NOVEMBER 29, 2023 / #ARRAYS

How to Use JavaScript's Array reduce() Method – Explained with Examples

Nathan Sebhastian

Hello again, friends! The `reduce()` method is one of the most confusing Array methods in JavaScript. So in this

I'm also going to show some examples that'll help you understand how and where you may want to use the method. Don't worry! `reduce()` is actually simple once you understand how it works.

How the `reduce()` Method Works

The `reduce()` method got its name from the functionality it provides, which is to iterate and "reduce" an array's values into one value.

The easiest way to understand how the `reduce()` method works is through an example, so let's see an easy one first.

Suppose you have an array of objects that contain a `name` and `price` property as follows:

```
const items = [
  { name: 'Apple', price: 1 },
  { name: 'Orange', price: 2 },
  { name: 'Mango', price: 3 },
];
```

Next, you are tasked to get the sum of the `price` property and get the total price. You can do this using the `forEach()` method as follows:

```
const items = [
  { name: 'Apple', price: 1 },
  { name: 'Orange', price: 2 },
```

```
items.forEach(item => {
  totalPrice += item.price;
})

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

First, we declare the `totalPrice` variable and assign `0` as its value. Next, we called the `forEach()` method to iterate over the `items` array, adding the price of each item to the `totalPrice`.

The code above works, but since we're looking to get a single value from the array, the `reduce()` method would be more suited for the task.

The `reduce()` method works in a similar manner to the `forEach()` method, with the added ability to collect the result of each iteration as a single value.

Let's try to get the total price using the `reduce()` method. First, you need to call the `reduce()` method and pass two parameters to the callback function: `accumulator` and `item`.

```
const totalPrice = items.reduce((accumulator ,item) => {
  return accumulator += item.price;
}, 0)
```

The `accumulator` parameter is the single value that will be returned by the `reduce()` method. It will contain the value returned by the

change in each iteration just like in the `forEach()` method.

In the first iteration, the `accumulator` parameter will contain whatever value you passed as the second argument of the `reduce()` method. In the above case, it's `0`.

When we use the `forEach()` method, we do addition assignments `+=` to the `totalPrice` variable, so we declare it using the `let` keyword. But when using the `reduce()` method, you can use the `const` keyword because we only assign a value to the `totalPrice` once.

And that's basically how the `reduce()` method works. It iterates over each element in your array, and each iteration returns a single value, which is the `accumulator`.

When the iteration is finished, the `accumulator` value will be returned from the method.

When to Use the `reduce()` Method

As shown above, the `reduce()` method is recommended when you need to have a single value returned from iterating over your array.

This includes:

- Summarizing your values into a single value
- Grouping similar items together

The single value returned by the method can also be an array of objects, therefore containing multiple values.

You've seen how to sum values in the previous section, so let's see some examples of grouping items and removing duplicates next.

How to Group Similar Items Together

Suppose you have an array of objects again, but this time, the objects have `name` and `category` properties:

```
const items = [
  { name: 'Apple', category: 'Fruit' },
  { name: 'Onion', category: 'Vegetable' },
  { name: 'Orange', category: 'Fruit' },
  { name: 'Lettuce', category: 'Vegetable' },
];
```

Now, you want to group these items based on their `category` value. You can use the `reduce()` method and return a single object value.

First, you call the reduce method and pass an empty object `{}` as the initial `accumulator` value:

```
const groupedItems = items.reduce((accumulator, item) => {

}, {})
```

If not, then declare that property as an empty array as follows:

```
const category = item.category;
if (!accumulator[category]) {
  accumulator[category] = []
}
```

After that, push the `item.name` property to the `accumulator[category]` property, and return the `accumulator` like this:

```
accumulator[category].push(item.name);
return accumulator
```

And that's it. Now you have an object that groups the items based on the `category` value:

```
const items = [
  { name: 'Apple', category: 'Fruit' },
  { name: 'Onion', category: 'Vegetable' },
  { name: 'Orange', category: 'Fruit' },
  { name: 'Lettuce', category: 'Vegetable' },
];

const groupedItems = items.reduce((accumulator, item) => {
  const category = item.category;
  if (!accumulator[category]) {
    accumulator[category] = []
```

```
console.log(groupedItems);
// { Fruit: [ 'Apple', 'Orange' ], Vegetable: [ 'Onion', 'Lettuce' ] }
```

Next, let's see how to remove duplicates using the `reducer()` method:

How to Remove Duplicates Using the `reduce()` Method

To remove duplicates using the `reduce()` method, you need to declare an array as the `accumulator` value.

In each iteration, check if the `item` is already included in the `accumulator` using the `includes()` method.

If the `item` isn't already included, push the `item` into the `accumulator`. See the example below:

```
const items = [1, 2, 3, 1, 2, 3, 7, 8, 7];

const noDuplicateItems = items.reduce((accumulator, item) => {
  if (!accumulator.includes(item)) {
    accumulator.push(item);
  }
  return accumulator;
}, []);

console.log(noDuplicateItems);
// [ 1, 2, 3, 7, 8 ]
```

Why the Initial Accumulator Value is Important

In all the examples above, you've seen how we added an initial value for the accumulator as the second argument passed to the `reduce()` method.

If you don't add the initial accumulator value, then `reduce()` will take the first item in your array as the initial accumulator value.

You can test this by logging the `accumulator` value inside the callback function as follows:

```
const items = [
  { name: 'Apple', price: 1 },
  { name: 'Orange', price: 2 },
  { name: 'Mango', price: 3 },
];

const totalPrice = items.reduce((accumulator ,item) => {
  console.log(accumulator); // log the accumulator
  return accumulator += item.price;
}, 0)
```

Run the code above, and you'll get the following result:

```
0
1
```

In the first iteration, the `accumulator` uses the initial value we passed as the second argument to the `reduce()` method, which is a `0`.

If you remove `0` from the code, then the output will be:

```
{ name: 'Apple', price: 1 }
[object Object]2
```

Because we didn't provide an initial value for the `accumulator`, the method takes whatever value we placed at index 0 as the initial value of the `accumulator`.

Depending on the content of your array, the initial value can be an object, an array, or a single value. Relying on the value of the array as the initial accumulator value is considered bad practice as it can lead to bugs, so you should always define the initial accumulator value.

Full `reduce()` Callback Parameters

One more thing before we conclude the article. In all examples above, we define 2 parameters for the callback function, but the `reduce()` method actually passes 4 arguments to the callback function:

- The `accumulator` value
- The `item` value
- The `index` of the current item in iteration

The full syntax of the method is as follows:

```
Array.reduce((accumulator, item, index, array) => {
  // TODO: Define the process for each iteration here
}, initialAccumulatorValue)
```

In most cases, you only need the first two parameters, but if you somehow need the index and array values, they are always available.

Conclusion

And there you have it! At first sight, the `reduce()` method looks more complicated than other JavaScript array methods like `forEach()` and `filter()`. But once you understand the concept of reducer and accumulator, the method is actually quite simple.

In programming terms, a "reducer" function always returns a single value, so the `reduce()` method iterates over the values defined in your array and reduces them into a single value.

Inside the callback function of the `reduce()` method, you can do any kind of operation you need to achieve a certain result, such as summarizing and grouping certain values, or removing duplicates.

This is done with the help of the "accumulator", which stores the value returned from the previous iteration. You set the initial value of the accumulator by passing a second argument to the `reduce()` method.

The book is designed to be easy to understand and accessible to anyone looking to learn JavaScript. It provides a step-by-step gentle guide that will help you understand how to use JavaScript to create a dynamic application.

Here's my promise: *You will actually feel like you understand what you're doing with JavaScript.*

Until next time!

Read more posts.

If you read this far, thank the author to show them you care.

Say Thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) charity organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can make a tax-deductible donation here.

Trending Books and Handbooks

Learn CSS Transform	Build a Static Blog	Build an AI Chatbot
What is Programming?	Python Code Examples	Open Source for Devs
HTTP Networking in JS	Write React Unit Tests	Learn Algorithms in JS
How to Write Clean Code	Learn PHP	Learn Java
Learn Swift	Learn Golang	Learn Node.js