reduce()

The **reduce()** method of [Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array) instances executes a user-supplied "reducer" callback function on each element of the array, in order, passing in the return value from the calculation on the preceding element. The final result of running the reducer across all elements of the array is a single value.

The first time that the callback is run there is no "return value of the previous calculation". If supplied, an initial value may be used in its place. Otherwise the array element at index 0 is used as the initial value and iteration starts from the next element (index 1 instead of index 0).

Perhaps the easiest-to-understand case for reduce() is to return the sum of all the elements in an array:

const array1 = [1, 2, 3, 4];

// 0 + 1 + 2 + 3 + 4

const initialValue = 0;

const sumWithInitial = array1.reduce((accumulator, currentValue) => accumulator + currentValue, initialValue);

console.log(sumWithInitial);

// Expected output: 10

The reducer walks through the array element-by-element, at each step adding the current array value to the result from the previous step (this result is the running sum of all the previous steps) — until there are no more elements to add.

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

JSCopy to Clipboard

reduce(callbackFn)

reduce(callbackFn, initialValue)

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

callbackFn

A function to execute for each element in the array. Its return value becomes the value of the accumulator parameter on the next invocation of callbackFn. For the last invocation, the return value becomes the return value of reduce(). The function is called with the following arguments:

accumulator

The value resulting from the previous call to callbackFn. On the first call, its value is initialValue if the latter is specified; otherwise its value is array[0].

currentValue

The value of the current element. On the first call, its value is array[0] if initialValue is specified; otherwise its value is array[1].

currentIndex

The index position of currentValue in the typed array. On the first call, its value is 0 if initialValue is specified, otherwise 1.

array

The array reduce() was called upon.

initialValue Optional

A value to which accumulator is initialized the first time the callback is called. If initialValue is specified, callbackFn starts executing with the first value in the array as currentValue. If initialValue is *not* specified, accumulator is initialized to the first value in the array, and callbackFn starts executing with the second value in the array as currentValue. In this case, if the array is empty (so that there's no first value to return as accumulator), an error is thrown.

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

The value that results from running the "reducer" callback function to completion over the entire array.

[**Exceptions**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce#exceptions)

[TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError)

Thrown if the array contains no elements and initialValue is not provided.

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

The reduce() method is an [iterative method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#iterative_methods). It runs a "reducer" callback function over all elements in the array, in ascending-index order, and accumulates them into a single value. Every time, the return value of callbackFn is passed into callbackFn again on next invocation as accumulator. The final value of accumulator (which is the value returned from callbackFn on the final iteration of the array) becomes the return value of reduce().

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).

Unlike other [iterative methods](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array#iterative_methods), reduce() does not accept a thisArg argument. callbackFn is always called with undefined as this, which gets substituted with globalThis if callbackFn is non-strict.

reduce() is a central concept in [functional programming](https://en.wikipedia.org/wiki/Functional_programming), where it's not possible to mutate any value, so in order to accumulate all values in an array, one must return a new accumulator value on every iteration. This convention propagates to JavaScript's reduce(): you should use [spreading](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax) or other copying methods where possible to create new arrays and objects as the accumulator, rather than mutating the existing one. If you decided to mutate the accumulator instead of copying it, remember to still return the modified object in the callback, or the next iteration will receive undefined.

reduce() does not mutate the array on which it is called, but the function provided as callbackFn can. 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 reduce() 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.

Examples

How reduce() works without an initial value

The code below shows what happens if we call reduce() with an array and no initial value.

JS

Copy to Clipboard

const array = [15, 16, 17, 18, 19];

function reducer(accumulator, currentValue, index) {

const returns = accumulator + currentValue;

console.log(

`accumulator: ${accumulator}, currentValue: ${currentValue}, index: ${index}, returns: ${returns}`,

);

return returns;

}

array.reduce(reducer);

The callback would be invoked four times, with the arguments and return values in each call being as follows:

accumulator currentValue index Return value

First call 15 16 1 31

Second call 31 17 2 48

Third call 48 18 3 66

Fourth call 66 19 4 85

The array parameter never changes through the process — it's always [15, 16, 17, 18, 19]. The value returned by reduce() would be that of the last callback invocation (85).

How reduce() works with an initial value

Here we reduce the same array using the same algorithm, but with an initialValue of 10 passed as the second argument to reduce():

JS

Copy to Clipboard

[15, 16, 17, 18, 19].reduce(

(accumulator, currentValue) => accumulator + currentValue,

10,

);

The callback would be invoked five times, with the arguments and return values in each call being as follows:

accumulator currentValue index Return value

First call 10 15 0 25

Second call 25 16 1 41

Third call 41 17 2 58

Fourth call 58 18 3 76

Fifth call 76 19 4 95

The value returned by reduce() in this case would be 95.

Sum of values in an object array

To sum up the values contained in an array of objects, you must supply an initialValue, so that each item passes through your function.

JS

Copy to Clipboard

const objects = [{ x: 1 }, { x: 2 }, { x: 3 }];

const sum = objects.reduce(

(accumulator, currentValue) => accumulator + currentValue.x,

0,

);

console.log(sum); // 6

Function sequential piping

The pipe function takes a sequence of functions and returns a new function. When the new function is called with an argument, the sequence of functions are called in order, which each one receiving the return value of the previous function.

JS

Copy to Clipboard

const pipe =

(...functions) =>

(initialValue) =>

functions.reduce((acc, fn) => fn(acc), initialValue);

// Building blocks to use for composition

const double = (x) => 2 \* x;

const triple = (x) => 3 \* x;

const quadruple = (x) => 4 \* x;

// Composed functions for multiplication of specific values

const multiply6 = pipe(double, triple);

const multiply9 = pipe(triple, triple);

const multiply16 = pipe(quadruple, quadruple);

const multiply24 = pipe(double, triple, quadruple);

// Usage

multiply6(6); // 36

multiply9(9); // 81

multiply16(16); // 256

multiply24(10); // 240

Running promises in sequence

Promise sequencing is essentially function piping demonstrated in the previous section, except done asynchronously.

JS

Copy to Clipboard

// Compare this with pipe: fn(acc) is changed to acc.then(fn),

// and initialValue is ensured to be a promise

const asyncPipe =

(...functions) =>

(initialValue) =>

functions.reduce((acc, fn) => acc.then(fn), Promise.resolve(initialValue));

// Building blocks to use for composition

const p1 = async (a) => a \* 5;

const p2 = async (a) => a \* 2;

// The composed functions can also return non-promises, because the values are

// all eventually wrapped in promises

const f3 = (a) => a \* 3;

const p4 = async (a) => a \* 4;

asyncPipe(p1, p2, f3, p4)(10).then(console.log); // 1200

asyncPipe can also be implemented using async/await, which better demonstrates its similarity with pipe:

JS

Copy to Clipboard

const asyncPipe =

(...functions) =>

(initialValue) =>

functions.reduce(async (acc, fn) => fn(await acc), initialValue);

Using reduce() with sparse arrays

reduce() skips missing elements in sparse arrays, but it does not skip undefined values.

JS

Copy to Clipboard

console.log([1, 2, , 4].reduce((a, b) => a + b)); // 7

console.log([1, 2, undefined, 4].reduce((a, b) => a + b)); // NaN

Calling reduce() on non-array objects

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

JS

Copy to Clipboard

const arrayLike = {

length: 3,

0: 2,

1: 3,

2: 4,

3: 99, // ignored by reduce() since length is 3

};

console.log(Array.prototype.reduce.call(arrayLike, (x, y) => x + y));

// 9

When to not use reduce()

Multipurpose higher-order functions like reduce() can be powerful but sometimes difficult to understand, especially for less-experienced JavaScript developers. If code becomes clearer when using other array methods, developers must weigh the readability tradeoff against the other benefits of using reduce().

Note that reduce() is always equivalent to a for...of loop, except that instead of mutating a variable in the upper scope, we now return the new value for each iteration:

JS

Copy to Clipboard

const val = array.reduce((acc, cur) => update(acc, cur), initialValue);

// Is equivalent to:

let val = initialValue;

for (const cur of array) {

val = update(val, cur);

}

As previously stated, the reason why people may want to use reduce() is to mimic functional programming practices of immutable data. Therefore, developers who uphold the immutability of the accumulator often copy the entire accumulator for each iteration, like this:

JS

Copy to Clipboard

const names = ["Alice", "Bob", "Tiff", "Bruce", "Alice"];

const countedNames = names.reduce((allNames, name) => {

const currCount = Object.hasOwn(allNames, name) ? allNames[name] : 0;

return {

...allNames,

[name]: currCount + 1,

};

}, {});

This code is ill-performing, because each iteration has to copy the entire allNames object, which could be big, depending how many unique names there are. This code has worst-case O(N^2) performance, where N is the length of names.

A better alternative is to mutate the allNames object on each iteration. However, if allNames gets mutated anyway, you may want to convert the reduce() to a simple for loop instead, which is much clearer:

JS

Copy to Clipboard

const names = ["Alice", "Bob", "Tiff", "Bruce", "Alice"];

const countedNames = names.reduce((allNames, name) => {

const currCount = allNames[name] ?? 0;

allNames[name] = currCount + 1;

// return allNames, otherwise the next iteration receives undefined

return allNames;

}, Object.create(null));

JS

Copy to Clipboard

const names = ["Alice", "Bob", "Tiff", "Bruce", "Alice"];

const countedNames = Object.create(null);

for (const name of names) {

const currCount = countedNames[name] ?? 0;

countedNames[name] = currCount + 1;

}

Therefore, if your accumulator is an array or an object and you are copying the array or object on each iteration, you may accidentally introduce quadratic complexity into your code, causing performance to quickly degrade on large data.

Some of the acceptable use cases of reduce() are given above (most notably, summing an array, promise sequencing, and function piping). There are other cases where better alternatives than reduce() exist.

Flattening an array of arrays. Use flat() instead.

JS

Copy to Clipboard

const flattened = array.reduce((acc, cur) => acc.concat(cur), []);

JS

Copy to Clipboard

const flattened = array.flat();

Grouping objects by a property. Use Object.groupBy() instead.

JS

Copy to Clipboard

const groups = array.reduce((acc, obj) => {

const key = obj.name;

const curGroup = acc[key] ?? [];

return { ...acc, [key]: [...curGroup, obj] };

}, {});

JS

Copy to Clipboard

const groups = Object.groupBy(array, (obj) => obj.name);

Concatenating arrays contained in an array of objects. Use flatMap() instead.

JS

Copy to Clipboard

const friends = [

{ name: "Anna", books: ["Bible", "Harry Potter"] },

{ name: "Bob", books: ["War and peace", "Romeo and Juliet"] },

{ name: "Alice", books: ["The Lord of the Rings", "The Shining"] },

];

const allBooks = friends.reduce((acc, cur) => [...acc, ...cur.books], []);

JS

Copy to Clipboard

const allBooks = friends.flatMap((person) => person.books);

Removing duplicate items in an array. Use Set and Array.from() instead.

JS

Copy to Clipboard

const uniqArray = array.reduce(

(acc, cur) => (acc.includes(cur) ? acc : [...acc, cur]),

[],

);

JS

Copy to Clipboard

const uniqArray = Array.from(new Set(array));

Eliminating or adding elements in an array. Use flatMap() instead.

JS

Copy to Clipboard

// Takes an array of numbers and splits perfect squares into its square roots

const roots = array.reduce((acc, cur) => {

if (cur < 0) return acc;

const root = Math.sqrt(cur);

if (Number.isInteger(root)) return [...acc, root, root];

return [...acc, cur];

}, []);

JS

Copy to Clipboard

const roots = array.flatMap((val) => {

if (val < 0) return [];

const root = Math.sqrt(val);

if (Number.isInteger(root)) return [root, root];

return [val];

});

If you are only eliminating elements from an array, you also can use filter().

Searching for elements or testing if elements satisfy a condition. Use find() and findIndex(), or some() and every() instead. These methods have the additional benefit that they return as soon as the result is certain, without iterating the entire array.

JS

Copy to Clipboard

const allEven = array.reduce((acc, cur) => acc && cur % 2 === 0, true);

JS

Copy to Clipboard

const allEven = array.every((val) => val % 2 === 0);

In cases where reduce() is the best choice, documentation and semantic variable naming can help mitigate readability drawbacks.