mdn web docs

Array.prototype.map()

Baseline Widely available

The `map()` method of `Array` instances creates a new array populated with the results of calling a provided function on every element in the calling array.

Try it

JavaScript Demo: Array.prototype.map()

```
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]

```

Run

Reset

Syntax

JS

```
map(callbackFn)
map(callbackFn, thisArg)
```

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.

Return value

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

Description

The `map()` method is an iterative method. It calls a provided `callbackFn` function once for each element in an array and constructs a new array from the results. Read the iterative methods section for more information about how these methods work in general.

`callbackFn` is invoked only for array indexes which have assigned values. It is not invoked for empty slots in sparse arrays.

The `map()` method is generic. 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` or `for...of` instead.

Examples

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

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 }
// ]
```

Using parseInt() with map()

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` 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, and the array. The third argument is ignored by `parseInt` — but *not* the second one! This is the source of possible confusion.

Here is a concise example of the iteration steps:

```JS
/* 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
```

To solve this, define another function that only takes one argument:

```JS
["1", "2", "3"].map((str) => parseInt(str, 10)); // [1, 2, 3]
```

You can also use the `Number` function, which only takes one argument:

```JS
```

```
["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, 10)); // [1, 2, 3]
```

See A JavaScript optional argument hazard by Allen Wirfs-Brock for more discussions.

Mapped array contains undefined

When `undefined` or nothing is returned, the resulting array contains `undefined`. If you want to delete the element instead, chain a `filter()` method, or use the `flatMap()` method and return an empty array to signify deletion.

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]
```

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.

```js
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:

```js
const productsWithPrice = products.map((product) => {
  return { ...product, price: 100 };
```

```
});
```

Using the third argument of callbackFn

The `array` argument is useful if you want to access another element in the array, especially when you don't have an existing variable that refers to the array. The following example first uses `filter()` to extract the positive values and then uses `map()` to create a new array where each element is the average of its neighbors and itself.

JS

```js
const numbers = [3, -1, 1, 4, 1, 5, 9, 2, 6];
const averaged = numbers
  .filter((num) => num > 0)
  .map((num, idx, arr) => {
    // Without the arr argument, there's no way to easily access the
    // intermediate array without saving it to a variable.
    const prev = arr[idx - 1];
    const next = arr[idx + 1];
    let count = 1;
    let total = num;
    if (prev !== undefined) {
      count++;
      total += prev;
    }
    if (next !== undefined) {
      count++;
      total += next;
    }
    const average = total / count;
    // Keep two decimal places
    return Math.round(average * 100) / 100;
  });
console.log(averaged); // [2, 2.67, 2, 3.33, 5, 5.33, 5.67, 4]
```

The `array` argument is *not* the array that is being built — there is no way to access the array being built from the callback function.

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]
```

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 ]
```

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 `option`s' values on the screen:

```JS
const elems = document.querySelectorAll("select option:checked");
const values = Array.prototype.map.call(elems, ({ value }) => value);
```

You can also use `Array.from()` to transform `elems` to an array, and then access the `map()` method.

Specifications

Specification
ECMAScript® 2026 Language Specification # sec-array.prototype.map

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
`map`	✓ 1	✓ 12	✓ 1.5	✓ 9.5	✓ 3	✓ 18	✓ 4	✓ 10.1	✓ 1	✓ 1	✓ 4.4	✓ 1	✓ 1	✓ 0.10

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

✓ Full support

See also

- Polyfill of `Array.prototype.map` in `core-js`
- es-shims polyfill of `Array.prototype.map`
- Indexed collections guide
- `Array`
- `Array.prototype.forEach()`
- `Array.from()`
- `TypedArray.prototype.map()`

- `Map`

Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on Mar 14, 2025 by MDN contributors.