Array.prototype.reduce() - JavaScript | MDN

The `reduce()` method 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:

Try it

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

```
// Arrow function
reduce((previousValue, currentValue) => { /* … */ } )
reduce((previousValue, currentValue, currentIndex) => {
```

```
/* … */ } )
reduce((previousValue, currentValue, currentIndex,
array) => { /* … */ } )

reduce((previousValue, currentValue) => { /* … */ } ,
initialValue)
reduce((previousValue, currentValue, currentIndex) => {
/* … */ } , initialValue)
reduce((previousValue, currentValue, currentIndex,
array) => { /* … */ }, initialValue)

// Callback function
reduce(callbackFn)
reduce(callbackFn, initialValue)

// Inline callback function
reduce(function(previousValue, currentValue) { /* … */
})
reduce(function(previousValue, currentValue,
currentIndex) { /* … */ })
reduce(function(previousValue, currentValue,
currentIndex, array) { /* … */ })

reduce(function(previousValue, currentValue) { /* … */
}, initialValue)
reduce(function(previousValue, currentValue,
currentIndex) { /* … */ }, initialValue)
reduce(function(previousValue, currentValue,
currentIndex, array) { /* … */ }, initialValue)
```

Copy to Clipboard

Parameters

`callbackFn`

`previousValue`

The value resulting from the previous call to `callbackFn`. On first call, `initialValue` if specified, otherwise the value of `array[0]`.

`currentValue`

The value of the current element. On first call, the value of `array[0]` if an `initialValue` was specified, otherwise the value of `array[1]`.

`currentIndex`

The index position of `currentValue` in the array. On first call, `0` if `initialValue` was specified, otherwise `1`.

`array`

The array being traversed.

`initialValue` Optional

A value to which `previousValue` is initialized the first time the callback is called. If `initialValue` is specified, that also causes `currentValue` to be initialized to the first value in the array. If `initialValue` is *not* specified, `previousValue` is initialized to the first value in the array, and `currentValue` is initialized to the second value in the array.

Description

The `reduce()` method takes two arguments: a callback function and an optional initial value. If an initial value is provided, `reduce()` calls the "reducer" callback function on each element in the array, in order. If

no initial value is provided, `reduce()` calls the callback function on each element in the array after the first element.

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

`reduce()` returns the value that is returned from the callback function on the final iteration of the array.

`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.

When to not use reduce()

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

Behavior during array mutations

The `reduce()` method itself does not mutate the array it is used on. However, it is possible for code inside the callback function to mutate the array. These are the possible scenarios of array mutations and how `reduce()` behaves in these scenarios:

- If elements are appended to the array *after* `reduce()` begins to iterate over the array, the callback function does not iterate over the appended elements.
- If existing elements of the array do get changed, the values passed to the callback function will be the values from the time that reduce() was first called on the array.
- Array elements that are deleted *after* the call to `reduce()` begins *and* before being iterated over are not visited by `reduce()`.

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:

```
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
```

Copy to Clipboard

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

How reduce() works without an initial value

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

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

function reducer(previousValue, currentValue, index) {
  const returns = previousValue + currentValue;
  console.log(
    `previousValue: ${previousValue}, currentValue:
${currentValue}, index: ${index}, returns: ${returns}`,
  );
  return returns;
}
```

```
array.reduce(reducer);
```

Copy to Clipboard

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

	`previousValue`	`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 the second argument to `reduce()`:

```
[15, 16, 17, 18, 19].reduce(
  (previousValue, currentValue) => previousValue +
currentValue,
  10,
);
```

Copy to Clipboard

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

	`previousValue`	`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.

```
const objects = [{ x: 1 }, { x: 2 }, { x: 3 }];
const sum = objects.reduce(
  (previousValue, currentValue) => previousValue +
currentValue.x,
  0,
);
```

```
console.log(sum); // logs 6
```

Copy to Clipboard

Flatten an array of arrays

```
const flattened = [
  [0, 1],
  [2, 3],
  [4, 5],
].reduce(
  (previousValue, currentValue) =>
previousValue.concat(currentValue),
  [],
);
// flattened is [0, 1, 2, 3, 4, 5]
```

Copy to Clipboard

Counting instances of values in an object

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

const countedNames = names.reduce((allNames, name) => {
  const currCount = allNames[name] ?? 0;
  return {
    ...allNames,
    [name]: currCount + 1,
  };
}, {});
```

```
// countedNames is:
// { 'Alice': 2, 'Bob': 1, 'Tiff': 1, 'Bruce': 1 }
```

Copy to Clipboard

Grouping objects by a property

```
const people = [
  { name: "Alice", age: 21 },
  { name: "Max", age: 20 },
  { name: "Jane", age: 20 },
];

function groupBy(objectArray, property) {
  return objectArray.reduce((acc, obj) => {
    const key = obj[property];
    const curGroup = acc[key] ?? [];

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

const groupedPeople = groupBy(people, "age");
// groupedPeople is:
// {
//   20: [
//     { name: 'Max', age: 20 },
//     { name: 'Jane', age: 20 }
//   ],
//   21: [{ name: 'Alice', age: 21 }]
// }
```

Copy to Clipboard

Concatenating arrays contained in an array of objects using the spread syntax and initialValue

```
// friends - an array of objects
// where object field "books" is a list of favorite
books
const friends = [
  {
    name: "Anna",
    books: ["Bible", "Harry Potter"],
    age: 21,
  },
  {
    name: "Bob",
    books: ["War and peace", "Romeo and Juliet"],
    age: 26,
  },
  {
    name: "Alice",
    books: ["The Lord of the Rings", "The Shining"],
    age: 18,
  },
];

// allbooks - list which will contain all friends'
books +
// additional list contained in initialValue
const allbooks = friends.reduce(
  (previousValue, currentValue) => [...previousValue,
...currentValue.books],
```

```
  ["Alphabet"],
);

// allbooks = [
//   'Alphabet', 'Bible', 'Harry Potter', 'War and
peace',
//   'Romeo and Juliet', 'The Lord of the Rings',
//   'The Shining'
// ]
```

Copy to Clipboard

Remove duplicate items in an array

Note: The same effect can be achieved with `Set` and `Array.from()` as `const arrayWithNoDuplicates = Array.from(new Set(myArray))` with better performance.

```
const myArray = ["a", "b", "a", "b", "c", "e", "e",
"c", "d", "d", "d", "d"];
const myArrayWithNoDuplicates = myArray.reduce(
  (previousValue, currentValue) => {
    if (!previousValue.includes(currentValue)) {
      return [...previousValue, currentValue];
    }
    return previousValue;
  },
  [],
);

console.log(myArrayWithNoDuplicates);
```

Copy to Clipboard

Replace .filter().map() with .reduce()

Using `filter()` then `map()` traverses the array twice, but you can achieve the same effect while traversing only once with `reduce()`, thereby being more efficient. (If you like `for` loops, you can filter and map while traversing once with `forEach()`.)

```
const numbers = [-5, 6, 2, 0];

const doubledPositiveNumbers =
numbers.reduce((previousValue, currentValue) => {
  if (currentValue > 0) {
    const doubled = currentValue * 2;
    return [...previousValue, doubled];
  }
  return previousValue;
}, []);

console.log(doubledPositiveNumbers); // [12, 4]
```

Copy to Clipboard

Running Promises in Sequence

```
/**
 * Chain a series of promise handlers.
 *
 * @param {array} arr - A list of promise handlers,
each one receiving the
 * resolved result of the previous handler and
```

```
returning another promise.
 * @param {*} input The initial value to start the
promise chain
 * @return {Object} Final promise with a chain of
handlers attached
 */
function runPromiseInSequence(arr, input) {
  return arr.reduce(
    (promiseChain, currentFunction) =>
promiseChain.then(currentFunction),
    Promise.resolve(input),
  );
}

// promise function 1
function p1(a) {
  return new Promise((resolve, reject) => {
    resolve(a * 5);
  });
}

// promise function 2
function p2(a) {
  return new Promise((resolve, reject) => {
    resolve(a * 2);
  });
}

// function 3 - will be wrapped in a resolved promise
by .then()
function f3(a) {
```

```
  return a * 3;
}

// promise function 4
function p4(a) {
  return new Promise((resolve, reject) => {
    resolve(a * 4);
  });
}

const promiseArr = [p1, p2, f3, p4];
runPromiseInSequence(promiseArr, 10).then(console.log);
// 1200
```

Copy to Clipboard

Function composition enabling piping

```
// Building-blocks to use for composition
const double = (x) => 2 * x;
const triple = (x) => 3 * x;
const quadruple = (x) => 4 * x;

// Function composition enabling pipe functionality
const pipe =
  (...functions) =>
  (initialValue) =>
    functions.reduce((acc, fn) => fn(acc),
initialValue);

// 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
```

Copy to Clipboard

Using reduce() with sparse arrays

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

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

Copy to Clipboard

Calling reduce() on non-array objects

The `reduce()` method reads the `length` property of `this` and then accesses each integer index.

```
const arrayLike = {
  length: 3,
  0: 2,
```

```
  1: 3,
  2: 4,
};
console.log(Array.prototype.reduce.call(arrayLike, (x,
y) => x + y));
// 9
```

Copy to Clipboard