Array.prototype.reduce()

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

JavaScript Demo: Array.reduce()

```
1  const array1 = [1, 2, 3, 4];
2
3  // 0 + 1 + 2 + 3 + 4
4  const initialValue = 0;
5  const sumWithInitial = array1.reduce(
6    (accumulator, currentValue) => accumulator + currentV
7    initialValue
8  );
9
10 console.log(sumWithInitial);
11 // expected output: 10
12
```

Run ›

Reset

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

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

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

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

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

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()`.

`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.

`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 elements are not visited.

> **Warning:** Concurrent modifications of the kind described above frequently lead to hard-to-understand code and are generally to be avoided (except in special cases).

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

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.

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
```

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.

```
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()`:

```
[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`

	`accumulator`	`currentValue`	`index`	Return value
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(
  (accumulator, currentValue) => accumulator + currentValue.x,
  0,
);

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

Flatten an array of arrays

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

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 }
```

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");
console.log(groupedPeople);
// {
//   20: [
//     { name: 'Max', age: 20 },
//     { name: 'Jane', age: 20 }
//   ],
//   21: [{ name: 'Alice', age: 21 }]
// }
```

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(
  (accumulator, currentValue) => [...accumulator, ...currentValue.books],
  ["Alphabet"],
);
console.log(allbooks);
// [
//   'Alphabet', 'Bible', 'Harry Potter', 'War and peace',
//   'Romeo and Juliet', 'The Lord of the Rings',
//   'The Shining'
// ]
```

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((accumulator, currentValue) => {
  if (!accumulator.includes(currentValue)) {
    return [...accumulator, currentValue];
  }
  return accumulator;
}, []);

console.log(myArrayWithNoDuplicates);
```

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((accumulator, currentValue) => {
  if (currentValue > 0) {
    const doubled = currentValue * 2;
    return [...accumulator, doubled];
  }
  return accumulator;
}, []);

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

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
```

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
```

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
```

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
```

Specifications

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

Browser compatibility

Report problems with this compatibility data on GitHub

	Desktop					Mobile			
	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS
`reduce`	✓ Chrome 3	✓ Edge 12	✓ Firefox 3	✓ Opera 10.5	✓ Safari 4	✓ Chrome 18 Android	✓ Firefox 4 for Android	✓ Opera 14 Android	✓ Safari 3.2 on iOS

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

✓ Full support

See also

- Polyfill of `Array.prototype.reduce` in `core-js`
- `Array.prototype.reduceRight()`

Last modified: Dec 13, 2022, by MDN contributors