How to Use Map, Filter, and Reduce in JavaScript

Peleke Sengstacke Nov 20, 2020 (Updated Oct 16, 2021)

13 likes | 16 mins | English

Functional programming has been making quite a splash in the development world these days. And for good reason: Functional techniques can help you write more declarative code that is easier to understand at a glance, refactor, and test.

One of the cornerstones of functional programming is its special use of lists and list operations. Those things are exactly what they sound like: arrays of things and the stuff you do to them. But the functional mindset treats them a bit differently than you might expect.

How to Use Map, Filter, & Reduce in JavaScript

This article will take a close look at what I like to call the "big three" list operations: `map`, `filter`, and `reduce`. Wrapping your head around these three functions is an important step towards being able to write clean, functional code, and it opens the doors to the vastly powerful techniques of functional and reactive programming.

Curious? Let's dive in.

A Map From List to List

Often, we find ourselves needing to take an array and modify every element in it in exactly the same way. Typical examples of this are squaring every element in an array of numbers, retrieving the name from a list of users, or running a regex against an array of strings.

`map` is a method built to do exactly that. It's defined on `Array.prototype`, so you can call it on any array, and it accepts a callback as its first argument.

The syntax for `map` is shown below.

```
let newArray = arr.map(callback(currentValue[, index[, array]]) {
  // return element for newArray, after executing something
}[, thisArg]);
```

When you call `map` on an array, it executes that callback on every element within it, returning a *new* array with all of the values that the callback returned.

Under the hood, `map` passes three arguments to your callback:

1. the *current item* in the array
2. the *array index* of the current item

3. the *entire array* you called `map` on

Let's look at some code.

`map` in Practice

Suppose we have an app that maintains an array of your tasks for the day. Each `task` is an object, each with a `name` and `duration` property:

```
// Durations are in minutes
const tasks = [
  {
    'name'     : 'Write for Envato Tuts+',
    'duration' : 120
  },
  {
    'name'     : 'Work out',
    'duration' : 60
  },
  {
    'name'     : 'Procrastinate on Duolingo',
    'duration' : 240
  }
];
```

Let's say we want to create a new array with just the name of each task, so we can take a look at everything we've done today. Using a `for` loop, we'd write something like this:

```
const task_names = [];

for (let i = 0, max = tasks.length; i < max; i += 1) {
    task_names.push(tasks[i].name);
}

console.log(task_names) // [ 'Write for Envato Tuts+', 'Work out', 'Procrastinate on
```

JavaScript also offers a `forEach` loop. It functions like a `for` loop, but manages all the messiness of checking our loop index against the array length for us:

```
const task_names = [];

tasks.forEach(function (task) {
    task_names.push(task.name);
});

console.log(task_names) // [ 'Write for Envato Tuts+', 'Work out', 'Procrastinate on
```

Using `map`, we can simply write:

```
const task_names = tasks.map(function (task, index, array) {
    return task.name;
});

console.log(task_names) // [ 'Write for Envato Tuts+', 'Work out', 'Procrastinate on
```

Here I included the `index` and `array` parameters to remind you that they're there if you need them. Since I didn't use them here, though, you could leave them out, and the code would run just fine.

An even more succinct way of writing `map` in modern JavaScript is with arrow functions.

```
const task_names = tasks.map(task => task.name)

console.log(task_names) // ['Write for Envato Tuts+', 'Work out', 'Procrastinate on D
```

Arrow functions are a short form for one-line functions that just have a `return` statement. It doesn't get much more readable than that.

There are a few important differences between the different approaches:

1. Using `map`, you don't have to manage the state of the `for` loop yourself.
2. With `map`, you can operate on the element directly, rather than having to index into the array.
3. You don't have to create a new array and `push` into it. `map` returns the finished product all in one go, so we can simply assign the return value to a new variable.
4. You *do* have to remember to include a `return` statement in your callback. If you don't, you'll get a new array filled with `undefined`.

Turns out, *all* of the functions we'll look at today share these characteristics.

The fact that we don't have to manually manage the state of the loop makes our code simpler and more maintainable. The fact that we can operate directly on the element instead of having to index into the array makes things more readable.

Using a `forEach` loop solves both of these problems for us. But `map` still has at least two distinct advantages:

1. `forEach` returns `undefined`, so it doesn't chain with other array methods. `map` returns an array, so you *can* chain it with other array methods.
2. `map` returns an array with the finished product, rather than requiring us to mutate an array inside the loop.

Keeping the number of places where you modify state to an absolute minimum is an important tenet of functional programming. It makes for safer and more intelligible code.

Gotchas

The callback you pass to `map` must have an explicit `return` statement, or `map` will spit out an array full of `undefined`. It's not hard to remember to include a `return` value
it's not hard to forget.

If you *do* forget, `map` won't complain. Instead, it'll quietly hand back an array full of nothing. Silent errors like that can be surprisingly hard to debug.

Fortunately, this is the *only* gotcha with `map`. But it's a common enough pitfall that I'm obliged to emphasize: Always make sure your callback contains a `return` statement!

Implementation

Reading implementations is an important part of understanding. So let's write our own lightweight `map` to better understand what's going on under the hood. If you want to see a production-quality implementation, check out Mozilla's polyfill at MDN.

```
let map = function (array, callback) {
  const new_array = [];

  array.forEach(function (element, index, array) {
    new_array.push(callback(element));
  });

  return new_array;
};
```

This code accepts an array and a callback function as arguments. It then creates a new array, executes the callback on each element on the array we passed in, pushes the results into the new array, and returns the new array. If you run this in your console, you'll get the same result as before.

While we're using a for loop under the hood, wrapping it up into a function hides the details and lets us work with the abstraction instead.

That makes our code more declarative—it says *what* to do, not *how* to do it. You'll appreciate how much more readable, maintainable, and, erm, *debuggable* this can make your code.

Filter Out the Noise

The next of our array operations is `filter`. It does exactly what it sounds like: It takes an array and filters out unwanted elements.

The syntax for filter is:

```
let newArray = arr.filter(callback(currentValue[, index[, array]]) {
  // return element for newArray, if true
}[, thisArg]);
```

Just like `map`, `filter` passes your callback three arguments:

1. the *current item*
2. the *current index*
3. the *array* you called `filter` on

Consider the following example, which filters out any string which is less than 8 characters.

```
const words = ['Python', 'Javascript', 'Go', 'Java', 'PHP', 'Ruby'];
const result = words.filter(word => word.length < 8);
console.log(result);
```

The expected result will be:

```
[ 'Python', 'Go', 'Java', 'PHP', 'Ruby' ]
```

Let's revisit our task example. Instead of pulling out the names of each task, let's say I want to get a list of just the tasks that took me two hours or more to get done.

Using `forEach`, we'd write:

```
const difficult_tasks = [];

tasks.forEach(function (task) {
    if (task.duration >= 120) {
        difficult_tasks.push(task);
    }
});

console.log(difficult_tasks)

//  [{ name: 'Write for Envato Tuts+', duration: 120 },
//   { name: 'Procrastinate on Duolingo', duration: 240 }
//   ]
```

With `filter`, we can simply write:

```
const difficult_tasks = tasks.filter((task) => task.duration >= 120 );
```

Just like `map`, `filter` lets us:

- avoid mutating an array inside a `forEach` or `for` loop
- assign its result directly to a new variable, rather than push into an array we defined elsewhere

Gotchas

The callback you pass to `map` has to include a return statement if you want it to function properly. With `filter`, you also have to include a return statement (unless you're using arrow functions), and you *must* make sure it returns a boolean value.

If you forget your return statement, your callback will return `undefined`, which `filter` will unhelpfully coerce to `false`. Instead of throwing an error, it will silently return an empty array!

If you go the other route and return something that's isn't explicitly `true` or `false`, then `filter` will try to figure out what you meant by applying **JavaScript's type coercion rules**. More often than not, this is a bug. And, just like forgetting your return statement, it'll be a silent one.

Always make sure your callbacks include an explicit return statement. And *always* make sure your callbacks in `filter` return `true` or `false`. Your sanity will thank you.

Implementation

Once again, the best way to understand a piece of code is... well, to write it. Let's roll our own lightweight `filter`. The good folks at Mozilla have an **industrial-strength polyfill** for you to read, too.

```
const filter = function (array, callback) {

    const filtered_array = [];

    array.forEach(function (element, index, array) {
        if (callback(element, index, array)) {
            filtered_array.push(element);
        }
    });

    return filtered_array;

};
```

The Reduce Method

The syntax for the `reduce` array method in JavaScript is:

```
let newArray = arr.filter(callback(currentValue, accumulatedValue) {
  // return the accumulated value, given the current and previous  accumulated
}, initialValue[, thisArg]);
```

`map` creates a new array by transforming every element in an array individually. `filter` creates a new array by removing elements that don't belong. `reduce`, on the other hand, takes all of the elements in an array and *reduces* them into a single value.

Just like `map` and `filter`, `reduce` is defined on `Array.prototype` and so is available on any array, and you pass a callback as its first argument. But it also takes a second argument: the value to start combining all your array elements into.

`reduce` passes your callback four arguments:

1. the *current value*
2. the *previous value*
3. the *current index*
4. the *array* you called `reduce` on

Notice that the callback gets a *previous value* on each iteration. On the first iteration, there *is* no previous value. This is why you have the option to pass `reduce` an initial value: It acts as the "previous value" for the first iteration, when there otherwise wouldn't be one.

Finally, bear in mind that `reduce` returns a single value, *not* an array containing a single item. This is more important than it might seem, and I'll come back to it in the examples.

`reduce` in Practice

Let's say you want to find the sum of a list of numbers. Using a loop, it would look like this:

```
let numbers = [1, 2, 3, 4, 5],
    total = 0;

numbers.forEach(function (number) {
    total += number;
});

```

```
console.log(total); // 15
```

While this isn't a bad use case for `forEach`, `reduce` still has the advantage of allowing us to avoid mutation. With `reduce`, we would write:

```
const total = [1, 2, 3, 4, 5].reduce(function (previous, current) {
    return previous + current;
}, 0);
console.log(total); // 15
```

First, we call `reduce` on our list of numbers. We pass it a callback, which accepts the previous value and current value as arguments, and returns the result of adding them together. Since we passed `0` as a second argument to `reduce`, it'll use that as the value of `previous` on the first iteration.

With arrow functions, we would write it like this:

```
const total = [1, 2, 3, 4, 5].reduce((previous, current) => previous+current),0;
console.log(total) // 15
```

If we take it step by step, it looks like this:

Iteration	Previous	Current	Total
1	0	1	1
2	1	2	3
3	3	3	6
4	6	4	10
5	10	5	15

If you're not a fan of tables, run this snippet in the console:

```
const total = [1, 2, 3, 4, 5].reduce(function (previous, current, index) {
    const val = previous + current;
    console.log("The previous value is " + previous +
                "; the current value is " + current +
                ", and the current iteration is " + (index + 1));
    return val;
}, 0);

console.log("The loop is done, and the final value is " + total + ".");
```

To recap: `reduce` iterates over all the elements of an array, combining them however you specify in your callback. On every iteration, your callback has access to the *previous value*, which is the *total-so-far*, or *accumulated value*; the *current value*; the *current index;* and the entire *array*, if you need them.

Let's turn back to our tasks example. We've gotten a list of task names from `map`, and a filtered list of tasks that took a long time with... well, `filter`.

What if we wanted to know the total amount of time we spent working today?

Using a `forEach` loop, you'd write:

```
let total_time = 0;

tasks.forEach(function (task) {
    // The plus sign just coerces
    // task.duration from a String to a Number
    total_time += (+task.duration);
});

console.log(total_time)

//expected result is 420
```

With `reduce`, that becomes:

```
total_time = tasks.reduce((previous, current) => previous + current.duration, 0);
console.log(total_time); //420
```

That's almost all there is to it. Almost, because JavaScript provides us with one more little-known method, called `reduceRight`. In the examples above, `reduce` started a *first* item in the array, iterating from left to right:

```
let array_of_arrays = [[1, 2], [3, 4], [5, 6]];
const concatenated = array_of_arrays.reduce( function (previous, current) {
        return previous.concat(current);
});

console.log(concatenated); // [1, 2, 3, 4, 5, 6];
```

`reduceRight` does the same thing, but in the opposite direction:

```
let array_of_arrays = [[1, 2], [3, 4], [5, 6]];
const concatenated = array_of_arrays.reduceRight( function (previous, current) {
        return previous.concat(current);
});

console.log(concatenated); // [5, 6, 3, 4, 1, 2];
```

I use `reduce` every day, but I've never needed `reduceRight`. I reckon you probably won't, either. But in the event you ever do, now you know it's there.

Gotchas

The three big gotchas with `reduce` are:

1. forgetting to `return`
2. forgetting an initial value
3. expecting an array when `reduce` returns a single value

Fortunately, the first two are easy to avoid. Deciding what your initial value should be depends on what you're doing, but you'll get the hang of it quickly.

The last one might seem a bit strange. If `reduce` only ever returns a single value, why would you expect an array?

There are a few good reasons for that. First, `reduce` always returns a single *value*, not always a single *number*. If you reduce an array of arrays, for instance, it will return a single array. If you're in the habit of reducing arrays, it would be fair to expect that an array containing a single item wouldn't be a special case.

Second, if `reduce` *did* return an array with a single value, it would naturally play nice with `map` and `filter`, and other functions on arrays that you're likely to be using with it.

Implementation

Time for our last look under the hood. As usual, Mozilla has a bulletproof polyfill for reduce if you want to check it out.

```
let reduce = function (array, callback, initial) {
    let accumulator = initial || 0;

    array.forEach(function (element) {
       accumulator = callback(accumulator, array[i]);
    });

    return accumulator;
};
```

Two things to note here:

1. This time, I used the name `accumulator` instead of `previous`. This is what you'll usually see in the wild.
2. I assign `accumulator` an initial value if a user provides one, and default to `0` if not. This is how the real `reduce` behaves, as well.

Putting It Together: Map, Filter, Reduce, and Chainability

At this point, you might not be *that* impressed. Fair enough: `map`, `filter`, and `reduce`, on their own, aren't awfully interesting. After all, their true power lies in their chainability.

Let's say I want to do the following:

1. Collect two days' worth of tasks.
2. Convert the task durations to hours instead of minutes.
3. Filter out everything that took two hours or more.
4. Sum it all up.
5. Multiply the result by a per-hour rate for billing.
6. Output a formatted dollar amount.

First, let's define our tasks for Monday and Tuesday:

```
const monday = [
        {
            'name'     : 'Write a tutorial',
            'duration' : 180
        },
        {
            'name'     : 'Some web development',
            'duration' : 120
        }
    ];

const tuesday = [
        {
            'name'     : 'Keep writing that tutorial',
            'duration' : 240
        },
        {
            'name'     : 'Some more web development',
            'duration' : 180
        },
        {
            'name'     : 'A whole lot of nothing',
            'duration'  : 240
        }
    ];

const tasks = [monday, tuesday];
```

And now, our lovely-looking transformation:

```
01  const result = tasks
02      // Concatenate our 2D array into a single list
03      .reduce((acc, current) => acc.concat(current))
04      // Extract the task duration, and convert minutes to hours
05      .map((task) => task.duration / 60)
06      // Filter out any task that took less than two hours
07      .filter((duration) => duration >= 2)
08      // Multiply each tasks' duration by our hourly rate
09      .map((duration) => duration * 25)
10      // Combine the sums into a single dollar amount
11      .reduce((acc, current) => [(+acc) + (+current)])
12      // Convert to a "pretty-printed" dollar amount
13      .map((amount) => '$' + amount.toFixed(2))
14      // Pull out the only element of the array we got from map
15      .reduce((formatted_amount) =>formatted_amount);
```

If you've made it this far, this should be pretty straightforward. There are two bits of weirdness to explain, though.

First, on line 10, I have to write:

```
// Remainder omitted
reduce(function (accumulator, current) {
    return [(+accumulator) + (+current_];
})
```

Two things to explain here:

1. The plus signs in front of `accumulator` and `current` coerce their values to numbers. If you don't do this, the return value will be the rather useless string, `"12510075100"`.
2. If you don't wrap that sum in brackets, `reduce` will spit out a single value, *not* an array. That would end up throwing a `TypeError`, because you can only use `map` on an array!

The second bit that might make you a bit uncomfortable is the last `reduce`, namely:

```
// Remainder omitted
map(function (dollar_amount) {
    return '$' + dollar_amount.toFixed(2);
}).reduce(function (formatted_dollar_amount) {
    return formatted_dollar_amount;
});
```

That call to `map` returns an array containing a single value. Here, we call `reduce` to pull out that value.

Finally, let's see how our friend the `forEach` loop would get it done:

```
let concatenated = monday.concat(tuesday),
    fees = [],
    hourly_rate = 25,
    total_fee = 0;

concatenated.forEach(function (task) {
    let duration = task.duration / 60;

    if (duration >= 2) {
        fees.push(duration * hourly_rate);
    }
});

fees.forEach(function (fee) {
    total_fee += fee;
});



console.log(total_fee); //400
```

Tolerable, but noisy.

Conclusion and Next Steps

In this tutorial, you've learned how `map`, `filter`, and `reduce` work; how to use them; and roughly how they're implemented. You've seen that they all allow you to avoid mutating state, which using `for` and `forEach` loops requires, and you should now have a good idea of how to chain them all together.

By now, I'm sure you're eager for practice and further reading. For a masterclass in functional programming in JavaScript, check out our online course.

Questions, comments, or confusions? Leave them in the comment section.

JavaScript   Functional Programming

Did you find this post useful?

👍 Yes   👎 No

Want a weekly email summary?

Subscribe below and we'll send you a weekly email summary of all new Code tutorials. Never miss out on learning about the next big thing.

Sign up

Peleke Sengstacke

Fayetteville, NY

Peleke Sengstacke is a web and mobile developer with a penchant for functional programming. When not hacking around in Node, he's hacking around on Android; and when not on Android, in Clojure.

He likes linguistics, avocados, and lifting weight. He dislikes mosquitoes, the scent of durian, and merge conflicts. Make his day by following him on Twitter (@PelekeS) or bookmarking his blog (http://peleke.me).

PelekeS

FEED LIKE FOLLOW

tuts+

30,489 Tutorials

1,316 Courses

50,290 Translations

Certified B Corporation