

**Baseline** Widely available

The **var** statement declares function-scoped or globally-scoped variables, optionally initializing each to a value.

Try it

JavaScript Demo: var statement

```
1 var x = 1;
2
3 if (x === 1) {
4   var x = 2;
5
6   console.log(x);
7   // Expected output: 2
8 }
9
10 console.log(x);
11 // Expected output: 2
12
```

RunReset

Syntax

JS



```
var name1;  
var name1 = value1;  
var name1 = value1, name2 = value2;  
var name1, name2 = value2;  
var name1 = value1, name2, /* ..., */ nameN = valueN;
```

nameN

The name of the variable to declare. Each must be a legal JavaScript [identifier](#) or a [destructuring binding pattern](#).

valueN Optional

Initial value of the variable. It can be any legal expression. Default value is `undefined`.

Description

The scope of a variable declared with `var` is one of the following curly-brace-enclosed syntaxes that most closely contains the `var` statement:

- Function body
- [Static initialization block](#)

Or if none of the above applies:

- The current [module](#), for code running in module mode
- The global scope, for code running in script mode.

JS




```
function foo() {  
  var x = 1;  
  function bar() {  
    var y = 2;  
    console.log(x); // 1 (function `bar` closes over `x`)  
    console.log(y); // 2 (`y` is in scope)  
  }  
  bar();  
  console.log(x); // 1 (`x` is in scope)
```





```
console.log(y); // ReferenceError, `y` is scoped to `bar`  
}  
  
foo();
```

Importantly, other block constructs, including [block statements](#), [try...catch](#), [switch](#), headers of [one of the for statements](#), do not create scopes for `var`, and variables declared with `var` inside such a block can continue to be referenced outside the block.

```
JS   
for (var a of [1, 2, 3]);  
console.log(a); // 3
```

In a script, a variable declared using `var` is added as a non-configurable property of the global object. This means its property descriptor cannot be changed and it cannot be deleted using [delete](#). JavaScript has automatic memory management, and it would make no sense to be able to use the `delete` operator on a global variable.

```
JS   
"use strict";  
var x = 1;  
Object.hasOwn(globalThis, "x"); // true  
delete globalThis.x; // TypeError in strict mode. Fails silently  
otherwise.  
delete x; // SyntaxError in strict mode. Fails silently otherwise.
```

In both NodeJS [CommonJS](#)  modules and native [ECMAScript modules](#), top-level variable declarations are scoped to the module, and are not added as properties to the global object.

The list that follows the `var` keyword is called a [binding list](#) and is separated by commas, where the commas are *not* [comma operators](#) and the `=` signs are *not* [assignment operators](#). Initializers of later variables can refer to earlier variables in the list and get the initialized value.



Hoisting

`var` declarations, wherever they occur in a script, are processed before any code within the script is executed. Declaring a variable anywhere in the code is equivalent to declaring it at the top. This also means that a variable can appear to be used before it's declared. This behavior is called [hoisting](#), as it appears that the variable declaration is moved to the top of the function, static initialization block, or script source in which it occurs.

Note: `var` declarations are only hoisted to the top of the current script. If you have two `<script>` elements within one HTML, the first script cannot access variables declared by the second before the second script has been processed and executed.

JS



```
bla = 2;  
var bla;
```

This is implicitly understood as:

JS



```
var bla;  
bla = 2;
```

For that reason, it is recommended to always declare variables at the top of their scope (the top of global code and the top of function code) so it's clear which variables are scoped to the current function.

Only a variable's declaration is hoisted, not its initialization. The initialization happens only when the assignment statement is reached. Until then the variable remains `undefined` (but declared):

JS



```
function doSomething() {  
  console.log(bar); // undefined  
  var bar = 111;  
}
```



```
console.log(bar); // 111  
}
```

This is implicitly understood as:

```
JS  
function doSomething() {  
  var bar;  
  console.log(bar); // undefined  
  bar = 111;  
  console.log(bar); // 111  
}
```

Redeclarations

Duplicate variable declarations using `var` will not trigger an error, even in strict mode, and the variable will not lose its value, unless the declaration has an initializer.

```
JS  
var a = 1;  
var a = 2;  
console.log(a); // 2  
var a;  
console.log(a); // 2; not undefined
```

`var` declarations can also be in the same scope as a `function` declaration. In this case, the `var` declaration's initializer always overrides the function's value, regardless of their relative position. This is because function declarations are hoisted before any initializer gets evaluated, so the initializer comes later and overrides the value.

```
JS  
var a = 1;  
function a() {}  
console.log(a); // 1
```

`var` declarations cannot be in the same scope as a `let`, `const`, `class`, or `import` declaration.

JS



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



Because `var` declarations are not scoped to blocks, this also applies to the following case:

JS



```
let a = 1;
{
  var a = 1; // SyntaxError: Identifier 'a' has already been declared
}
```



It does not apply to the following case, where `let` is in a child scope of `var`, not the same scope:

JS



```
var a = 1;
{
  let a = 2;
}
```



A `var` declaration within a function's body can have the same name as a parameter.

JS



```
function foo(a) {
  var a = 1;
  console.log(a);
}
```

```
foo(2); // Logs 1
```



A `var` declaration within a `catch` block can have the same name as the `catch`-bound identifier, but only if the `catch` binding is a simple identifier, not a

destructuring pattern. This is a [deprecated syntax](#) and you should not rely on it. In this case, the declaration is hoisted to outside the `catch` block, but any value assigned within the `catch` block is not visible outside.

JS



```
try {  
  throw 1;  
} catch (e) {  
  var e = 2; // Works  
}  
  
console.log(e); // undefined
```



Examples

Declaring and initializing two variables

JS



```
var a = 0,  
    b = 0;
```

Assigning two variables with single string value

JS



```
var a = "A";  
var b = a;
```

This is equivalent to:

JS



```
var a, b = a = "A";
```

Be mindful of the order:

JS




```
var x = y,  
    y = "A";  
console.log(x, y); // undefined A
```




Here, `x` and `y` are declared before any code is executed, but the assignments occur later. At the time `x = y` is evaluated, `y` exists so no `ReferenceError` is thrown and its value is `undefined`. So, `x` is assigned the undefined value. Then, `y` is assigned the value `"A"`.

Initialization of several variables

Be careful of the `var x = y = 1` syntax — `y` is not actually declared as a variable, so `y = 1` is an [unqualified identifier assignment](#), which creates a global variable in non-strict mode.

```
JS   
  
var x = 0;  
function f() {  
  var x = y = 1; // Declares x locally; declares y globally.  
}  
f();  
  
console.log(x, y); // 0 1  
  
// In non-strict mode:  
// x is the global one as expected;  
// y is leaked outside of the function, though!
```


The same example as above but with a strict mode:

```
JS   
  
"use strict";  
  
var x = 0;  
function f() {  
  var x = y = 1; // ReferenceError: y is not defined  
}  
f();  
  
console.log(x, y);
```



Implicit globals and outer function scope

Variables that appear to be implicit globals may be references to variables in an outer function scope:

```
JS 
```

```
var x = 0; // Declares x within file scope, then assigns it a value of 0.

console.log(typeof z); // "undefined", since z doesn't exist yet

function a() {
  var y = 2; // Declares y within scope of function a, then assigns it a
  value of 2.

  console.log(x, y); // 0 2


  function b() {
    x = 3; // Assigns 3 to existing file scoped x.
    y = 4; // Assigns 4 to existing outer y.
    z = 5; // Creates a new global variable z, and assigns it a value of 5.
    // (Throws a ReferenceError in strict mode.)
  }

  b(); // Creates z as a global variable.
  console.log(x, y, z); // 3 4 5
}

a(); // Also calls b.
console.log(x, z); // 3 5
console.log(typeof y); // "undefined", as y is local to function a
```

Declaration with destructuring

The left-hand side of each `=` can also be a binding pattern. This allows creating multiple variables at once.

```
JS 
```

```
const result = /(a+)(b+)(c+)/.exec("aaabcc");
var [, a, b, c] = result;
console.log(a, b, c); // "aaa" "b" "cc"
```



For more information, see [Destructuring](#).