**How to Use .reduce() in JavaScript –**

[Alex Pickering](https://pickeringacw.medium.com/?source=post_page-----8f5413f17dac--------------------------------) Apr 22, 2020

An overview and quick guide to using some native JavaScript array functionality

When I first started trying to figure out what the .reduce() function does, it baffled me for quite a while. For some reason, I just couldn’t get it. Now that I’ve finally cracked the mystery, I’m hoping I may assist someone else in having their eureka moment. Let’s get started.

**How Does the .reduce() Function Work?**

Basically, .reduce() accepts a reducer function that we provide, which will be executed against each item within the given array. The result will be passed on to the next iteration as the accumulator variable. This function returns only the accumulator variable and will do so after the last iteration. As one might expect, the function iterates from left to right — you’ll see why I mention this in a bit.

Let’s take the following example. We have a group of tourists, and we want to find out the average age of the tour group. We could use a conventional for loop to determine this, as the example below illustrates.

But there is a cleaner way, with an added and often overlooked benefit. Let’s use the .reduce() function here to combine the ages and see what it might look like.

Here we can see that we no longer need to use a conventional for loop. We can employ the native functionality, provided to us by JavaScript itself, to make the code somewhat easier to read and more concise, with the added benefit that our new combinedAges variable has been created immutably.

Now that we have reduced the ages of the tour group to a grand total, we can quite simply determine the average age of the group. Phew! That was easy.

Another brief example may be a scenario in which we need to reduce an array of objects into a single object. We may not know the number of objects being returned, or we may be building up some configuration object, whatever the case may be. Let’s take a look at this.

The result, in this case, would be a flat object where the keys have been combined, and more notably, the next iteration’s currentValue overrides keys of the accumulator . If we have an unknown number of objects in the array, we can’t use the spread syntax, which makes this an alternative that may come in handy.

**What if We Need the Index?**

In some use cases, we might need to know the current index of the iteration. Thankfully, the reducer that we create and pass into the .reduce() function doesn’t just take two parameters; it can take four.

const fooReducer = (acc, value, index, array) => (...)const bar = [...]  
const value = bar.reduce(fooReducer)

The first parameter is always the accumulator value followed by the currentValue; but then, we can pass the index should we need it, as well as the array that the function was called on.

This allows for a lot of flexibility when it comes to the logic of the reducer we create and provide.

**Arrays Don’t Start at 1**

The actual .reduce() function itself doesn’t just take the reducer that we pass it as a parameter; it can take an initial value as well.

Above, we can see in the first reduction that we don’t provide any initial accumulator value. As a result, on the first iteration, the accumulator is set to the first value in the array on which it was called.

In the second case, we provide an initial accumulator value of 0, and on the first iteration, we can see that the accumulator is in fact set to 0.

This may be something to keep in mind, depending of course on your use case for the function.

**Left to Right and Back to Left**

If you’re new to using the .reduce function, then you may not be aware that there is also, in fact, a .reduceRight function as well. This function, however, works subtly differently and can be useful in certain use cases where the ordering in which the values are being accumulated may matter.

So how does .reduceRight differ from the .reduce function? Quite simply, instead of iterating left to right, the .reduceRight function iterates from right to left. It iterates backwards through the given array.

In all other aspects, these two functions perform exactly the same, though.

**Finally a Conclusion!**

In conclusion, using JavaScript’s native array functionalities can not only make our code more concise and easier to read, but it can also help usher in object immutability,which is often overlooked.

Thank you for reading. I hope you enjoyed and learned something. If you happen to have any feedback, criticism, or contributions, feel free to jot them down in the comment section below.

Auf Wiedersehen!

**References**

Here are some references to resources that may pique your interest should you wish to either learn more or begin to implement this native function in your own codebase, if you’re not already doing so.

[MDN .reduce()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce)  
[MDN .reduceRight()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/ReduceRight)