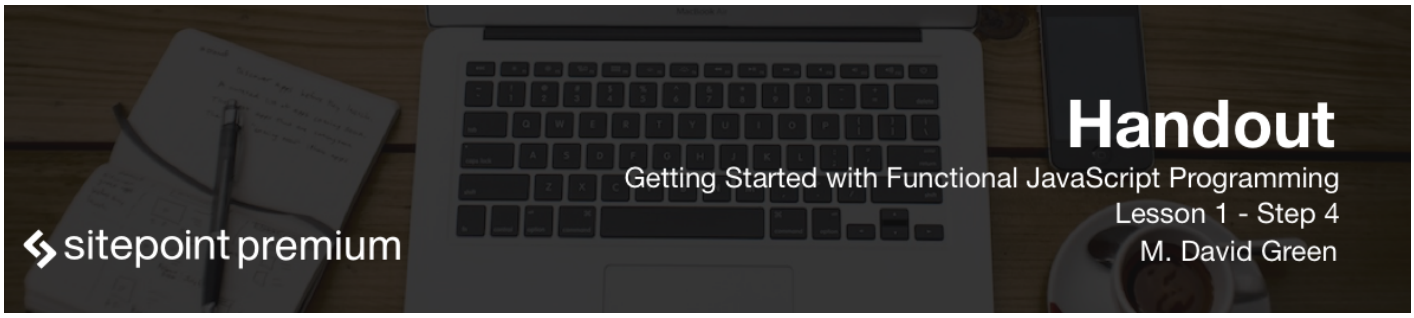

JavaScript, as I've mentioned before, is a very versatile language. One of the things that makes it so versatile is that it can be used in an imperative style, it can be used in an object oriented style, and as you're about to see, it can also be used in a functional style.

That doesn't mean that any of these is necessarily the perfect solution for every problem. But it's good to know what the options are and it's good to know how to use them, and that's what this course is all about.

So what is functional style anyway? First of all, functional style takes advantage of the fact that JavaScript allows you to use functions as first-class objects. Which means that you can use them as arguments and you can pass them around from function to function, just as you can any other variable. Functional style also encourages you not to make changes to values that are outside of your functions or change the state of the application while you're executing the function.

Ideally, with a functional solution, what you're going to describe is the problem or the solution, not the step by step way of getting that problem solved. Here is an example:

```
(function() {
  "use strict";
  var capify = function(str) {
    return [str.charAt(0).toUpperCase(), str.substring(1)].join("");
  };
  var processWords = function(fn, str) {
    return str.split(" ").map(fn).join(" ");
  };
  var getValue = function(e) {
    var something = prompt("Give me something to capitalize");
    alert(processWords(capify, something));
  };
  document.getElementById("main_button").addEventListener("click", getValue);
}());
```

What is better about this example? You can see right away that the code is much more concise, which makes it a little bit easier to read and easier to reason about too. The functions here are defined in an independent and reusable way. Our `capify` function could be dropped into any place that you need to pass in a string and capitalize the first letter of that string. Our `processWords` function doesn't depend on `capify`. It depends on whatever function you happen to pass in that performs a function on a single word. There's also no reliance on some abstract object being passed around. Nothing is happening outside of these functions that's changing as a result of what's inside the functions. The functions take a value, process it and then return it so that whatever needs to happen next can happen next.

As a result, this code is much easier to unit test, because you can write a simple test that verifies that each of these functions does what you expect it to do with a variety of inputs. And those functions should behave that way regardless of what context they're in.

You can also notice that we used the new `map` method, and that's actually something that we've had since ECMAScript 5. In functional programming, map allows us to create cleaner functions that don't rely on a lot of extra variables and `for` loops.

Now that you've seen the ECMAScript 5 version and we've walked through it, I'd like to show you just how tight this example can get using ECMAScript 2015 syntax. If you are working in JS Bin, you'll need to select the Babel processor. We do it to avoid any syntax highlighting errors that would show up when we use ES 2015 syntax in this interface. In your browser, and in most modern JavaScript environments, ES 2015 is supported natively.

```
{
  "use strict";
  const capify = str => [str.charAt(0).toUpperCase(), str.substring(1)].join("");
  const processWords = (fn, str) => str.split(" ").map(fn).join(" ");
  const getValue = () => {
    let something = prompt("Give me something to capitalize");
    alert(processWords(capify, something));
  }
  document.getElementById("main_button").addEventListener("click", getValue);
}
```

The `{ }` is a block that, when encountered, will immediately execute just like an immediately invoked function expression.

So after seeing all of those examples, what we end up with is code that's this short, this concise, and does everything that we wanted to do. And what's different about this code compared to what we've done before? Well, first of all, it is certainly extremely concise. Everything fits into essentially four lines of code. There are no mutable variables, we haven't used the `var` keyword anywhere, and we didn't need to because we were able to define everything as constants that we didn't need to worry about changing. We defined our functions with implied return values. We didn't actually have to use the `return` keyword.

The block syntax provided us with the value of an immediately invoked function expression. And we did still use strict and because we're using things in a modular way, we can limit the effective use strict to the code that we're writing. What we have here is a very dense code, which may be a little bit harder to scan and understand depending on your familiarity with ECMAScript 2015 syntax.

You may choose to use a little bit of one and a little bit of another, fortunately it's mix and match. And defining our functions as anonymous arrow functions also may make them a little bit harder to debug than defining them as functions. Another thing that comes up with ECMAScript 2015 code is that because it's so concise, it may not be as self-documenting.

It might not be as clear exactly what you're trying to accomplish. And finally, if you're using this code in the context of older code, for example you're updating a code base, it's more difficult to integrate ECMAScript 2015 code into existing code that might have been written according to ECMAScript 5 standards. You might prefer the fact that your ECMAScript 2015 code stands out and looks different. On the other hand, you might want your code to fit in and blend more appropriately with the code that you've written before. Ultimately, you need to choose your own approach to the way that you write your code.

The imperative code is very easy to reason about and to see from top to bottom what's happening. The object-oriented code makes a lot of sense to people who think in terms of objects and how they can pass around data. Functional code, using pre-ECMAScript 2015 syntax, is self-documenting and very concise. But for maximum conciseness you can use

functional code with ECMAScript 2015 syntax and get very tight, very clean results. For this course, we're going to be using a mix but we are gonna be focusing on the ECMAScript 2015 way of creating our functional code. So it's worth learning and understanding.