**For loops**

**Regular loop**

The **for** statement creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a statement (usually a [block statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/block)) to be executed in the loop.

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

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

for (initialization; condition; afterthought)

statement

Copy to Clipboard

initialization Optional

An expression (including [assignment expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment)) or variable declaration evaluated once before the loop begins. Typically used to initialize a counter variable. This expression may optionally declare new variables with var or let keywords. Variables declared with var are not local to the loop, i.e. they are in the same scope the for loop is in. Variables declared with let are local to the statement.

The result of this expression is discarded.

condition Optional

An expression to be evaluated before each loop iteration. If this expression [evaluates to true](https://developer.mozilla.org/en-US/docs/Glossary/Truthy), statement is executed. If the expression [evaluates to false](https://developer.mozilla.org/en-US/docs/Glossary/Falsy), execution exits the loop and goes to the first statement after the for construct.

This conditional test is optional. If omitted, the condition always evaluates to true.

afterthought Optional

An expression to be evaluated at the end of each loop iteration. This occurs before the next evaluation of condition. Generally used to update or increment the counter variable.

statement

A statement that is executed as long as the condition evaluates to true. You can use a [block statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/block) to execute multiple statements. To execute no statement within the loop, use an [empty statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/Empty) (;).

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

[**Using for**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for#using_for)

The following for statement starts by declaring the variable i and initializing it to 0. It checks that i is less than nine, performs the two succeeding statements, and increments i by 1 after each pass through the loop.

for (let i = 0; i < 9; i++) {

console.log(i);

// more statements

}

Copy to Clipboard

[**Initialization block syntax**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for#initialization_block_syntax)

The initialization block accepts both expressions and variable declarations. However, expressions cannot use the [in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/in) operator unparenthesized, because that is ambiguous with a [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loop.

for (let i = "start" in window ? window.start : 0; i < 9; i++) {

console.log(i);

}

// SyntaxError: 'for-in' loop variable declaration may not have an initializer.

// Parenthesize the whole initializer

for (let i = ("start" in window ? window.start : 0); i < 9; i++) {

console.log(i);

}

// Parenthesize the `in` expression

for (let i = ("start" in window) ? window.start : 0; i < 9; i++) {

console.log(i);

}

[**Optional for expressions**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for#optional_for_expressions)

All three expressions in the head of the for loop are optional. For example, it is not required to use the initialization block to initialize variables:

let i = 0;

for (; i < 9; i++) {

console.log(i);

// more statements

}

Copy to Clipboard

Like the initialization block, the condition part is also optional. If you are omitting this expression, you must make sure to break the loop in the body in order to not create an infinite loop.

for (let i = 0; ; i++) {

console.log(i);

if (i > 3) break;

// more statements

}

Copy to Clipboard

You can also omit all three expressions. Again, make sure to use a [break](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/break) statement to end the loop and also modify (increase) a variable, so that the condition for the break statement is true at some point.

let i = 0;

for (;;) {

if (i > 3) break;

console.log(i);

i++;

}

Copy to Clipboard

However, in the case where you are not fully using all three expression positions — especially if you are not declaring variables with the first expression but mutating something in the upper scope — consider using a [while](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/while) loop instead, which makes the intention clearer.

let i = 0;

while (i <= 3) {

console.log(i);

i++;

}

Copy to Clipboard

[**Lexical declarations in the initialization block**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for#lexical_declarations_in_the_initialization_block)

Declaring a variable within the initialization block has important differences from declaring it in the upper [scope](https://developer.mozilla.org/en-US/docs/Glossary/Scope), especially when creating a [closure](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures) within the loop body. For example, for the code below:

for (let i = 0; i < 3; i++) {

setTimeout(() => {

console.log(i);

}, 1000);

}

Copy to Clipboard

It logs 0, 1, and 2, as expected. However, if the variable is defined in the upper scope:

let i = 0;

for (; i < 3; i++) {

setTimeout(() => {

console.log(i);

}, 1000);

}

Copy to Clipboard

It logs 3, 3, and 3. The reason is that each setTimeout creates a new closure that closes over the i variable, but if the i is not scoped to the loop body, all closures will reference the same variable when they eventually get called — and due to the asynchronous nature of [setTimeout](https://developer.mozilla.org/en-US/docs/Web/API/setTimeout), it will happen after the loop has already exited, causing the value of i in all queued callbacks' bodies to have the value of 3.

This also happens if you use a var statement as the initialization, because variables declared with var are only function-scoped, but not lexically scoped (i.e. they can't be scoped to the loop body).

for (var i = 0; i < 3; i++) {

setTimeout(() => {

console.log(i);

}, 1000);

}

// Logs 3, 3, 3

Copy to Clipboard

The scoping effect of the initialization block can be understood as if the declaration happens within the loop body, but just happens to be accessible within the condition and afterthought parts. More precisely, let declarations are special-cased by for loops — if initialization is a let declaration, then every time, after the loop body is evaluated, the following happens:

1. A new lexical scope is created with new let-declared variables.
2. The binding values from the last iteration are used to re-initialize the new variables.
3. afterthought is evaluated in the new scope.

So re-assigning the new variables within afterthought does not affect the bindings from the previous iteration.

Creating closures allows you to get hold of a binding during any particular iteration. This explains why closures created within the initialization section do not get updated by re-assignments of i in the afterthought.

for (let i = 0, getI = () => i; i < 3; i++) {

console.log(getI());

}

// Logs 0, 0, 0

Copy to Clipboard

This does not log "0, 1, 2", like what would happen if getI is declared in the loop body. This is because getI is not re-evaluated on each iteration — rather, the function is created once and closes over the i variable, which refers to the variable declared when the loop was first initialized. Subsequent updates to the value of i actually create new variables called i, which getI does not see. A way to fix this is to re-compute getI every time i updates:

for (let i = 0, getI = () => i; i < 3; i++, getI = () => i) {

console.log(getI());

}

// Logs 0, 1, 2

Copy to Clipboard

In fact, you can capture the initial binding of the i variable and re-assign it later, and this updated value will not be visible to the loop body, which sees the next new binding of i.

for (

let i = 0, getI = () => i, incrementI = () => i++;

getI() < 3;

incrementI()

) {

console.log(i);

}

// Logs 0, 0, 0

Copy to Clipboard

This logs "0, 0, 0", because the i variable in each loop evaluation is actually a separate variable, but getI and incrementI both read and write the *initial* binding of i, not what was subsequently declared.

[**Using for without a body**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for#using_for_without_a_body)

The following for cycle calculates the offset position of a node in the afterthought section, and therefore it does not require the use of a statement section, a semicolon is used instead.

function showOffsetPos(id) {

let left = 0;

let top = 0;

for (

let itNode = document.getElementById(id); // initialization

itNode; // condition

left += itNode.offsetLeft,

top += itNode.offsetTop,

itNode = itNode.offsetParent // afterthought

); // semicolon

console.log(

`Offset position of "${id}" element:

left: ${left}px;

top: ${top}px;`,

);

}

showOffsetPos("content");

// Logs:

// Offset position of "content" element:

// left: 0px;

// top: 153px;

Copy to Clipboard

Note that the semicolon after the for statement is mandatory, because it stands as an [empty statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/Empty). Otherwise, the for statement acquires the following console.log line as its statement section, which makes the log execute multiple times.

[**Using for with two iterating variables**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for#using_for_with_two_iterating_variables)

You can create two counters that are updated simultaneously in a for loop using the [comma operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comma_Operator). Multiple let and var declarations can also be joined with commas.

const arr = [1, 2, 3, 4, 5, 6];

for (let l = 0, r = arr.length - 1; l < r; l++, r--) {

console.log(arr[l], arr[r]);

}

// 1 6

// 2 5

// 3 4

Copy to Clipboard

# for await...of

The **for await...of** statement creates a loop iterating over [async iterable objects](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_async_iterator_and_async_iterable_protocols) as well as [sync iterables](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_iterable_protocol). This statement can only be used in contexts where [await](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await) can be used, which includes inside an [async function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) body and in a [module](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules).

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

**Note:** for await...of doesn't work with async iterators that are not async iterables.

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

for await (variable of iterable)

statement

Copy to Clipboard

variable

Receives a value from the sequence on each iteration. May be either a declaration with [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const), [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let), or [var](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var), or an [assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment) target (e.g. a previously declared variable or an object property).

iterable

An async iterable or sync iterable. The source of the sequence of values on which the loop operates.

statement

A statement to be executed on every iteration. May reference variable. You can use a [block statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/block) to execute multiple statements.

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

When a for await...of loop iterates over an iterable, it first gets the iterable's [[@@asyncIterator]()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/asyncIterator) method and calls it, which returns an [async iterator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_async_iterator_and_async_iterable_protocols). If the @asyncIterator method does not exist, it then looks for an [[@@iterator]()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/iterator) method, which returns a [sync iterator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_iterator_protocol). The sync iterator returned is then wrapped into an async iterator by wrapping every object returned from the next(), return(), and throw() methods into a resolved or rejected promise, with the value property resolved if it's also a promise. The loop then repeatedly calls the final async iterator's [next()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_iterator_protocol) method and [awaits](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await) the returned promise, producing the sequence of values to be assigned to variable.

If the for await...of loop exited early (e.g. a break statement is encountered or an error is thrown), the [return()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_iterator_protocol) method of the iterator is called to perform any cleanup. The returned promise is awaited before the loop exits.

for await...of generally functions the same as the [for...of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of) loop and shares many of the same syntax and semantics. There are a few differences:

* for await...of works on both sync and async iterables, while for...of only works on sync iterables.
* for await...of can only be used in contexts where [await](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await) can be used, which includes inside an [async function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) body and in a [module](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules). Even when the iterable is sync, the loop still awaits the return value for every iteration, leading to slower execution due to repeated promise unwrapping.
* If the iterable is a sync iterable that yields promises, for await...of would produce a sequence of resolved values, while for...of would produce a sequence of promises. (However, beware of error handling and cleanup — see [Iterating over sync iterables and generators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for-await...of#iterating_over_sync_iterables_and_generators))
* For for await...of, the variable can be the identifier async (e.g. for await (async of foo)); for...of forbids this case.

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

### [Iterating over async iterables](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for-await...of#iterating_over_async_iterables)

You can also iterate over an object that explicitly implements async iterable protocol:

const LIMIT = 3;

const asyncIterable = {

[Symbol.asyncIterator]() {

let i = 0;

return {

next() {

const done = i === LIMIT;

const value = done ? undefined : i++;

return Promise.resolve({ value, done });

},

return() {

// This will be reached if the consumer called 'break' or 'return' early in the loop.

return { done: true };

},

};

},

};

(async () => {

for await (const num of asyncIterable) {

console.log(num);

}

})();

// 0

// 1

// 2

Copy to Clipboard

### [Iterating over async generators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for-await...of#iterating_over_async_generators)

Since the return values of async generator functions conform to the async iterable protocol, they can be looped using for await...of.

async function\* asyncGenerator() {

let i = 0;

while (i < 3) {

yield i++;

}

}

(async () => {

for await (const num of asyncGenerator()) {

console.log(num);

}

})();

// 0

// 1

// 2

Copy to Clipboard

For a more concrete example of iterating over an async generator using for await...of, consider iterating over data from an API.

This example first creates an async iterable for a stream of data, then uses it to find the size of the response from the API.

async function\* streamAsyncIterable(stream) {

const reader = stream.getReader();

try {

while (true) {

const { done, value } = await reader.read();

if (done) return;

yield value;

}

} finally {

reader.releaseLock();

}

}

// Fetches data from URL and calculates response size using the async generator.

async function getResponseSize(url) {

const response = await fetch(url);

// Will hold the size of the response, in bytes.

let responseSize = 0;

// The for-await-of loop. Async iterates over each portion of the response.

for await (const chunk of streamAsyncIterable(response.body)) {

// Incrementing the total response length.

responseSize += chunk.length;

}

console.log(`Response Size: ${responseSize} bytes`); // "Response Size: 1071472"

return responseSize;

}

getResponseSize("https://jsonplaceholder.typicode.com/photos");

Copy to Clipboard

### [Iterating over sync iterables and generators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for-await...of#iterating_over_sync_iterables_and_generators)

for await...of loop also consumes sync iterables and generators. In that case it internally awaits emitted values before assign them to the loop control variable.

function\* generator() {

yield 0;

yield 1;

yield Promise.resolve(2);

yield Promise.resolve(3);

yield 4;

}

(async () => {

for await (const num of generator()) {

console.log(num);

}

})();

// 0

// 1

// 2

// 3

// 4

// compare with for-of loop:

for (const numOrPromise of generator()) {

console.log(numOrPromise);

}

// 0

// 1

// Promise { 2 }

// Promise { 3 }

// 4

Copy to Clipboard

**Note:** Be aware of yielding rejected promises from a sync generator. In such case, for await...of throws when consuming the rejected promise and DOESN'T CALL finally blocks within that generator. This can be undesirable if you need to free some allocated resources with try/finally.

function\* generatorWithRejectedPromises() {

try {

yield 0;

yield 1;

yield Promise.resolve(2);

yield Promise.reject(3);

yield 4;

throw 5;

} finally {

console.log("called finally");

}

}

(async () => {

try {

for await (const num of generatorWithRejectedPromises()) {

console.log(num);

}

} catch (e) {

console.log("caught", e);

}

})();

// 0

// 1

// 2

// caught 3

// compare with for-of loop:

try {

for (const numOrPromise of generatorWithRejectedPromises()) {

console.log(numOrPromise);

}

} catch (e) {

console.log("caught", e);

}

// 0

// 1

// Promise { 2 }

// Promise { <rejected> 3 }

// 4

// caught 5

// called finally

Copy to Clipboard

To make finally blocks of a sync generator always called, use the appropriate form of the loop — for await...of for the async generator and for...of for the sync one — and await yielded promises explicitly inside the loop.

(async () => {

try {

for (const numOrPromise of generatorWithRejectedPromises()) {

console.log(await numOrPromise);

}

} catch (e) {

console.log("caught", e);

}

})();

// 0

// 1

// 2

// caught 3

// called finally

Copy to Clipboard

# for...in

The **for...in** statement iterates over all [enumerable string properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Enumerability_and_ownership_of_properties) of an object (ignoring properties keyed by [symbols](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol)), including inherited enumerable properties.

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

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

for (variable in object)

statement

Copy to Clipboard

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

variable

Receives a string property name on each iteration. May be either a declaration with [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const), [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let), or [var](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var), or an [assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment) target (e.g. a previously declared variable or an object property).

object

Object whose non-symbol enumerable properties are iterated over.

statement

A statement to be executed on every iteration. May reference variable. You can use a [block statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/block) to execute multiple statements.

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

The loop will iterate over all enumerable properties of the object itself and those the object inherits from its prototype chain (properties of nearer prototypes take precedence over those of prototypes further away from the object in its prototype chain).

A for...in loop only iterates over enumerable, non-symbol properties. Objects created from built–in constructors like Array and Object have inherited non–enumerable properties from Array.prototype and Object.prototype, such as [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)'s [indexOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf) method or [Object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object)'s [toString()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/toString) method, which will not be visited in the for...in loop.

The traversal order, as of modern ECMAScript specification, is well-defined and consistent across implementations. Within each component of the prototype chain, all non-negative integer keys (those that can be array indices) will be traversed first in ascending order by value, then other string keys in ascending chronological order of property creation.

The variable part of for...in accepts anything that can come before the = operator. You can use [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const) to declare the variable as long as it's not reassigned within the loop body (it can change between iterations, because those are two separate variables). Otherwise, you can use [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let). You can use [destructuring](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) or an object property like for (x.y in iterable) as well.

A [legacy syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Deprecated_and_obsolete_features#statements) allows var declarations of the loop variable to have an initializer. This throws a [syntax error](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors/Invalid_for-in_initializer) in strict mode and is ignored in non–strict mode.

### [Deleted, added, or modified properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in#deleted_added_or_modified_properties)

If a property is modified in one iteration and then visited at a later time, its value in the loop is its value at that later time. A property that is deleted before it has been visited will not be visited later. Properties added to the object over which iteration is occurring may either be visited or omitted from iteration.

In general, it is best not to add, modify, or remove properties from the object during iteration, other than the property currently being visited. There is no guarantee whether an added property will be visited, whether a modified property (other than the current one) will be visited before or after it is modified, or whether a deleted property will be visited before it is deleted.

### [Array iteration and for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in#array_iteration_and_for...in)

Array indexes are just enumerable properties with integer names and are otherwise identical to general object properties. The for...in loop will traverse all integer keys before traversing other keys, and in strictly increasing order, making the behavior of for...in close to normal array iteration. However, the for...in loop will return all enumerable properties, including those with non–integer names and those that are inherited. Unlike for...of, for...in uses property enumeration instead of the array's iterator. In [sparse arrays](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#sparse_arrays), for...of will visit the empty slots, but for...in will not.

It is better to use a [for](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for) loop with a numeric index, [Array.prototype.forEach()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach), or the [for...of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of) loop, because they will return the index as a number instead of a string, and also avoid non-index properties.

### [Iterating over own properties only](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in#iterating_over_own_properties_only)

If you only want to consider properties attached to the object itself, and not its prototypes, you can use one of the following techniques:

* [Object.keys(myObject)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys)
* [Object.getOwnPropertyNames(myObject)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyNames)

Object.keys will return a list of enumerable own string properties, while Object.getOwnPropertyNames will also contain non-enumerable ones.

Many JavaScript style guides and linters recommend against the use of for...in, because it iterates over the entire prototype chain which is rarely what one wants, and may be a confusion with the more widely-used for...of loop. for...in is most practically used for debugging purposes, being an easy way to check the properties of an object (by outputting to the console or otherwise). In situations where objects are used as ad hoc key-value pairs, for...in allows you check if any of those keys hold a particular value.

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

### [Using for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in#using_for...in)

The for...in loop below iterates over all of the object's enumerable, non-symbol properties and logs a string of the property names and their values.

const obj = { a: 1, b: 2, c: 3 };

for (const prop in obj) {

console.log(`obj.${prop} = ${obj[prop]}`);

}

// Logs:

// "obj.a = 1"

// "obj.b = 2"

// "obj.c = 3"

Copy to Clipboard

### [Iterating own properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in#iterating_own_properties)

The following function illustrates the use of [Object.hasOwn()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn): the inherited properties are not displayed.

const triangle = { a: 1, b: 2, c: 3 };

function ColoredTriangle() {

this.color = "red";

}

ColoredTriangle.prototype = triangle;

const obj = new ColoredTriangle();

for (const prop in obj) {

if (Object.hasOwn(obj, prop)) {

console.log(`obj.${prop} = ${obj[prop]}`);

}

}

// Logs:

// "obj.color = red"

Copy to Clipboard

# for...of

The **for...of** statement executes a loop that operates on a sequence of values sourced from an [iterable object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_iterable_protocol). Iterable objects include instances of built-ins such as [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array), [String](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String), [TypedArray](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypedArray), [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map), [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set), [NodeList](https://developer.mozilla.org/en-US/docs/Web/API/NodeList) (and other DOM collections), as well as the [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments) object, [generators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator) produced by [generator functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*), and user-defined iterables.

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

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

for (variable of iterable)

statement

Copy to Clipboard

variable

Receives a value from the sequence on each iteration. May be either a declaration with [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const), [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let), or [var](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var), or an [assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Assignment) target (e.g. a previously declared variable or an object property).

iterable

An iterable object. The source of the sequence of values on which the loop operates.

statement

A statement to be executed on every iteration. May reference variable. You can use a [block statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/block) to execute multiple statements.

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

A for...of loop operates on the values sourced from an iterable one by one in sequential order. Each operation of the loop on a value is called an iteration, and the loop is said to iterate over the iterable. Each iteration executes statements that may refer to the current sequence value.

When a for...of loop iterates over an iterable, it first calls the iterable's [[@@iterator]()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/iterator) method, which returns an [iterator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_iterator_protocol), and then repeatedly calls the resulting iterator's [next()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_iterator_protocol) method to produce the sequence of values to be assigned to variable.

A for...of loop exits when the iterator has completed (the iterator's next() method returns an object containing done: true). You may also use control flow statements to change the normal control flow. [break](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/break) exits the loop and goes to the first statement after the loop body, while [continue](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/continue) skips the rest of the statements of the current iteration and proceeds to the next iteration.

If the for...of loop exited early (e.g. a break statement is encountered or an error is thrown), the [return()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_iterator_protocol) method of the iterator is called to perform any cleanup.

The variable part of for...of accepts anything that can come before the = operator. You can use [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const) to declare the variable as long as it's not reassigned within the loop body (it can change between iterations, because those are two separate variables). Otherwise, you can use [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let).

const iterable = [10, 20, 30];

for (let value of iterable) {

value += 1;

console.log(value);

}

// 11

// 21

// 31

Copy to Clipboard

**Note:** Each iteration creates a new variable. Reassigning the variable inside the loop body does not affect the original value in the iterable (an array, in this case).

You can use [destructuring](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) or an object property like for (x.y of iterable) as well.

However, a special rule forbids using async as the variable name. This is invalid syntax:

let async;

for (async of [1, 2, 3]); // SyntaxError: The left-hand side of a for-of loop may not be 'async'.

This is to avoid syntax ambiguity with the valid code for (async of => {};;), which is a [for](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for) loop.

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

### [Iterating over an Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of#iterating_over_an_array)

const iterable = [10, 20, 30];

for (const value of iterable) {

console.log(value);

}

// 10

// 20

// 30

Copy to Clipboard

### [Iterating over a string](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of#iterating_over_a_string)

Strings are [iterated by Unicode code points](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/@@iterator).

const iterable = "boo";

for (const value of iterable) {

console.log(value);

}

// "b"

// "o"

// "o"

Copy to Clipboard

### [Iterating over a TypedArray](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of#iterating_over_a_typedarray)

const iterable = new Uint8Array([0x00, 0xff]);

for (const value of iterable) {

console.log(value);

}

// 0

// 255

Copy to Clipboard

### [Iterating over a Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of#iterating_over_a_map)

const iterable = new Map([

["a", 1],

["b", 2],

["c", 3],

]);

for (const entry of iterable) {

console.log(entry);

}

// ['a', 1]

// ['b', 2]

// ['c', 3]

for (const [key, value] of iterable) {

console.log(value);

}

// 1

// 2

// 3

Copy to Clipboard

### [Iterating over a Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of#iterating_over_a_set)

const iterable = new Set([1, 1, 2, 2, 3, 3]);

for (const value of iterable) {

console.log(value);

}

// 1

// 2

// 3

Copy to Clipboard

### [Iterating over the arguments object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of#iterating_over_the_arguments_object)

You can iterate over the [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments) object to examine all parameters passed into a function.

function foo() {

for (const value of arguments) {

console.log(value);

}

}

foo(1, 2, 3);

// 1

// 2

// 3

Copy to Clipboard

### [Iterating over a NodeList](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of#iterating_over_a_nodelist)

The following example adds a read class to paragraphs that are direct descendants of the [<article>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/article) element by iterating over a [NodeList](https://developer.mozilla.org/en-US/docs/Web/API/NodeList) DOM collection.

const articleParagraphs = document.querySelectorAll("article > p");

for (const paragraph of articleParagraphs) {

paragraph.classList.add("read");

}

Copy to Clipboard

### [Iterating over a user-defined iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of#iterating_over_a_user-defined_iterable)

Iterating over an object with an @@iterator method that returns a custom iterator:

const iterable = {

[Symbol.iterator]() {

let i = 1;

return {

next() {

if (i <= 3) {

return { value: i++, done: false };

}

return { value: undefined, done: true };

},

};

},

};

for (const value of iterable) {

console.log(value);

}

// 1

// 2

// 3

Copy to Clipboard

Iterating over an object with an @@iterator generator method:

const iterable = {

\*[Symbol.iterator]() {

yield 1;

yield 2;

yield 3;

},

};

for (const value of iterable) {

console.log(value);

}

// 1

// 2

// 3

Copy to Clipboard

Iterable iterators (iterators with a [@@iterator]() method that returns this) are a fairly common technique to make iterators usable in syntaxes expecting iterables, such as for...of.

let i = 1;

const iterator = {

next() {

if (i <= 3) {

return { value: i++, done: false };

}

return { value: undefined, done: true };

},

[Symbol.iterator]() {

return this;

},

};

for (const value of iterator) {

console.log(value);

}

// 1

// 2

// 3

Copy to Clipboard

### [Iterating over a generator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of#iterating_over_a_generator)

function\* source() {

yield 1;

yield 2;

yield 3;

}

const generator = source();

for (const value of generator) {

console.log(value);

}

// 1

// 2

// 3

Copy to Clipboard

### [Early exiting](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of#early_exiting)

Execution of the break statement in the first loop causes it to exit early. The iterator is not finished yet, so the second loop will continue from where the first one stopped at.

const source = [1, 2, 3];

const iterator = source[Symbol.iterator]();

for (const value of iterator) {

console.log(value);

if (value === 1) {

break;

}

console.log("This string will not be logged.");

}

// 1

// Another loop using the same iterator

// picks up where the last loop left off.

for (const value of iterator) {

console.log(value);

}

// 2

// 3

// The iterator is used up.

// This loop will execute no iterations.

for (const value of iterator) {

console.log(value);

}

// [No output]

Copy to Clipboard

Generators implement the [return()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator/return) method, which causes the generator function to early return when the loop exits. This makes generators not reusable between loops.

function\* source() {

yield 1;

yield 2;

yield 3;

}

const generator = source();

for (const value of generator) {

console.log(value);

if (value === 1) {

break;

}

console.log("This string will not be logged.");

}

// 1

// The generator is used up.

// This loop will execute no iterations.

for (const value of generator) {

console.log(value);

}

// [No output]

### [Difference between for...of and for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of#difference_between_for...of_and_for...in)

Both for...in and for...of statements iterate over something. The main difference between them is in what they iterate over.

The [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) statement iterates over the [enumerable string properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Enumerability_and_ownership_of_properties) of an object, while the for...of statement iterates over values that the [iterable object](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols#the_iterable_protocol) defines to be iterated over.

The following example shows the difference between a for...of loop and a for...in loop when used with an [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array).

Object.prototype.objCustom = function () {};

Array.prototype.arrCustom = function () {};

const iterable = [3, 5, 7];

iterable.foo = "hello";

for (const i in iterable) {

console.log(i);

}

// "0", "1", "2", "foo", "arrCustom", "objCustom"

for (const i in iterable) {

if (Object.hasOwn(iterable, i)) {

console.log(i);

}

}

// "0" "1" "2" "foo"

for (const i of iterable) {

console.log(i);

}

// 3 5 7

Copy to Clipboard

The object iterable inherits the properties objCustom and arrCustom because it contains both Object.prototype and Array.prototype in its [prototype chain](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain).

The for...in loop logs only [enumerable properties](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Enumerability_and_ownership_of_properties) of the iterable object. It doesn't log array elements 3, 5, 7 or "hello" because those are not properties — they are values. It logs array indexes as well as arrCustom and objCustom, which are actual properties. If you're not sure why these properties are iterated over, there's a more thorough explanation of how [array iteration and for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in#array_iteration_and_for...in) work.

The second loop is similar to the first one, but it uses [Object.hasOwn()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwn) to check if the found enumerable property is the object's own, i.e. not inherited. If it is, the property is logged. Properties 0, 1, 2 and foo are logged because they are own properties. Properties arrCustom and objCustom are not logged because they are inherited.

The for...of loop iterates and logs values that iterable, as an array (which is [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/@@iterator)), defines to be iterated over. The object's elements 3, 5, 7 are shown, but none of the object's properties are.