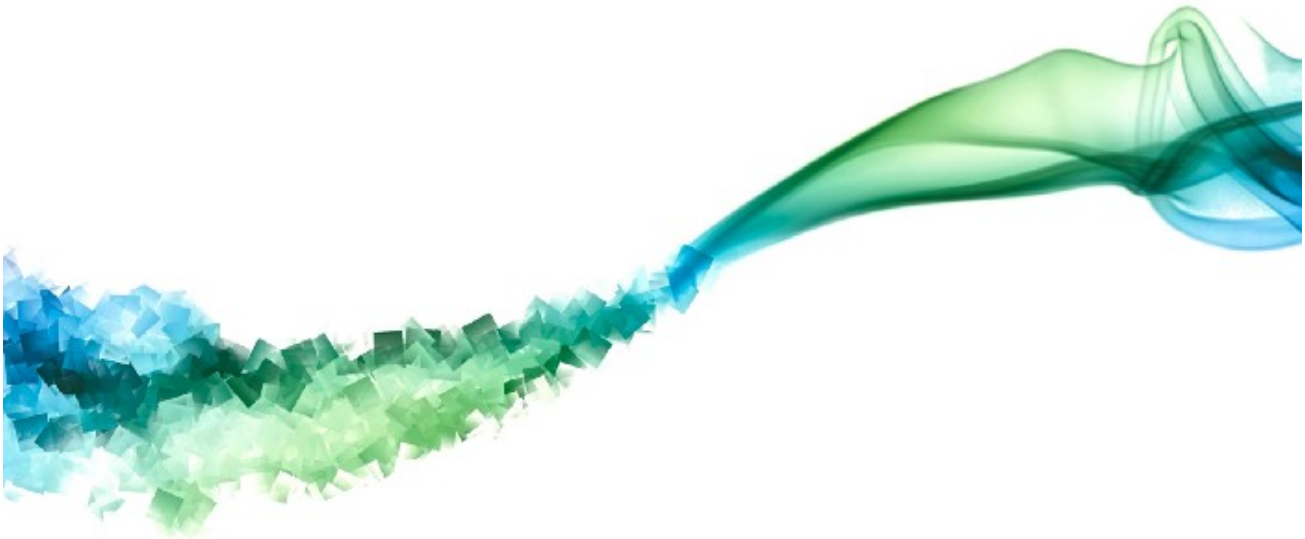
Reduce (Composing Software)

Eric Elliott Follow
Mar 5, 2017 · 6 min read



Smoke Art Cubes to Smoke — MattysFlicks — (CC BY 2.0)

> ***Note:*** *This is part of the "Composing Software" series* ***(now a book!)*** *on learning functional programming and compositional software techniques in JavaScript ES6+ from the ground up. Stay tuned. There's a lot more of this to come!*
> *< Previous* | *<< Start over at Part 1* | *Next >*

Reduce (aka: fold, accumulate) utility commonly used in functional programming that lets you iterate over a list, applying a function to an accumulated value and the next item in the list, until the iteration is complete and the accumulated value gets returned. Many useful things can be implemented with reduce. Frequently, it's the most elegant way to do any non-trivial processing on a collection of items.

Reduce takes a reducer function and an initial value, and returns the accumulated value. For `Array.prototype.reduce()`, the initial list is provided by `this`, so it's not one of the arguments:

```
array.reduce(
  reducer: (accumulator: Any, current: Any) => Any,
  initialValue: Any
) => accumulator: Any
```

Let's sum an array:

```
[2, 4, 6].reduce((acc, n) => acc + n, 0); // 12
```

For each element in the array, the reducer is called and passed the accumulator and the current value. The reducer's job is to "fold" the current value into the accumulated value somehow. How is not specified, and specifying how is the purpose of the reducer function. The reducer returns the new accumulated value, and `reduce()` moves on to the next value in the array. The reducer may need an initial value to start with, so most implementations take an initial value as a parameter.

In the case of this summing reducer, the first time the reducer is called, `acc` starts at `0` (the value we passed to `.reduce()` as the second parameter). The reducer returns `0 + 2` (`2` was the first element in the array), which is `2`. For the next call, `acc = 2, n = 4` and the reducer returns the result of `2 + 4` (`6`). In the last iteration, `acc = 6, n = 6`, and the reducer returns `12`. Since the iteration is finished, `.reduce()` returns the final accumulated value, `12`.

In this case, we passed in an anonymous reducing function, but we can abstract it and give it a name:

```
const summingReducer = (acc, n) => acc + n;
[2, 4, 6].reduce(summingReducer, 0); // 12
```

Normally, `reduce()` works left to right. In JavaScript, we also have `[].reduceRight()`, which works right to left. In other words, if you applied

`.reduceRight()` to `[2, 4, 6]`, the first iteration would use `6` as the first value for `n`, and it would work its way back and finish with `2`.

Reduce is Versatile

Reduce is versatile. It's easy to define `map()`, `filter()`, `forEach()` and lots of other interesting things using reduce:

Map:

```
const map = (fn, arr) => arr.reduce((acc, item, index, arr) => {
  return acc.concat(fn(item, index, arr));
}, []);
```

For map, our accumulated value is a new array with a new element for each value in the original array. The new values are generated by applying the passed in mapping function (`fn`) to each element in the `arr` argument. We accumulate the new array by calling `fn` with the current element, and concatenating the result to the accumulator array, `acc`.

Filter:

```
const filter = (fn, arr) => arr.reduce((newArr, item) => {
  return fn(item) ? newArr.concat([item]) : newArr;
}, []);
```

Filter works in much the same way as map, except that we take a predicate function and *conditionally* append the current value to the new array if the element passes the predicate check (`fn(item)` returns `true`).

For each of the above examples, you have a list of data, iterate over that data applying some function and folding the results into an accumulated value. Lots of applications spring to mind. But what if *your data is a list of functions?*

Compose:

Reduce is also a convenient way to compose functions. Remember function

composition: If you want to apply the function `f` to the result of `g` of `x` i.e., the composition, `f . g`, you could use the following JavaScript:

```
f(g(x))
```

Reduce lets us abstract that process to work on any number of functions, so you could easily define a function that would represent:

```
f(g(h(x)))
```

To make that happen, we'll need to run reduce in reverse. That is, right-to-left, rather than left-to-right. Thankfully, JavaScript provides a `.reduceRight()` method:

```
const compose = (...fns) => x => fns.reduceRight((v, f) => f(v),
x);
```

> *Note: If JavaScript had not provided `[].reduceRight()`, you could still implement `reduceRight()` -- using `reduce()`. I'll leave it to adventurous readers to figure out how.*

Pipe:

`compose()` is great if you want to represent the composition from the inside-out -- that is, in the math notation sense. But what if you want to think of it as a sequence of events?

Imagine we want to add `1` to a number and then double it. With `compose(), that would be:

```
const add1 = n => n + 1;
const double = n => n * 2;

const add1ThenDouble = compose(
  double,
  add1
```

```
);

add1ThenDouble(2); // 6
// ((2 + 1 = 3) * 2 = 6)
```

See the problem? The first step is listed last, so in order to understand the sequence, you'll need to start at the bottom of the list and work your way backwards to the top.

Or we can reduce left-to-right as you normally would, instead of right-to-left:

```
const pipe = (...fns) => x => fns.reduce((v, f) => f(v), x);
```

Now you can write `add1ThenDouble()` like this:

```
const add1ThenDouble = pipe(
  add1,
  double
);

add1ThenDouble(2); // 6
// ((2 + 1 = 3) * 2 = 6)
```

This is important because sometimes if you compose backwards, you get a different result:

```
const doubleThenAdd1 = pipe(
  double,
  add1
);

doubleThenAdd1(2); // 5
```

We'll go into more details on `compose()` and `pipe()` later. What you should understand right now is that `reduce()` is a very powerful tool, and you really need to learn it. Just be aware that if you get very tricky with reduce, some people may have a hard time following along.

A Word on Redux

You may have heard the term “reducer” used to describe the important state update bits of Redux. As of this writing, Redux is the most popular state management library/architecture for web applications built using React and Angular (the latter via `ngrx/store`).

Redux uses reducer functions to manage application state. A Redux-style reducer takes the current state and an action object and returns the new state:

```
reducer(state: Any, action: { type: String, payload: Any}) =>
newState: Any
```

Redux has some reducer rules you need to keep in mind:

1. A reducer called with no parameters should return its valid initial state.
2. If the reducer isn’t going to handle the action type, it still needs to return the state.
3. Redux reducers **must be pure functions.**

Let’s rewrite our summing reducer as a Redux-style reducer that reduces over action objects:

```
const ADD_VALUE = 'ADD_VALUE';

const summingReducer = (state = 0, action = {}) => {
  const { type, payload } = action;

  switch (type) {
    case ADD_VALUE:
      return state + payload.value;
    default: return state;
  }
};
```

The cool thing about Redux is that the reducers are just standard reducers that you

can plug into any `reduce()` implementation which respects the reducer function signature, including `[].reduce()`. That means you can create an array of action objects and reduce over them to get a snapshot of state representing the same state you'd have if those same actions were dispatched to your store:

```
const actions = [
  { type: 'ADD_VALUE', payload: { value: 1 } },
  { type: 'ADD_VALUE', payload: { value: 1 } },
  { type: 'ADD_VALUE', payload: { value: 1 } },
];

actions.reduce(summingReducer, 0); // 3
```

That makes unit testing Redux-style reducers a breeze.

Conclusion

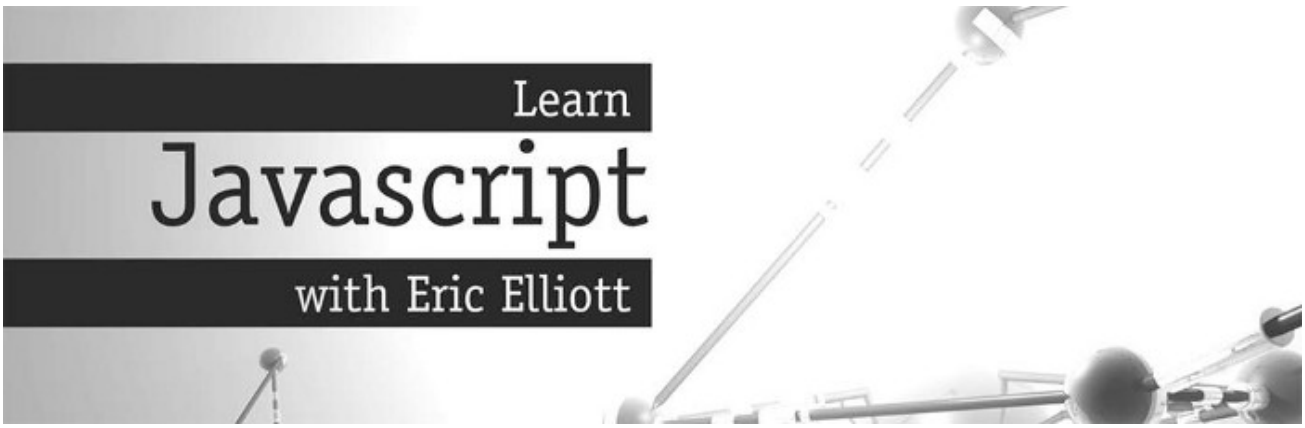
You should be starting to see that reduce is an incredibly useful and versatile abstraction. It's definitely a little trickier to understand than map or filter, but it is an essential tool in your functional programming utility belt — one you can use to make a lot of other great tools.

Next: Functors & Categories >

Next Steps

Want to learn more about functional programming in JavaScript?

Learn JavaScript with Eric Elliott. If you're not a member, you're missing out!



***Eric Elliott** is the author of "Programming JavaScript Applications" (O'Reilly), and*

"Learn JavaScript with Eric Elliott". He has contributed to software experiences for ***Adobe Systems****,* ***Zumba Fitness****,* ***The Wall Street Journal****,* ***ESPN****,* ***BBC****, and top recording artists including* ***Usher****,* ***Frank Ocean****,* ***Metallica****, and many more.*

He spends most of his time in the San Francisco Bay Area with the most beautiful woman in the world.

Thanks to JS_Cheerleader.

JavaScript Functional Programming