Reduce

Learn the last of the four powerful array functions. We'll cover how 'reduce' enables us to convert a collection into a single value.

`Array.reduce`

This is the hardest of these four array functions. Read this lesson multiple times if necessary. This isn't easy to get the hang of.

`reduce` is essentially a version of `forEach` that is meant to be a pure function.

We're doing something with each item in the array, but we're not necessarily being returned an array. We're also not supposed to be affecting the outer scope at all.

The purpose of `reduce` is to turn an array of items into a *single value*. We're going to start with a beginning value and each item in the array should transform our value in some way. **The end result after all items have transformed our value is what we get back from the function**.

The arguments that `reduce` gives our callback are a little different than the three we're used to. We also have the option of giving it an extra parameter. Let's show an example and then discuss it.

```
const arr = ['Hello', 'there!', 'How', 'are', 'you', 'doing?'];

const str = arr.reduce((accumulator, nextItem) => {
    var accumulatorWithNextItem = accumulator + nextItem + ' ';
    return accumulatorWithNextItem;
}, ''); // <- Passing in an extra parameter to reduce

console.log(str);
```

Here's another way to write the same code as above. I want to stress the empty string being passed in to `reduce`.

```
const arr = ['Hello', 'there!', 'How', 'are', 'you', 'doing?'];

function addNextItemToAccumulator(accumulator, nextItem) {
    return accumulator + nextItem + ' ';
}

const str = arr.reduce(addNextItemToAccumulator, ''); // <-
console.log(str);
```

We said before that we want to start with a value and let each item in our array transform it. We pass in an empty string `''` as the starting value here.

The first time `reduce` runs, it will pass our starting value into our callback as the first argument, `accumulator`. It'll pass our first array item as the second argument, `nextItem`.

So the first time reduce runs our callback, `accumulator` = `''` and `nextItem` = `'Hello'`.

Inside the callback, we want to do something with the accumulator and the next item. In this case, we're adding the next item to the end of our accumulator and adding a space afterward. This new string goes into the variable `accumulatorWithNextItem`.

We're returning this new string at the end. As `reduce` runs our callback a second time, this new string will be given to our callback as `accumulator`. `nextItem` will be the next item in the array.

So the second time the callback runs, `accumulator` = `'Hello '` and `nextItem` = `'there!'`. Again, we add the next item to the end of the accumulating string, and this new string gets passed back into our callback the next time around.

When `reduce` finally gets to the end of our array and is calling our callback for the last time, it behaves a little differently. **The last time our callback runs, the value it returns is what ends up in `str`.**

Let's log out each value of the accumulator and next item in the function above. Take a look at what's being printed.

```
const arr = ['Hello', 'there!', 'How', 'are', 'you', 'doing?'];

const str = arr.reduce((accumulator, nextItem) => {
    console.log('accumulator:', accumulator);
    console.log('nextItem:', nextItem);
    console.log('\n');

    const accumulatorWithNextItem = accumulator + nextItem + ' ';
    return accumulatorWithNextItem;
}, '');

console.log('Final string:', str);
```

We said before that the second parameter to `reduce`, the starting value, is optional. If provided, it's given to our callback as the first parameter the first time it runs.

If it's not provided, the first time our callback runs, `reduce` will simply pass the first item in our array as the `accumulator` and the second item in our array will be the `nextItem`. For further calls to our callback, it continues as usual.

In our case, it slightly changes our final value. There's no space between `Hello` and `there!` when we leave out the empty string parameter. See if you can spot why. Compare all the log outputs here to the block above.

```
const arr = ['Hello', 'there!', 'How', 'are', 'you', 'doing?'];

const str = arr.reduce((accumulator, nextItem) => {
    console.log('accumulator:', accumulator);
    console.log('nextItem:', nextItem);
    console.log('\n');

    const accumulatorWithNextItem = accumulator + nextItem + ' ';
    return accumulatorWithNextItem;
}); // <- No empty string being passed in

console.log('Final string:', str);
```

Like the other three array methods, `reduce` passes in the index and original array. They are the third and fourth arguments respectively.

```
const arr = ['a', 'b', 'c'];
const arr = ['a', 'b', 'c'];

arr.reduce((accum, next, index, array) => {
    console.log(
        `accum: ${accum}, next: ${next}, index: ${index}, array: ${JSON.stringify(array)}`
    );
    return accum + next;
}, '');
```

Which one should I use?

`reduce` and `forEach` both let us do the same thing. Most things that we want to do with one, we can do with the other. You may feel that `reduce` is more complex and requires a little more brainpower to reason about. However, it’s a pure function and using it correctly will prevent scope pollution, unnecessary variable declarations, and messy code.

`forEach` is easier to think about but causes the problems listed above. `forEach` can do everything `reduce` can do, however, and it’s easier to use, think about, and read, at least for now. So which one should we use?

The “correct” thing to do would be to use `reduce` when all we want is a value generated by processing our array. It’s a pure function that will prevent some bad practices and it also shows clear intent to other developers.

There are instances where side effects are necessary and in that case, it would be more appropriate to use `forEach`.

It’s ultimately up to you. When I learned these functions, I challenged myself to use `reduce` wherever it was appropriate instead of falling back to `forEach`. After some use, `reduce` will become second nature and I think it’s worth using until you get there with your understanding.

Many developers actively avoid `reduce` due to its difficulty in use and understanding. Even if you choose not to use it, make sure you understand what it does and how it works well enough to understand other people’s code.

Bonus: Try writing `reduce` yourself!

Writing `reduce` Exercise

```
function reduce() {
};
```

That’s it for `forEach` and `reduce`.