map()

The **map()** method of [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) instances creates a new array populated with the results of calling a provided function on every element in the calling array.

const array1 = [1, 4, 9, 16];

// Pass a function to map

const map1 = array1.map((x) => x \* 2);

console.log(map1);

// Expected output: Array [2, 8, 18, 32]

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

JSCopy to Clipboard

map(callbackFn)

map(callbackFn, thisArg)

[**Parameters**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map#parameters)

callbackFn

A function to execute for each element in the array. Its return value is added as a single element in the new array. The function is called with the following arguments:

element

The current element being processed in the array.

index

The index of the current element being processed in the array.

array

The array map() was called upon.

thisArg Optional

A value to use as this when executing callbackFn. See [iterative methods](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#iterative_methods).

[**Return value**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map#return_value)

A new array with each element being the result of the callback function.

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

The map() method is an [iterative method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#iterative_methods). It calls a provided callbackFn function once for each element in an array and constructs a new array from the results.

callbackFn is invoked only for array indexes which have assigned values. It is not invoked for empty slots in [sparse arrays](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections#sparse_arrays).

The map() method is a [copying method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#copying_methods_and_mutating_methods). It does not alter this. However, the function provided as callbackFn can mutate the array. Note, however, that the length of the array is saved *before* the first invocation of callbackFn. Therefore:

* callbackFn will not visit any elements added beyond the array's initial length when the call to map() began.
* Changes to already-visited indexes do not cause callbackFn to be invoked on them again.
* If an existing, yet-unvisited element of the array is changed by callbackFn, its value passed to the callbackFn will be the value at the time that element gets visited. [Deleted](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete) elements are not visited.

The map() method is [generic](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#generic_array_methods). It only expects the this value to have a length property and integer-keyed properties.

Since map builds a new array, calling it without using the returned array is an anti-pattern; use [forEach](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach) or [for...of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of) instead.

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

[**Mapping an array of numbers to an array of square roots**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map#mapping_an_array_of_numbers_to_an_array_of_square_roots)

The following code takes an array of numbers and creates a new array containing the square roots of the numbers in the first array.

JS

const numbers = [1, 4, 9];

const roots = numbers.map((num) => Math.sqrt(num));

// roots is now [1, 2, 3]

// numbers is still [1, 4, 9]

[**Using map to reformat objects in an array**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map#using_map_to_reformat_objects_in_an_array)

The following code takes an array of objects and creates a new array containing the newly reformatted objects.

JS

const kvArray = [

{ key: 1, value: 10 },

{ key: 2, value: 20 },

{ key: 3, value: 30 },

];

const reformattedArray = kvArray.map(({ key, value }) => ({ [key]: value }));

console.log(reformattedArray); // [{ 1: 10 }, { 2: 20 }, { 3: 30 }]

console.log(kvArray);

// [

// { key: 1, value: 10 },

// { key: 2, value: 20 },

// { key: 3, value: 30 }

// ]

[**Mapping an array of numbers using a function containing an argument**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map#mapping_an_array_of_numbers_using_a_function_containing_an_argument)

The following code shows how map works when a function requiring one argument is used with it. The argument will automatically be assigned from each element of the array as map loops through the original array.

JS

const numbers = [1, 4, 9];

const doubles = numbers.map((num) => num \* 2);

console.log(doubles); // [2, 8, 18]

console.log(numbers); // [1, 4, 9]

[**Side-effectful mapping**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map#side-effectful_mapping)

The callback can have side effects.

JS

const cart = [5, 15, 25];

let total = 0;

const withTax = cart.map((cost) => {

total += cost;

return cost \* 1.2;

});

console.log(withTax); // [6, 18, 30]

console.log(total); // 45

This is not recommended, because copying methods are best used with pure functions. In this case, we can choose to iterate the array twice.

JS

const cart = [5, 15, 25];

const total = cart.reduce((acc, cost) => acc + cost, 0);

const withTax = cart.map((cost) => cost \* 1.2);

Sometimes this pattern goes to its extreme and the *only* useful thing that map() does is causing side effects.

JSCopy to Clipboard

const products = [

{ name: "sports car" },

{ name: "laptop" },

{ name: "phone" },

];

products.map((product) => {

product.price = 100;

});

As mentioned previously, this is an anti-pattern. If you don't use the return value of map(), use forEach() or a for...of loop instead.

JS

products.forEach((product) => {

product.price = 100;

});

Or, if you want to create a new array instead:

JSCopy to Clipboard

const productsWithPrice = products.map((product) => {

return { ...product, price: 100 };

});

[**Calling map() on non-array objects**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map#calling_map_on_non-array_objects)

The map() method reads the length property of this and then accesses each property whose key is a nonnegative integer less than length.

JS

const arrayLike = {

length: 3,

0: 2,

1: 3,

2: 4,

3: 5, // ignored by map() since length is 3

};

console.log(Array.prototype.map.call(arrayLike, (x) => x \*\* 2));

// [ 4, 9, 16 ]

[**Using map() generically on a NodeList**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map#using_map_generically_on_a_nodelist)

This example shows how to iterate through a collection of objects collected by querySelectorAll. This is because querySelectorAll returns a NodeList (which is a collection of objects).

In this case, we return all the selected options' values on the screen:

JSCopy to Clipboard

const elems = document.querySelectorAll("select option:checked");

const values = Array.prototype.map.call(elems, ({ value }) => value);

An easier way would be the [Array.from()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from) method.

[**Using map() on sparse arrays**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map#using_map_on_sparse_arrays)

A sparse array remains sparse after map(). The indices of empty slots are still empty in the returned array, and the callback function won't be called on them.

JS

console.log(

[1, , 3].map((x, index) => {

console.log(`Visit ${index}`);

return x \* 2;

}),

);

// Visit 0

// Visit 2

// [2, empty, 6]

[**Using parseInt() with map()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map#using_parseint_with_map)

([inspired by this blog post](https://wirfs-brock.com/allen/posts/166))

It is common to use the callback with one argument (the element being traversed). Certain functions are also commonly used with one argument, even though they take additional optional arguments. These habits may lead to confusing behaviors.

Consider:

JS

["1", "2", "3"].map(parseInt);

While one might expect [1, 2, 3], the actual result is [1, NaN, NaN].

[parseInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt) is often used with one argument, but takes two. The first is an expression and the second is the radix to the callback function, Array.prototype.map passes 3 arguments:

* the element
* the index
* the array

The third argument is ignored by [parseInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt)—but *not* the second one! This is the source of possible confusion.

Here is a concise example of the iteration steps:

JS

// parseInt(string, radix) -> map(parseInt(value, index))

/\* first iteration (index is 0): \*/ parseInt("1", 0); // 1

/\* second iteration (index is 1): \*/ parseInt("2", 1); // NaN

/\* third iteration (index is 2): \*/ parseInt("3", 2); // NaN

Then let's talk about solutions.

JS

const returnInt = (element) => parseInt(element, 10);

["1", "2", "3"].map(returnInt); // [1, 2, 3]

// Actual result is an array of numbers (as expected)

// Same as above, but using the concise arrow function syntax

["1", "2", "3"].map((str) => parseInt(str)); // [1, 2, 3]

// A simpler way to achieve the above, while avoiding the "gotcha":

["1", "2", "3"].map(Number); // [1, 2, 3]

// But unlike parseInt(), Number() will also return a float or (resolved) exponential notation:

["1.1", "2.2e2", "3e300"].map(Number); // [1.1, 220, 3e+300]

// For comparison, if we use parseInt() on the array above:

["1.1", "2.2e2", "3e300"].map((str) => parseInt(str)); // [1, 2, 3]

One alternative output of the map method being called with [parseInt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/parseInt) as a parameter runs as follows:

JS

const strings = ["10", "10", "10"];

const numbers = strings.map(parseInt);

console.log(numbers);

// Actual result of [10, NaN, 2] may be unexpected based on the above description.

[**Mapped array contains undefined**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map#mapped_array_contains_undefined)

When [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined) or nothing is returned:

JS

const numbers = [1, 2, 3, 4];

const filteredNumbers = numbers.map((num, index) => {

if (index < 3) {

return num;

}

});

// index goes from 0, so the filterNumbers are 1,2,3 and undefined.

// filteredNumbers is [1, 2, 3, undefined]

// numbers is still [1, 2, 3, 4]