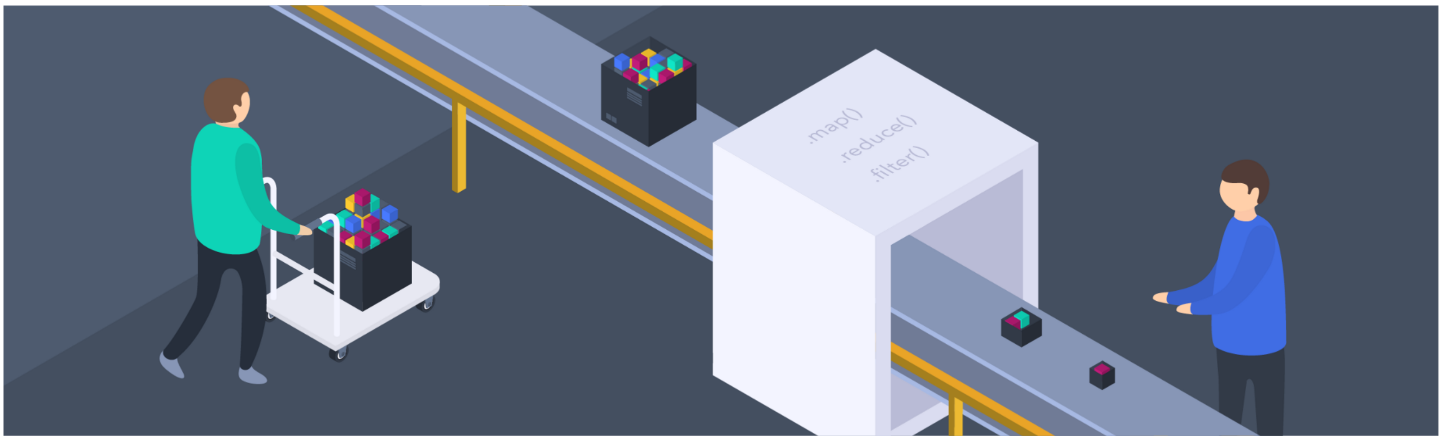

Visuals by: Julie Dorion-Bélanger. Thanks Julie!

Simplify your JavaScript – Use .map(), .reduce(), and .filter()

Etienne Talbot Follow
Jan 29 · 6 min read

If you're starting in JavaScript, maybe you haven't heard of `.map()` , `.reduce()` , and `.filter()` . For me, it took a while as I had to support Internet Explorer 8 until a couple years ago. But if you don't need to be compatible with this very old browser, you have to become familiar with those methods.

Take note that this article most likely applies to whatever other programming language you might be using, as these are concepts that exist in many other languages.

.map()

Let me explain how it works with a simple example. Say you have received an array containing multiple objects – each one representing a person. The thing you really need in the end, though, is an array containing only the id of each person.

```
// What you have
var officers = [
  { id: 20, name: 'Captain Piett' },
  { id: 24, name: 'General Veers' },
  { id: 56, name: 'Admiral Ozzel' },
  { id: 88, name: 'Commander Jerjerrod' }
];

// What you need
[20, 24, 56, 88]
```

There are multiple ways to achieve this. You might want to do it by creating an empty array, then using `.forEach()` , `.for(...of)` , or a simple `.for()` to meet your goal.

Let’s compare!

Using `.forEach()` :

```
var officersIds = [];

officers.forEach(function (officer) {
  officersIds.push(officer.id);
});
```

Notice how you have to create an empty array beforehand? Let’s see what it looks like when using `.map()` :

```
var officersIds = officers.map(function (officer) {
  return officer.id
});
```

We can even be more concise with arrow functions (requires ES6 support, Babel or TypeScript)

```
const officersIds = officers.map(officer => officer.id);
```

So how does `.map()` work? Basically is takes 2 arguments, a callback and an optional context (will be considered as `this` in the callback) which I did not use in the previous example. The callback runs for **each value in the array** and **returns each new value** in the resulting array.

Keep in mind that the resulting array will always be the same length as the original array.

.reduce()

Just like `.map()`, `.reduce()` also runs a callback for each element of an array. What's different here is that reduce passes the result of this callback (the accumulator) from one array element to the other.

The accumulator can be pretty much anything (integer, string, object, etc.) and must be instantiated or passed when calling `.reduce()`.

Time for an example! Say you have an array with these pilots and their respective years of experience:

```
var pilots = [
  {
    id: 10,
    name: "Poe Dameron",
    years: 14,
  },
  {
    id: 2,
    name: "Temmin 'Snap' Wexley",
    years: 30,
  },
  {
    id: 41,
    name: "Tallissan Lintra",
    years: 16,
  },
  {
    id: 99,
    name: "Ello Asty",
    years: 22,
  }
];
```

We need to know the total years of experience of all of them.
With `.reduce()`, it's pretty straightforward:

```
var totalYears = pilots.reduce(function (accumulator,
pilot) {
  return accumulator + pilot.years;
}, 0);
```

Notice that I've set the starting value as 0. I could have also used an existing variable if necessary. After running the callback for each element of the array, reduce will return the final value of our accumulator (in our case: `82`).

Let's see how this can be shortened with ES6's arrow functions:

```
const totalYears = pilots.reduce((acc, pilot) => acc +
pilot.years, 0);
```

Now let's say I want to find which pilot is the most experienced one. For that, I can use reduce as well:

```
var mostExpPilot = pilots.reduce(function (oldest, pilot) {
  return (oldest.years || 0) > pilot.years ? oldest :
pilot;
}, {});
```

I named my accumulator `oldest` . My callback compares the accumulator to each pilot. If a pilot has more years of experience than `oldest` , then that pilot becomes the new `oldest` so that's the one I return.

As you can see, using `.reduce()` is an easy way to generate a single value or object from an array.

.filter()

What if you have an array, but only want some of the elements in it? That's where `.filter()` comes in!

Here's our data:

```
var pilots = [
  {
    id: 2,
    name: "Wedge Antilles",
    faction: "Rebels",
  },
  {
    id: 8,
    name: "Ciena Ree",
    faction: "Empire",
  },
  {
    id: 40,
    name: "Iden Versio",
    faction: "Empire",
  },
  {
    id: 66,
    name: "Thane Kyrell",
    faction: "Rebels",
  }
];
```

Say we want two arrays now: one for rebel pilots, the other one for imperials. With `.filter()` it couldn't be easier!

```
var rebels = pilots.filter(function (pilot) {
  return pilot.faction === "Rebels";
});

var empire = pilots.filter(function (pilot) {
  return pilot.faction === "Empire";
});
```

That's it! And it's even shorter with arrow functions:

```
const rebels = pilots.filter(pilot => pilot.faction ===
"Rebels");
const empire = pilots.filter(pilot => pilot.faction ===
"Empire");
```

Basically, if the callback function **returns true**, the current element **will be in the resulting array**. If it returns false, it won't be.

Combining .map(), .reduce(), and .filter()

Since all three are called on arrays and since `.map()` and `.filter()` both return arrays, we can easily chain our calls.

Let's check out another example. Here's our data:

```
var personnel = [
  {
    id: 5,
    name: "Luke Skywalker",
    pilotingScore: 98,
    shootingScore: 56,
    isForceUser: true,
  },
  {
    id: 82,
    name: "Sabine Wren",
    pilotingScore: 73,
    shootingScore: 99,
    isForceUser: false,
  },
  {
    id: 22,
    name: "Zeb Orellios",
    pilotingScore: 20,
    shootingScore: 59,
    isForceUser: false,
  },
  {
    id: 15,
    name: "Ezra Bridger",
    pilotingScore: 43,
    shootingScore: 67,
    isForceUser: true,
  },
  {
    id: 11,
    name: "Caleb Dume",
    pilotingScore: 71,
    shootingScore: 85,
    isForceUser: true,
  },
];
```

Our objective: get the total score of force users only. Let’s do it step by step!

First, we need to filter out the personnel who can’t use the force:

```
var jediPersonnel = personnel.filter(function (person) {
  return person.isForceUser;
});

// Result: [{...}, {...}, {...}] (Luke, Ezra and Caleb)
```

With that we have 3 elements left in our resulting array. We now need to create an array containing the total score of each Jedi.

```
var jediScores = jediPersonnel.map(function (jedi) {
  return jedi.pilotingScore + jedi.shootingScore;
});

// Result: [154, 110, 156]
```

And let's use reduce to get the total:

```
var totalJediScore = jediScores.reduce(function (acc,
score) {
  return acc + score;
}, 0);

// Result: 420
```

And now here's the fun part… we can chain all of this to get what we want in a single line:

```
var totalJediScore = personnel
  .filter(function (person) {
    return person.isForceUser;
  })
  .map(function (jedi) {
    return jedi.pilotingScore + jedi.shootingScore;
  })
  .reduce(function (acc, score) {
    return acc + score;
  }, 0);
```

And look how pretty it is with arrow functions:

```
const totalJediScore = personnel
  .filter(person => person.isForceUser)
  .map(jedi => jedi.pilotingScore + jedi.shootingScore)
  .reduce((acc, score) => acc + score, 0);
```

Boom! 💥

Note: In my previous example, `.map()` and `.filter()` weren't even necessary. We could easily achieve the same result with only `.reduce()`. I left them in there for the sake of this example. Can you guess how we could *only keep `.reduce()` and get the same result with one line of code? See the solution on CodePen*

Why not use .forEach()?

I used to use `for` loops everywhere instead of `.map()`, `.reduce()`, and `.filter()`. But a couple of years ago I started working a lot more with data that came from an API. That's where I began to see the advantages of leaving `.forEach` behind.

Formatting

Say you need to display a list of people, with their name and job title.

```
var data = [
  {
    name: "Jan Dodonna",
    title: "General",
  },
  {
    name: "Gial Ackbar",
    title: "Admiral",
  },
]
```

The API gives you the above data, but you only need the title and the last name of each person… You need to format the data. However, your app also needs to have a single view for each person, so you must write a data formatting function that both works in a list view and in a single view.

That means you can't have the `.forEach` loop inside of your formatting function, or else you would have to wrap your single element in an array before you pass it to the function just to make it work, like so:

```
var result = formatElement([element])[0];
// Yeah... that's not right at all
```

So your loop has to wrap the call of the function, like this:

```
data.forEach(function (element) {
  var formatted = formatElement(element);
  // But what then....
});
```

But `.forEach()` doesn't return anything. It can't. That means you have to push the results inside a predetermined array.

```
var results = [];

data.forEach(function (element) {
  var formatted = formatElement(element);
  results.push(formatted);
});
```

As a result, you have 2 functions: your `formatElement()` function and your function that pushes the results in your array.

Why have 2 functions when you can have just one?

```
var results = data.map(formatElement);
```

Testing is easier

If you write unit tests for your code, you'll find it simpler to test the functions you call with `.map()`, `.reduce()`, or `.filter()`.

All you have to do is provide inbound data for the function and expect a result to come out. Basically "what comes out if **this** is passed?". Less manipulation, less `beforeEach()`s and `afterEach()`s. It's straightforward, simple testing.

Try it!

Try to replace some of your `for` loops with `.map()`, `.reduce()`, `.filter()` where it seems to fit. I guarantee your code will be way less clunky and much easier to read.

Keep coding!