map() method

The JavaScript map() method in JavaScript creates an array by calling a specific function on each element present in the parent array. It is a non-mutating method. Generally, the map() method is used to iterate over an array and calling function on every element of the array.

**Syntax**:

// Arrow function

map((element) => { /\* … \*/ })

map((element, index) => { /\* … \*/ })

map((element, index, array) => { /\* … \*/ })

// Callback function

map(callbackFn)

map(callbackFn, thisArg)

// Inline callback function

map(function (element) { /\* … \*/ })

map(function (element, index) { /\* … \*/ })

map(function (element, index, array) { /\* … \*/ })

map(function (element, index, array) { /\* … \*/ }, thisArg)

**Parameters: This method accepts two parameters as mentioned above and described below:**

1. function(currentValue, index, arr): It is a required parameter and it runs on each element of the array. It contains three parameters which are listed below:
2. currentValue/callbackfn: It is a required parameter and it holds the value of the current element.
3. index: It is an optional parameter and it holds the index of the current element.
4. arr: It is an optional parameter and it holds the array.
5. thisValue: It is an optional parameter and is used to hold the value passed to the function.
6. Return Value: It returns a new array and elements of arrays are the result of the callback function.

The map() method 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]

The map() method creates another array form an array

**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.

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 a copying method. 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 elements are not visited.

**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 }

// ]

**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

Copy to Clipboard

const numbers = [1, 4, 9];

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

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

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

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**

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:

JS

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

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

An easier way would be the Array.from() method.

**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()

(inspired by this blog post)

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

Copy to Clipboard

["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:

1. the element
2. the index
3. 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

Copy to Clipboard

// 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

Copy to Clipboard

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 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

When 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]