1000; i++) {
  // NOTE: We're using the newer arrow function syntax
  // using the "function(x) { ..." syntax from the pre
  // 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, image 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 is 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, especi
// 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 be easy to understand. This would be a terrible way to add numbers. But the general technique of capturing data in anonymous function closures might be encountered in the real world in other contexts. YMMV.

Share Follow

edited Oct 9, 2017 at 22:24

answered Aug 18, 2014 at 0:58

abroz
391 ● 3 ● 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, 2015 at 8:59

Correction: Technically, Arrow function syntax described here => [developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/…]() – Barton Mar 16, 2015 at 6:58

3 Actually, you don't need `let value = i;`. The `for` statement creates a lexical block. – Toothbrush Oct 22, 2015 at 22:38

28

The explanation is taken from the article I wrote at [Medium]():

> Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope by the parser which reads the source code into an intermediate representation before the actual code execution starts by the JavaScript interpreter. So, it actually doesn’t matter where variables or functions are declared, they will be moved to the top of their scope regardless of whether their scope is global or local. This means that

```
console.log (hi);
var hi = "say hi";
```

is actually interpreted to

```
var hi = undefined;
console.log (hi);
hi = "say hi";
```

So, as we saw just now, `var` variables are being hoisted to the top of their scope and are being initialized with the value of undefined which means that we can actually assign their value before actually declaring them in the code like so:

```
hi = “say hi”
console.log (hi); // say hi
var hi;
```

Regarding function declarations, we can invoke them before actually declaring them like so:

```
sayHi(); // Hi

function sayHi() {
   console.log('Hi');
};
```

Function expressions, on the other hand, are not hoisted, so we’ll get the following error:

```
sayHi(); //Output: "TypeError: sayHi is not a func

var sayHi = function() {
  console.log('Hi');
};
```

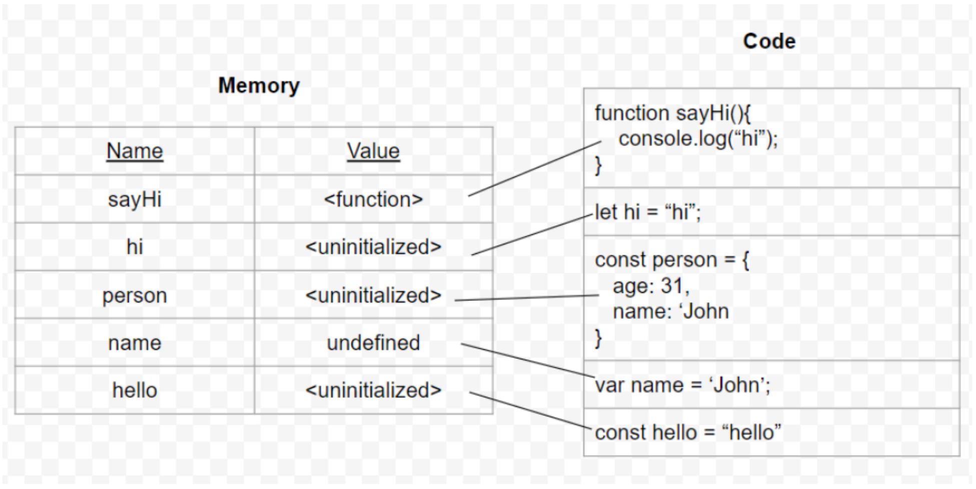
ES6 introduced JavaScript developers the `let` and `const` keywords. While `let` and `const` are block-scoped and not function scoped as `var` it shouldn't make a difference while discussing their hoisting behavior. We'll start from the end, JavaScript hoists `let` and `const`.

```
console.log(hi); // Output: Cannot access 'hi' bef
let hi = 'Hi';
```

As we can see above, `let` doesn't allow us to use undeclared variables, hence the interpreter explicitly output a reference error indicating that the `hi` variable cannot be accessed before initialization. The same error will occur if we change the above `let` to `const`

```
console.log(hi); // Output: Cannot access 'hi' bef
const hi = 'Hi';
```

So, bottom line, the JavaScript parser searches for variable declarations and functions and hoists them to the top of their scope before code execution and assign values to them in the memory so in case the interpreter will encounter

> them while executing the code he will recognize them and will be able to execute the code with their assigned values. Variables declared with `let` or `const` remain uninitialized at the beginning of execution while that variables declared with `var` are being initialized with a value of `undefined`.
>
> I added this visual illustration to better help understanding of how are the hoisted variables and function are being saved in the memory
>
> 

Memory

Name	Value
sayHi	<function>
hi	<uninitialized>
person	<uninitialized>
name	undefined
hello	<uninitialized>

Code

```
function sayHi(){
   console.log(“hi”);
}
let hi = “hi”;
const person = {
   age: 31,
   name: ‘John
}
var name = ‘John’;
const hello = “hello”
```

Share Follow

edited Sep 23, 2023 at 7:21

answered Jan 12, 2022 at 18:20

Ran Turner
17.8k ● 8 ● 56 ● 59

Proper attribution for quoted content is required on Stack Overflow. This includes explicit disclosure of affiliation and clearly displaying when content is copied from another location... even if you are the author. – TylerH Feb 11, 2022 at 14:28

19

Function VS block scope:

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 () {
  if(true) {
    var foo = 'foo';
  }

  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
```

variables with `var` **do** get hoisted:

```
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).

Share Follow

edited Sep 10, 2018 at 7:39
Suraj Rao
29.6k ● 11 ● 95 ● 104

answered Sep 9, 2018 at 13:08
Willem van der Veen
36.4k ● 18 ● 204 ● 176

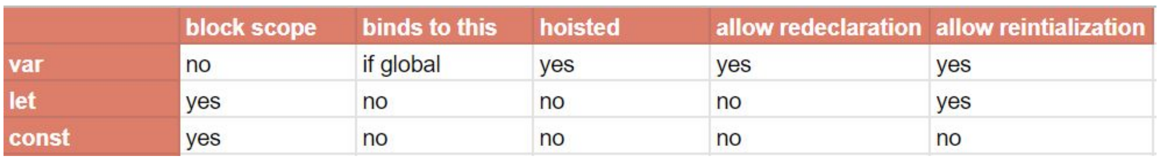
"`let` *variables don't get hoisted:*" [Are variables declared with let or const hoisted?]() - `let` *declarations* are hoisted, not the *initialisation*. That is the entire reason the [Temporal Dead Zone]() exists - because the runtime can know about the declaration before reaching it. Thus know it's an error because the variable was referenced before it was declared. Because the declaration was hoisted. – VLAZ Sep 18, 2023 at 12:15

19

ES6 introduced two new keyword(**let** and **const**) alternate to **var**.

When you need a block level deceleration you can go with let and const instead of var.

The below table summarize the difference between var, let and const



	block scope	binds to this	hoisted	allow redeclaration	allow reintialization
var	no	if global	yes	yes	yes
let	yes	no	no	no	yes
const	yes	no	no	no	no

Share Follow

edited Apr 13, 2021 at 18:50

TylerH
21.2k ● 76 ● 79 ● 110

answered Jan 26, 2020 at 11:39

Srikrushna
4,865 ● 2 ● 43 ● 49

11 The hoisted column is incorrect. They all hoist variable. The difference with `var` is that they hoist but do not initialize to the `undefined` value. If they did not hoist, they would not mask variables of the same name in enclosing blocks: stackoverflow.com/q/63337235/2326961 – Géry Ogam Aug 11, 2020 at 9:43

also `var x=2;` is different than `x=2;` – a55 May 29 at 2:33

17

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

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

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

```
.as-console-wrapper { max-height: 100% !important; }
```

Run code snippet   Expand snippet

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

Run code snippet   Expand snippet

Only counts [0, 1].

Share Follow

edited Sep 18, 2023 at 12:12
VLAZ
28.8k ● 9 ● 62 ● 82

answered Jul 8, 2016 at 0:21
Dmytro
5,213 ● 4 ● 44 ● 55

yes, it adds much more confusion than necessary and where there should be none. – Bekim Bacaj Oct 4, 2021 at 19:27

@Bekim Bacaj This is a contrived example illustrating a distinction between let and var. That is, at the end of the loop, the variable declared by let goes out of scope, whereas var remains. It is up to the programmer to decide which constructs they opt to incorporate into their code, based on their intent, and prior experience. The intent of this example is not to cause confusion, but to give the reader a starting point to play with the let construct in creative ways to familiarize themselves with it. – Dmytro Oct 6, 2021 at 2:29

16

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.

This won't generate a compile error:

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

This will:

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

Share Follow

edited Nov 28, 2016 at 9:31

John Slegers
47k ● 23 ● 203 ● 172

answered Aug 11, 2015 at 0:35

RDoc
215 ● 3 ● 8

15

```
var   --> Function scope
let   --> Block scope
const --> Block scope
```

var

In this code sample, variable `i` is declared using `var`. Therefore, it has a *function scope*. It means you can access `i` from only inside the `function x`. You can't read it from outside the `function x`

```
function x(){
  var i = 100;
  console.log(i); // 100
}

console.log(i); // Error. You can't do this

x();
```

Run code snippet   Expand snippet

In this sample, you can see `i` is declared inside a `if` block. But it's declared using `var`. Therefore, it gets function scope. It means still you can access variable `i` inside `function x`. Because `var` always get scoped to functions. Even though variable `i` is declared inside `if` block, because of it's using `var` it get scoped to parent `function x`.

```
function x(){
  if(true){
    var i = 100;
  }
  console.log(i);
}

x();
```

Run code snippet   Expand snippet

Now variable `i` is declared inside the `function y`. Therefore, `i` scoped to `function y`. You can access `i` inside `function y`. But not from outside `function y`.

```
function x(){
  function y(){
    var i = 100;
    console.log(i);
  }

  y();
}

x();
```

Run code snippet   Expand snippet

```
function x(){
  function y(){
    var i = 100;
  }
  console.log(i); // ERROR
}

x();
```

Run code snippet   Expand snippet

let, const

let and const has block scope.

`const` and `let` behave same. But the difference is, when you assign value to `const` you can't re-assign. But you can re-assign values with `let`.

In this example, variable `i` is declared inside an `if` block. So it can be only accessed from inside that `if` block. We can't access it from outside that `if` block. (here `const` work same as `let`)

```
if(true){
  let i = 100;
  console.log(i); // Output: 100
}

console.log(i); // Error
```

Run code snippet Expand snippet

```
function x(){
  if(true){
    let i = 100;
    console.log(i); // Output: 100
  }
  console.log(i); // Error
}

x();
```

Run code snippet Expand snippet

Another difference with `(let, const)` vs `var` is you can access `var` defined variable before declaring it. It will give you `undefined`. But if you do that with `let` or `const` defined variable it will give you an error.

```
console.log(x);
var x = 100;
```

Run code snippet Expand snippet

```
console.log(x); // ERROR
let x = 100;
```

Run code snippet Expand snippet

Share Follow

answered Sep 10, 2022 at 10:48

Kasun Jalitha

771 ● 9 ● 22

11

`var` is global scope (hoist-able) variable.

`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

Share Follow

answered Oct 28, 2017 at 12:42

Moslem Shahsavan
1,337 ● 19 ● 19

11

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
    }

    SomeConstructor.prototype.showPrivate = function (
        console.log(privateScope.hiddenProperty); // b
    }

}

var myInstance = new SomeConstructor();

myInstance.showPublic();
myInstance.showPrivate();

console.log(privateScope.hiddenProperty); // error
```

See '[Emulating private interfaces]'

Share Follow

edited Jan 12, 2019 at 5:29

answered Oct 14, 2016 at 5:01

Daniel Sokolowski
12.4k ● 4 ● 73 ● 58

Can you elaborate on how Immediately Invoked Function Expressions do not provide “code protection” and `let` does? (I assume you mean IIFE with “self invoking function”.) – Robert Siemer Mar 1, 2020 at 3:58

And why do you set `hiddenProperty` in the constructor? There is only one `hiddenProperty` for all instances in your “class”. – Robert Siemer Mar 1, 2020 at 12:52

9

When Using `let`

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 variabl
    {
        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(
        function a() {
            console.log(i); //print 0 to 9, that is li
        },
        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.

Share Follow

edited May 22, 2018 at 13:22

answered May 22, 2018 at 13:12

Ankur Soni
5,998 ● 5 ● 56 ● 88

"*let variables cannot be accessed in the window object because they cannot be globally accessed.*" no, `let` declarations ***can be global***. However, *they aren't attached to the global object*. – VLAZ Sep 18, 2023 at 12:06

7

Some hacks with `let`:

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",
                   name: "foo"
               };

    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" } =

    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)
```

Share Follow

edited Nov 26, 2016 at 16:44

Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Jul 21, 2016 at 17:42

zloctb
11.2k ● 10 ● 76 ● 91

---

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 ? – AlainIb Jun 15, 2017 at 7:55

---

1 In example 3 you are re-declaring node which cause exception. These all examples also work perfectly with `var` too. – Rehan Haider Jan 9, 2019 at 10:57

This doesn't answer the question; it could benefit from an explanation as to what each block of code is doing. – TylerH Apr 13, 2021 at 18:50

1., 2., and 3. are not reliant on `let`. That's destructuring. Not something inherent to `let` declarations. You can destructure using `var` as well. Moreover, it's not at all a "hack" but part of the official language spec. jsbin.com/tibiquqohe/1/edit?js,console Basically the same with the "*Getter and setter with* `let`:" section but it shows off object initialisation syntax. Again, unrelated to any mechanics of `let`. jsbin.com/qobeyamoda/1/edit?js,console This answer does not address the question at all. – VLAZ Sep 18, 2023 at 12:04

6

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

```
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.

Share Follow

answered Sep 7, 2019 at 11:25

Piklu Dey
190 ● 1 ● 13

6

I just came across one use case that I had to use `var` over `let` to introduce new variable. Here's a case:

I want to create a new variable with dynamic variable names.

```
let variableName = 'a';
eval("let " + variableName + '= 10;');
console.log(a);   // this doesn't work
```

```
var variableName = 'a';
eval("var " + variableName + '= 10;');
console.log(a);   // this works
```

The above code doesn't work because `eval` introduces a new block of code. The declaration using `var` will declare a variable outside of this block of code since `var` declares a variable in the function scope.

`let`, on the other hand, declares a variable in a block scope. So, `a` variable will only be visible in `eval` block.

Share Follow

edited Dec 30, 2020 at 20:05

Ashish Kamble
2,627 ● 4 ● 22 ● 29

answered Oct 25, 2020 at 17:15

Sarvar Nishonboyev
13.1k ● 11 ● 78 ● 78

2 When will you ever have to create a dynamic variable name, and have to access it later? It is so much better to create an object and assign keys and values to it. – user14029620 Nov 9, 2020 at 19:18

In fact, that's because the re-declaration of a JavaScript `let` **proposition** is not allowed. – Bekim Bacaj Oct 4, 2021 at 17:46

Refer to "Variable" variables in JavaScript and use an object. – VLAZ Sep 18, 2023 at 11:56

5

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

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 
function (like var cat).
```

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

console.log(giraffe); //will print 'giraffe', as gir
(var).
console.log(lion); //will print UNDEFINED, as lion i
now out of scope.
```

Run code snippet   Expand snippet

Share Follow

edited Sep 18, 2023 at 11:57

VLAZ
28.8k ● 9 ● 62 ● 82

answered Apr 18, 2019 at 0:49

daCoda
3,825 ● 5 ● 36 ● 40

1 "*var variables are global and can be accessed basically everywhere*" ***THIS IS NOT TRUE AT ALL*** `var` declarations are either global or contained to a function. They've never been only global. `console.log(giraffe); //will print 'giraffe', as giraffe is a global variable (var).` is also completely wrong and has never been the case. – VLAZ Sep 18, 2023 at 11:59

Running the code gives: Error: { "message": "ReferenceError: giraffe is not defined", "filename": "stacksnippets.net/js", "lineno": 26, "colno": 13 } – Chris Hall Jun 5 at 11:02

1 @ChrisHall correct. That's what my comment above is. This answer is completely wrong and misleading. As is the code it provides to supposedly illustrate the non-existent language

rules. This is ***not*** how JS scoping works. – VLAZ Jun 5 at 11:06

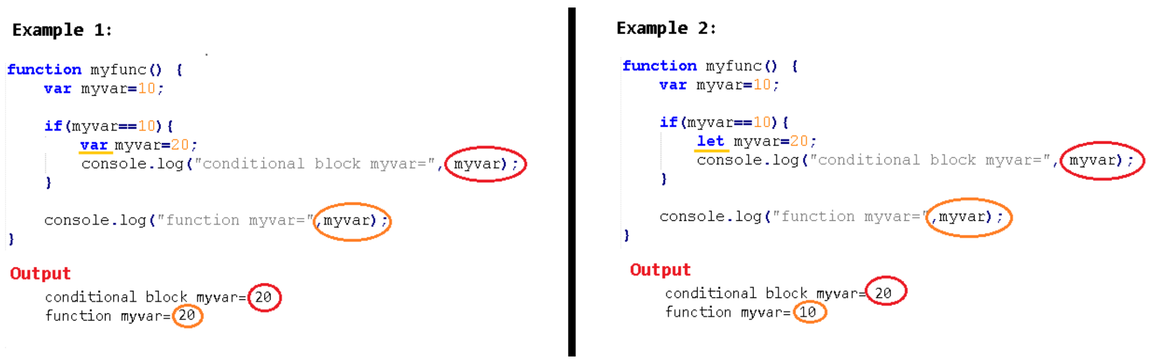
4

As mentioned above:

> 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 agian 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.

Share Follow

edited Aug 13, 2018 at 14:02

answered Aug 7, 2018 at 10:25

N Randhawa

9,461 ● 4 ● 46 ● 48

4 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, 2018 at 17:29

4

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.

However, you cannot reference the variable declared with let or const until it is declared. If you try to use it before it is declared, then an exception will be raised during the Execution Phase of the Execution Context. Now the variable will still be in memory, courtesy of the 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.

Share Follow

answered Feb 13, 2019 at 16:07

Daniel Viglione
9,378 ● 9 ● 77 ● 119

"*variables defined with var will be attached to the 'this' object.*" that is incorrect - it ***only*** happens in the global context and ***only*** because in the global context `this` is the same as the global object. There is no mechanism to attach `var` declarations to `this` otherwise. Just happens to be the cross section of two different mechanisms. jsbin.com/yujezitafo/1/edit?js,console – VLAZ Sep 18, 2023 at 11:45

"*if it cannot find [var declaration] there, it will continue searching the Scope Chain. This is not the case with let and const.*" This is also untrue - the scope chain will still be checked for `let` or `const` declarations jsbin.com/murobobiqe/1/edit?js,console – VLAZ Sep 18, 2023 at 11:47

"*The difference between let and const is that const variables need to be declared using an initializer, or it will generate an error.*" this is a backwards explanation. The main difference is that `const` *does not allow reassignment*. Thus *because* it cannot be reassigned, it needs to be initialised immediately. Otherwise an uninitialised declaration is an error as you might expect since you cannot assign a value later. – VLAZ Sep 18, 2023 at 11:51

3

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

```
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.

Share Follow

edited Nov 26, 2016 at 16:33

Peter Mortensen
31.6k ● 22 ● 109 ● 133

answered Jul 1, 2016 at 8:22

swaraj patil
209 ● 2 ● 5

"*Hoisting is not supported using let.*" Are variables declared with let or const hoisted? - the *declaration* is hoisted, not the *initialisation*. – VLAZ Sep 18, 2023 at 11:53

3

This article clearly defines the difference between var, let and const

> `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

Share Follow

edited Apr 30, 2017 at 17:11

FelixSFD
6,092 ● 10 ● 44 ● 124

answered Dec 27, 2016 at 9:44

anandharshan
6,332 ● 4 ● 37 ● 35

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.