Array.prototype.reduce()

Baseline Widely available ✓ ✓ ✓ ✓

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).

Try it

JavaScript Demo: Array.prototype.reduce()

```
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

```

Run

Reset

Syntax

JS

```
reduce(callbackFn)
reduce(callbackFn, initialValue)
```

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 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

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

Exceptions

`TypeError`

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

Description

The `reduce()` method is an iterative method. 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()`. 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.

Unlike other 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, 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 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. However, note that copying the accumulator may in turn lead to increased memory usage and degraded performance — see When to not use reduce() for more details. In such cases, to avoid bad performance and unreadable code, it's better to use a `for` loop instead.

The `reduce()` method is generic. It only expects the `this` value to have a `length` property and integer-keyed properties.

Edge cases

If the array only has one element (regardless of position) and no `initialValue` is provided, or if `initialValue` is provided but the array is empty, the solo value will be returned *without* calling `callbackFn`.

If `initialValue` is provided and the array is not empty, then the reduce method will always invoke the callback function starting at index 0.

If `initialValue` is not provided then the reduce method will act differently for arrays with length larger than 1, equal to 1 and 0, as shown in the following example:

JS

```js
const getMax = (a, b) => Math.max(a, b);

// callback is invoked for each element in the array starting at index 0
[1, 100].reduce(getMax, 50); // 100
[50].reduce(getMax, 10); // 50

// callback is invoked once for element at index 1
[1, 100].reduce(getMax); // 100

// callback is not invoked
[50].reduce(getMax); // 50
[].reduce(getMax, 1); // 1
```

```
[].reduce(getMax); // TypeError
```

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

```
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

```js
[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

```js
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

```
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](https://developer.mozilla.org) is essentially function piping demonstrated in the previous section, except done asynchronously.

JS

```
// 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));
```

```js
// 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

```js
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

```js
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

```js
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

```js
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

```js
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 `for` loop instead, which is much clearer:

JS

```js
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

```js
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. This has happened in real-world code — see for example Making Tanstack Table 1000x faster with a 1 line change.

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

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

  JS

  ```js
  const flattened = array.flat();
  ```
  ✓

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

  JS

  ```js
  const groups = array.reduce((acc, obj) => {
    const key = obj.name;
    const curGroup = acc[key] ?? [];
    return { ...acc, [key]: [...curGroup, obj] };
  }, {});
  ```
  ✗

  JS

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

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

  JS

  ```js
  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 ✓

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

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

  JS ✗

  ```js
  const uniqArray = array.reduce(
    (acc, cur) => (acc.includes(cur) ? acc : [...acc, cur]),
    [],
  );
  ```

  JS ✓

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

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

  JS ✗

  ```js
  // 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 ✓

  ```js
  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

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

JS

```js
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.

Specifications

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

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
`reduce`	✓ 3	✓ 12	✓ 3	✓ 10.5	✓ 4	✓ 18	✓ 4	✓ 14	✓ 3.2	✓ 1	✓ 4.4	✓ 3.2	✓ 1	✓ 0.10

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

✓ Full support

See also

- Polyfill of `Array.prototype.reduce` in `core-js`
- es-shims polyfill of `Array.prototype.reduce`
- Indexed collections guide
- `Array`
- `Array.prototype.map()`
- `Array.prototype.flat()`
- `Array.prototype.flatMap()`
- `Array.prototype.reduceRight()`
- `TypedArray.prototype.reduce()`
- `Object.groupBy()`
- `Map.groupBy()`

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.