**Const & let**

The **const** declaration creates block-scoped constants, much like variables declared using the [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let) keyword. The value of a constant can't be changed through reassignment (i.e. by using the [assignment operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment)), and it can't be redeclared (i.e. through a [variable declaration](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_types#declarations)). However, if a constant is an [object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object) or [array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) its properties or items can be updated or removed.

[**Try it**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const#try_it)

[**Syntax**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const#syntax)

const name1 = value1 [, name2 = value2 [, ... [, nameN = valueN]]]

Copy to Clipboard

nameN

The constant's name, which can be any legal [identifier](https://developer.mozilla.org/en-US/docs/Glossary/Identifier).

valueN

The constant's value. This can be any legal expression, including a function expression.

The [destructuring assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) syntax can also be used to declare variables.

const { bar } = foo; // where foo = { bar:10, baz:12 };

/\* This creates a constant with the name 'bar', which has a value of 10 \*/

Copy to Clipboard

[**Description**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const#description)

This declaration creates a constant whose scope can be either global or local to the block in which it is declared. Global constants do **not** become properties of the [window](https://developer.mozilla.org/en-US/docs/Web/API/Window) object, unlike [var](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var) variables.

An initializer for a constant is required. You must specify its value in the same declaration. (This makes sense, given that it can't be changed later.)

The **const declaration** creates a read-only reference to a value. It does **not** mean the value it holds is immutable—just that the variable identifier cannot be reassigned. For instance, in the case where the content is an object, this means the object's contents (e.g., its properties) can be altered.

All the considerations about the [temporal dead zone](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal_dead_zone_tdz) apply to both [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let) and const. For this reason, const declarations are commonly regarded as [non-hoisted](https://developer.mozilla.org/en-US/docs/Glossary/Hoisting).

A constant cannot share its name with a function or a variable in the same scope.

Unlike var, const begins [*declarations*, not *statements*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements#difference_between_statements_and_declarations). That means you cannot use a lone const declaration as the body of a block (which makes sense, since there's no way to access the variable).

if (true) const a = 1; // SyntaxError: Unexpected token 'const'

[**Examples**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const#examples)

[**Basic const usage**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const#basic_const_usage)

Constants can be declared with uppercase or lowercase, but a common convention is to use all-uppercase letters.

// define MY\_FAV as a constant and give it the value 7

const MY\_FAV = 7;

// this will throw an error - Uncaught TypeError: Assignment to constant variable.

MY\_FAV = 20;

// MY\_FAV is 7

console.log("my favorite number is: " + MY\_FAV);

// trying to redeclare a constant throws an error

// Uncaught SyntaxError: Identifier 'MY\_FAV' has already been declared

const MY\_FAV = 20;

// the name MY\_FAV is reserved for constant above, so this will fail too

var MY\_FAV = 20;

// this throws an error too

let MY\_FAV = 20;

Copy to Clipboard

[**Block scoping**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const#block_scoping)

It's important to note the nature of block scoping.

if (MY\_FAV === 7) {

// this is fine and creates a block scoped MY\_FAV variable

// (works equally well with let to declare a block scoped non const variable)

let MY\_FAV = 20;

// MY\_FAV is now 20

console.log("my favorite number is " + MY\_FAV);

// this gets hoisted into the global context and throws an error

var MY\_FAV = 20;

}

// MY\_FAV is still 7

console.log("my favorite number is " + MY\_FAV);

Copy to Clipboard

[**const needs to be initialized**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const#const_needs_to_be_initialized)

// throws an error

// Uncaught SyntaxError: Missing initializer in const declaration

const FOO;

[**const in objects and arrays**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const#const_in_objects_and_arrays)

const also works on objects and arrays. Attempting to overwrite the object throws an error "Assignment to constant variable".

const MY\_OBJECT = { key: "value" };

MY\_OBJECT = { OTHER\_KEY: "value" };

However, object keys are not protected, so the following statement is executed without problem.

MY\_OBJECT.key = "otherValue";

Copy to Clipboard

You would need to use [Object.freeze()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze) to make an object immutable.

The same applies to arrays. Assigning a new array to the variable throws an error "Assignment to constant variable".

const MY\_ARRAY = [];

MY\_ARRAY = ["B"];

Still, it's possible to push items into the array and thus mutate it.

MY\_ARRAY.push("A"); // ["A"]

# let

The **let** declaration declares a block-scoped local variable, optionally initializing it to a value.

## [Try it](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#try_it)

## [Syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#syntax)

let name1 [= value1] [, name2 [= value2]] [, ..., nameN [= valueN]

Copy to Clipboard

### [Parameters](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#parameters)

nameN

The names of the variable or variables to declare. Each must be a legal JavaScript identifier.

valueN Optional

For each variable declared, you may optionally specify its initial value to any legal JavaScript expression.

Alternatively, the [Destructuring Assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) syntax can also be used to declare variables.

let { bar } = foo; // where foo = { bar:10, baz:12 };

/\* This creates a variable with the name 'bar', which has a value of 10 \*/

Copy to Clipboard

## [Description](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#description)

**let** allows you to declare variables that are limited to the scope of a [block](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/block) statement, or expression on which it is used, unlike the [var](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var) keyword, which declares a variable globally, or locally to an entire function regardless of block scope. The other difference between [var](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var) and let is that the latter can only be accessed after its declaration is reached (see [temporal dead zone](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal_dead_zone_tdz)). For this reason, let declarations are commonly regarded as [non-hoisted](https://developer.mozilla.org/en-US/docs/Glossary/Hoisting).

Just like [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const#description) the let does not create properties of the [window](https://developer.mozilla.org/en-US/docs/Web/API/Window) object when declared globally (in the top-most scope).

An explanation of why the name "**let**" was chosen can be found [here](https://stackoverflow.com/questions/37916940/why-was-the-name-let-chosen-for-block-scoped-variable-declarations-in-javascri).

Many issues with let variables can be avoided by declaring them at the top of the scope in which they are used (doing so may impact readability).

Unlike var, let begins [declarations, not statements](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements#difference_between_statements_and_declarations). 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).

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

## [Examples](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#examples)

### [Scoping rules](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#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:

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

}

Copy to Clipboard

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

Copy to Clipboard

### [Redeclarations](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#redeclarations)

Redeclaring the same variable within the same function or block scope raises a [SyntaxError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError).

if (x) {

let foo;

let foo; // SyntaxError thrown.

}

You may encounter errors in [switch](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch) statements because there is only one block.

let x = 1;

switch(x) {

case 0:

let foo;

break;

case 1:

let foo; // SyntaxError for redeclaration.

break;

}

However, it's important to point out that a block nested inside a case clause will create a new block scoped lexical environment, which will not produce the redeclaration errors shown above.

let x = 1;

switch(x) {

case 0: {

let foo;

break;

}

case 1: {

let foo;

break;

}

}

Copy to Clipboard

### [Temporal dead zone (TDZ)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#temporal_dead_zone_tdz)

A let or const variable is said to be in a "temporal dead zone" (TDZ) from the start of the block until code execution reaches the line 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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError). The variable is initialized with a value when execution reaches the line of 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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var#var_hoisting) 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 line in which they are declared.

{ // TDZ starts at beginning of scope

console.log(bar); // undefined

console.log(foo); // ReferenceError

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.

{

// 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!

}

Copy to Clipboard

#### The TDZ and typeof

Using the typeof operator for a let variable in its TDZ will throw a [ReferenceError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError):

// results in a 'ReferenceError'

console.log(typeof i);

let i = 10;

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

// prints out 'undefined'

console.log(typeof undeclaredVariable);

Copy to Clipboard

#### TDZ combined with lexical scoping

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

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 private scope of the for 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.

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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let#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.

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

Copy to Clipboard

However, this combination of **var** and **let** declaration below is a [SyntaxError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError) due to **var** being hoisted to the top of the block. This results in an implicit re-declaration of the variable.

let x = 1;

{

var x = 2; // SyntaxError for re-declaration

}