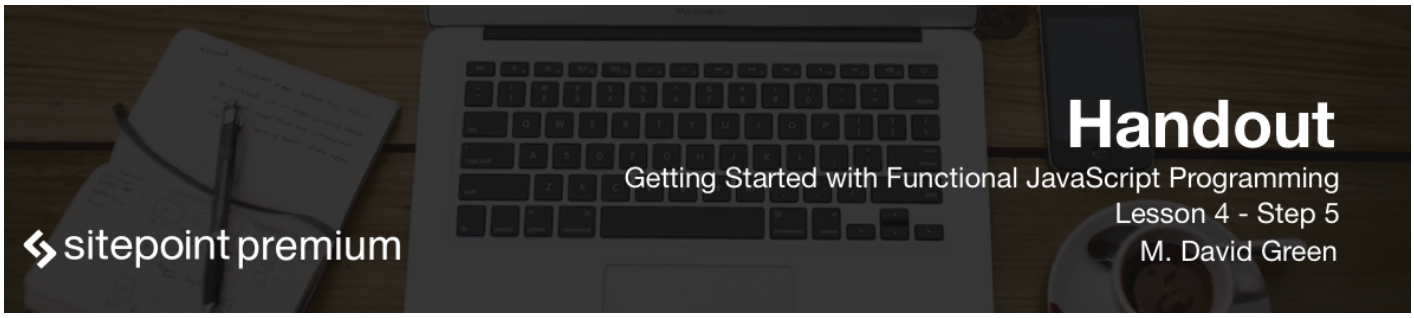

So in this lesson I've introduced you to the `map` method, the `reduce` method, and the `filter` method. All of which were introduced in ECMAScript 5, and provide some good functional capabilities inside of JavaScript. But before we move on to the next topic, I do want to talk a little bit about performance with these methods, and also something about how to use them effectively.

If you've been reading a little bit about functional programming in JavaScript, you've probably heard some concerns raised about the performance of these functional methods in JavaScript. It's absolutely true that loops currently perform much better in most browsers, and in most JavaScript engines in general. These engines just are not optimized yet to take advantage of these methods.

And in fact, if you look at the statistics from Browserscope, you can see that in a recent test comparing `for` and `while` loops to the `map` method, for example, you can perform many, many more for loops than you can `map` operations in the same time period. But I still argue that you should go ahead and start using all of these methods today whenever it seems to make sense for the type of code that you're writing.

One of the first reasons is because the new engines that are being written are going to be improved with these methods in mind. Secondly, if you think about the way that your code is being used, I would bet you that most of the time, the bottleneck is not in your loop.

And the only way to find out whether or not it is, is to run your code in production and see what actually happens, find out whether or not your performance is being significantly affected by the loop itself. You should always write your code for optimal cleanliness, readability, and maintainability.

Remember that the next person who inherits your code could be you six months down the road, and you want to be able to step into your code and understand what it means and how to modify it effectively. And eventually, if you do need to optimize it for performance, you should only do that when a real world situation actually calls for it.

If you look at your statistics and you can see yes, my code is being bottlenecked and the bottleneck is in the loop, then you'll know that that's the place where you need to go in and fix your code. And you can always keep that in your back pocket while maintaining the cleanest code possible until you need to make that modification.

And when I say that this can actually make your code look much cleaner, I've only been showing you isolated examples of map, reduce, and filter. But let's imagine if we're trying to solve a problem that we needed all three of these methods to solve effectively. For example, what if we wanted to create a StudlyCapped string of three-letter words from an array of strings.

The StudlyCaps, in this case, are capitalized letters at the beginnings of each word and all concatenated together into a single word. Let's see how we might need to do that if we didn't use any of these functional methods on the array, and just used more traditional vanilla JavaScript.

```
const animals = ["cat","dog","fish"];
let threeLetterAnimalsArray = [];
let threeLetterAnimals;
```

```
for (let count = 0; count < animals.length; count++){
  let item = animals[count];
  if (item.length === 3) {
    item = item.charAt(0).toUpperCase() + item.slice(1);
    threeLetterAnimalsArray.push(item);
  }
}
threeLetterAnimals = threeLetterAnimalsArray.join("");
console.log(threeLetterAnimals); // "CatDog"
```

In fact, I had to cut a few corners just in order to make my code a little bit shorter. Like, for example, not defining `animals.length` as a separate variable, but instead having it calculated every single time that it went through the loop. I had to create a bunch of extra variables along the way, like `threeLetterAnimalsArray` and `threeLetterAnimals` and of course the `count` and the `item`. I embedded some custom functionality, in this case the functionality that capitalizes the strings, and left it here sitting inside of our `for` loop, where it can't be used by anything else and it couldn't be inherited from another source. In addition, I had to create this extra operation at the end for joining our values together, which really I didn't want to have to do.

So, how could I have done this better using functional methods? Well, the first thing that I always want to do when I'm looking at converting code to functional code is to pull out the pure functions. And analyzing this code, I can see some places where I can extract pure functions out of our code, and then build up a new function that takes advantage of them.

```
const animals = ["cat","dog","fish"];
const exactlyThree = word => word.length === 3;
const capitalize = word => word.charAt(0).toUpperCase() + word.slice(1);
const mergeWords = (words, word) => words + word;
const getStudlyCaps = words => {
  let threeLetters = words.filter(exactlyThree);
  let capitalized = threeLetters.map(capitalize);
  let merged = capitalized.reduce(mergeWords);
  return merged;
}
console.log(getStudlyCaps(animals)); // "CatDog"
```

I'm going to take this one step further because at this point we still have a lot of intermediate variables being created inside of our `getStudlyCaps`. Even though they are contained inside of that block and so they don't pollute anything outside of that function, we can do a little bit better. Fortunately, the new functional methods on array are also chainable. Which means that we can take `map`, `reduce`, and `filter`, and we can chain them, connecting them with dots to pass their return values along and not need any intermediate variables along the way.

```
const animals = ["cat","dog","fish"];
const exactlyThree = word => word.length === 3;
const capitalize = word => word.charAt(0).toUpperCase() + word.slice(1);
const mergeWords = (words, word) => words + word;
let threeLetterAnimals = animals.filter(exactlyThree).map(capitalize).reduce(mergeWords);
console.log(threeLetterAnimals); // "CatDog"
```

The value of these new methods isn't so much that they're changing or introducing new functionality that JavaScript never had before. What they're providing is another way that you can approach the exact same problems to present them in cleaner, easier to manage code. And they're versatile enough, the chaining can be formatted the way that you like, you can use intermediate methods, you can use intermediate functions, whatever you prefer.

The key is to be able to use these new functional methods to your advantage to make your code cleaner and more maintainable. So the next time that you step into it, you'll be able to understand it better and the next person who needs to work with it will be able to understand it just as well.

And now that we've gone over mapping, reducing, and filtering and you have a very good understanding of how those work, I'm going to introduce you to another functional concept in the next lesson and that's composition.