mdn web docs

let

Baseline Widely available

The `let` declaration declares re-assignable, block-scoped local variables, optionally initializing each to a value.

Try it

JavaScript Demo: let declaration

```
let x = 1;

if (x === 1) {
  let x = 2;

  console.log(x);
  // Expected output: 2
}

console.log(x);
// Expected output: 1
```

Run

Reset

Syntax

JS

```
let name1;
let name1 = value1;
let name1 = value1, name2 = value2;
let name1, name2 = value2;
let name1 = value1, name2, /* …, */ nameN = valueN;
```

Parameters

`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 `let` is one of the following curly-brace-enclosed syntaxes that most closely contains the `let` declaration:

- Block statement
- `switch` statement
- `try...catch` statement
- Body of one of the `for` statements, if the `let` is in the header of the 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.

Compared with `var`, `let` declarations have the following differences:

- `let` declarations are scoped to blocks as well as functions.
- `let` declarations can only be accessed after the place of declaration is reached (see temporal dead zone). For this reason, `let` declarations are commonly regarded as non-hoisted.
- `let` declarations do not create properties on `globalThis` when declared at the top level of a script.
- `let` declarations cannot be redeclared by any other declaration in the same scope.
- `let` begins *declarations, not statements*. That means you cannot use a lone `let` declaration as the body of a block (which makes sense, since there's no way to access the variable).

```JS
if (true) let a = 1; // SyntaxError: Lexical declaration cannot appear in a single-statement context
```

Note that `let` is allowed as an identifier name when declared with `var` or `function` in non-strict mode, but you should avoid using `let` as an identifier name to prevent unexpected syntax ambiguities.

Many style guides (including MDN's) recommend using `const` over `let` whenever a variable is not reassigned in its scope. This makes the intent clear that a variable's type (or value, in the case of a primitive) can never change. Others may prefer `let` for non-primitives that are mutated.

The list that follows the `let` 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.

Temporal dead zone (TDZ)

A variable declared with `let`, `const`, or `class` is said to be in a "temporal dead zone" (TDZ) from the start of the block until code execution reaches the place where the variable is declared and initialized.

While inside the TDZ, the variable has not been initialized with a value, and any attempt to access it will result in a `ReferenceError`. The variable is initialized with a value when execution reaches the place in the code where it was declared. If no initial value was specified with the variable declaration, it will be initialized with a value of `undefined`.

This differs from `var` variables, which will return a value of `undefined` if they are accessed before they are declared. The code below demonstrates the different result when `let` and `var` are accessed in code before the place where they are declared.

JS

```js
{
  // TDZ starts at beginning of scope
  console.log(bar); // "undefined"
  console.log(foo); // ReferenceError: Cannot access 'foo' before initialization
  var bar = 1;
  let foo = 2; // End of TDZ (for foo)
}
```

The term "temporal" is used because the zone depends on the order of execution (time) rather than the order in which the code is written (position). For example, the code below works because, even though the function that uses the `let` variable appears before the variable is declared, the function is *called* outside the TDZ.

JS

```js
{
  // TDZ starts at beginning of scope
  const func = () => console.log(letVar); // OK

  // Within the TDZ letVar access throws `ReferenceError`

  let letVar = 3; // End of TDZ (for letVar)
  func(); // Called outside TDZ!
}
```

Using the `typeof` operator for a variable in its TDZ will throw a `ReferenceError`:

```js
{
  typeof i; // ReferenceError: Cannot access 'i' before initialization
  let i = 10;
}
```

This differs from using `typeof` for undeclared variables, and variables that hold a value of `undefined`:

```js
console.log(typeof undeclaredVariable); // "undefined"
```

> **Note:** `let` and `const` declarations are only processed when the current script gets processed. If you have two `<script>` elements running in script mode within one HTML, the first script is not subject to the TDZ restrictions for top-level `let` or `const` variables declared in the second script, although if you declare a `let` or `const` variable in the first script, declaring it again in the second script will cause a redeclaration error.

Redeclarations

`let` declarations cannot be in the same scope as any other declaration, including `let`, `const`, `class`, `function`, `var`, and `import` declaration.

```js
{
  let foo;
  let foo; // SyntaxError: Identifier 'foo' has already been declared
}
```

A `let` declaration within a function's body cannot have the same name as a parameter. A `let` declaration within a `catch` block cannot have the same name

as the `catch`-bound identifier.

```js
function foo(a) {
  let a = 1; // SyntaxError: Identifier 'a' has already been declared
}
try {
} catch (e) {
  let e; // SyntaxError: Identifier 'e' has already been declared
}
```

If you're experimenting in a REPL, such as the Firefox web console (**Tools** > **Web Developer** > **Web Console**), and you run two `let` declarations with the same name in two separate inputs, you may get the same re-declaration error. See further discussion of this issue in Firefox bug 1580891. The Chrome console allows `let` re-declarations between different REPL inputs.

You may encounter errors in `switch` statements because there is only one block.

```js
let x = 1;

switch (x) {
  case 0:
    let foo;
    break;
  case 1:
    let foo; // SyntaxError: Identifier 'foo' has already been declared
    break;
}
```

To avoid the error, wrap each `case` in a new block statement.

```js
let x = 1;

switch (x) {
```

```
  case 0: {
    let foo;
    break;
  }
  case 1: {
    let foo;
    break;
  }
}
```

Examples

Scoping rules

Variables declared by `let` have their scope in the block for which they are declared, 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:

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

function letTest() {
  let x = 1;
  {
    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:

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

TDZ combined with lexical scoping

The following code results in a `ReferenceError` at the line shown:

```js
function test() {
  var foo = 33;
  if (foo) {
    let foo = foo + 55; // ReferenceError
  }
}
test();
```

The `if` block is evaluated because the outer `var foo` has a value. However due to lexical scoping this value is not available inside the block: the identifier `foo` *inside* the `if` block is the `let foo`. The expression `foo + 55` throws a `ReferenceError` because initialization of `let foo` has not completed — it is still in the temporal dead zone.

This phenomenon can be confusing in a situation like the following. The instruction `let n of n.a` is already inside the scope of the `for...of` loop's block. So, the identifier `n.a` is resolved to the property `a` of the `n` object located in the first part of the instruction itself (`let n`). This is still in the temporal dead zone as its declaration statement has not been reached and terminated.

```js
function go(n) {
  // n here is defined!
  console.log(n); // { a: [1, 2, 3] }

  for (let n of n.a) {
    //          ^ ReferenceError
```

```
    console.log(n);
  }
}

go({ a: [1, 2, 3] });
```

Other situations

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.

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

{
  var a = 11; // the scope is global
  let b = 22; // the scope is inside the block

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

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

However, this combination of `var` and `let` declarations below is a `SyntaxError` because `var` not being block-scoped, leading to them being in the same scope. This results in an implicit re-declaration of the variable.

```JS
let x = 1;

{
  var x = 2; // SyntaxError for re-declaration
}
```

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");
let [, a, b, c] = result;
console.log(a, b, c); // "aaa" "b" "cc"
```

For more information, see Destructuring.

Specifications

Specification
ECMAScript® 2026 Language Specification # sec-let-and-const-declarations

Browser compatibility

Report problems with this compatibility data • View data on GitHub

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android	WebView on iOS	Deno	Node.js
`let`	✓ 49 ...	✓ 14 ...	✓ 44 *	✓ 17	✓ 10	✓ 49 ...	✓ 44 *	✓ 18	✓ 10	✓ 5 ...	✓ 49 ...	✓ 10	✓ 1	✓ 6

Tip: you can click/tap on a cell for more information.

✓ Full support Partial support * See implementation notes.

... Has more compatibility info.

See also