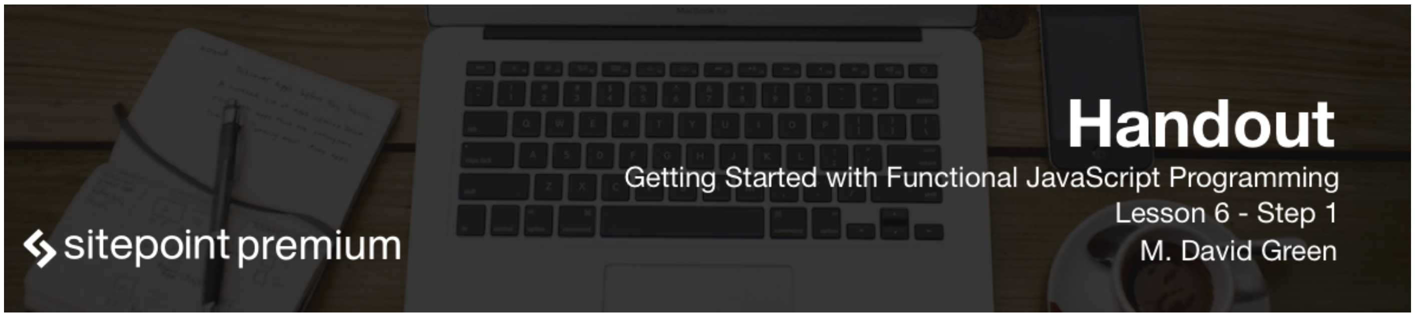

Welcome back to the 6th and final lesson in Functional Programming in JavaScript from SitePoint. In this lesson, I'd just like to go over some of the things that we've learned, focus your attention on some of the key takeaways, and discuss how you can take this to the next level in your own code.

The goal of this course has been to provide you with an overview of functional techniques that you can start using in your JavaScript code today. I wanted to present the opportunity for you to start thinking about functional programming not as something distant and philosophical, but rather as something practical that you can start using.

I wanted to provide you with a few key functional techniques in JavaScript that you can start using right away. Most important, I want you not to be afraid of the new terminology that comes from functional programming. Just because you haven't heard of these things before, doesn't mean that they're not practical and useful. After taking this course I hope you'll take the opportunity to look at your code and try doing some things differently, just to see how they work for you. You might go back to the way you've always done it, or you might find something new that's going to be useful going forward.

So some of the things that we learned in this course were how pure functions work. Pure functions don't rely on the state of the code that they're called from. They don't create side effects that alter the variables outside of themselves. And they return one, and only one, result for a given set of arguments. You don't want to be writing impure functions, and that's just good programming technique regardless of whether you use anything else from this functional programming course.

We also talked about recursion, how you can eliminate loops to make your code cleaner. In this case, we can have the factorial loop that we demonstrated that doesn't use any for loops, doesn't use any internal variables, doesn't use any state, and simply returns the result.

Recursion depends on defining a terminal condition, and then making a recursive call to the parent that will keep operating until it reaches its terminal state. You read that recursive call as if it were the return result of everything you've passed in. With a reminder to structure your code for proper tail calls and pay attention to entail call optimization makes it out to the browsers, and to the other JavaScript engines that you're going to be using.

We also talked about currying, which is using partial application to create sets of related functions that can share code but they can provide customizable features.

We also talked about mapping, which allows you to perform an operation on each element of an array and return a new modified array that's exactly the same length.

We talked about reducing, which produces a summary result based on the values in an array, and an optional but recommended accumulator value to start things off.

We also talked about chaining methods, and how you can produce a result from sequential methods without using intermediate steps.

Finally, we learned about composition, which allows us to build up complex functionality from simple pure functions. We created our own `compose` utility and demonstrated why it's important to keep track of the order in which you pass in your functions.