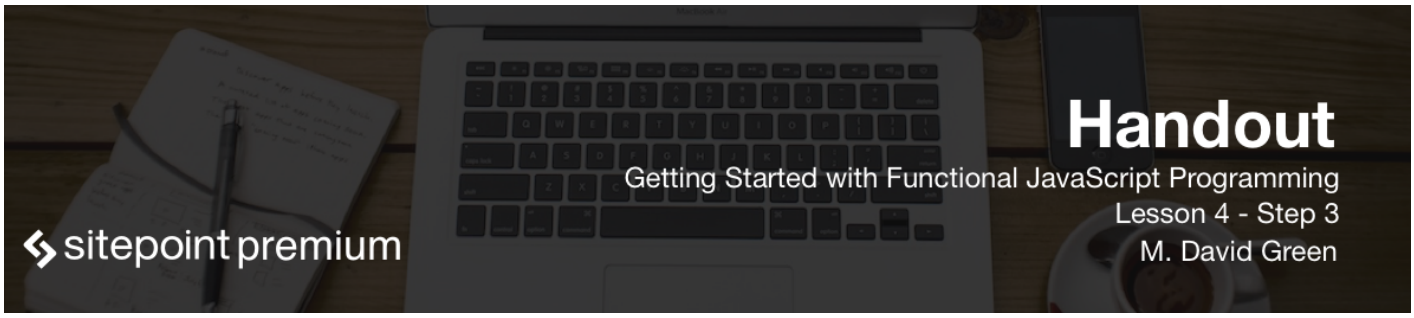

The next functional method for arrays, that was introduced with ECMAScript 5, is the `reduce` method which takes all of the values in an array, goes through them one by one, and then produces a single result, based on a calculation performed across the series, while carrying a value forward.

Reducing an array to produce a summary value operates on each item of the array, and maintains a running total while you're going. You can optionally set an initial value for that running total. When it's completed, reduce returns a new value derived from the entire set of operations, leaving the original array completely unchanged.

And of course, like all functional methods, you don't want to alter the state of anything outside of the array that's being operated on. You don't want to make changes to that array. And you don't want to rely on any values in the state of the application that you're operating in.

Reducing is a little bit more complicated than mapping, but I think it's easy enough to understand when we get an example. And I think a good example of how to use reduce would be getting a total character length count from an array of strings. And I'll show you how that would look using a `for` loop to start with.

```
const animals = ["cat","dog","fish"];
const animalsLength = animals.length;
let letterCount = 0;
for (let count = 0; count < animalsLength; count++){
  letterCount += animals[count].length;
}
console.log(letterCount); //10
```

Now the same example with the `reduce` method:

```
const animals = ["cat","dog","fish"];
let letterCount = animals.reduce((sum, item) => sum + item.length, 0);
console.log(letterCount); //10
```

How did we work this magic? The `reduce` method helped us. We were able to use an anonymous inline function that took a sum and an item, in this case each item being the item in the array, and the sum being the value that we want to carry forward. It simply returned that sum plus the length of that item. We set the initial value of that sum to 0 as our initial accumulator value as part of our definition of the result. Reduce went through the entire array, looked at each item, added that item's length to our sum, and kept carrying the sum forward.

When we were done, that sum was the value assigned to our letter count. And one of the interesting things about reduce is that accumulator value that we said at the end is actually not mandatory. You can legally call the reduced method without setting an initial value for the accumulator.

But thanks to type coercion and operator overloading in JavaScript, that final value actually sets the value and the type for the running total to start with. So it's optional, unless you need to control the type or the value at the beginning of your operation in order to make sure that you get the result you're expecting.

And in most cases, it's a good idea to set it. For example, let me show you what happens if we don't set it in this example that we used:

```
const animals = ["cat","dog","fish"];
let letterCount = animals.reduce((sum, item) => sum + item.length);
console.log(letterCount); //"cat34"
```

Why do we get *cat34*? It's because of the way the JavaScript overloads the addition operator. The addition operator can be used to add numbers together if all the values are numbers.

But if one of the values is a string, the addition operator is going to treat the result as if it's going to be a string. So in this case, because we didn't set an accumulator in the first place, our accumulator assumed that the first value in the array should be the value set for the sum.

And because the first value was a string, it assumed that the plus operator should be used to add together string values and return a string. So the *cat* was assigned as the first value, then the length of *dog* was added and then the length of *fish* was added, giving us *cat34* - certainly not the result that we were expecting.

When you're looking at `reduce` for the first time, it can be a little complicated figuring out how to read what you're seeing. And for that reason, our trick of pulling out the anonymous inline function and giving it a name, and then passing it to `reduce`, can make this `reduce` a little bit easier to read.

```
const animals = ["cat","dog","fish"];
const addLength = (sum, item) => sum + item.length;
let letterCount = animals.reduce(addLength, 0);
console.log(letterCount); //10
```

Here we're just defining our function and we're calling it `addLength`. It doesn't rely on anything in the state of the application and it always returns the same value for the same item that's being passed in. We're passing that function into the `reduce`, and setting our initial accumulator value because that helps us make sure that the return type is what we're expecting.

So keep in mind that a good reducing function is always pure. It takes two arguments, an accumulator, and a value. And it operates on that value without producing any side effects. And it should always produce the same result for the same input values, reduce returns and accumulated value that's ready for a subsequent operation, or that can simply be returned.

And ideally, the functions that you pass into reduce will be pure, which means that they won't alter variables outside of their own scope. And they could be used without modification in other contexts, making them much more versatile.