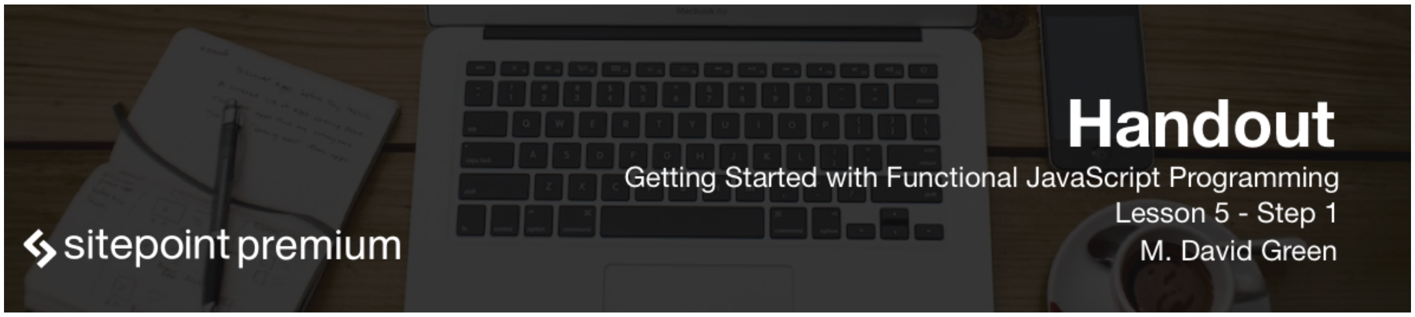

Handout

Getting Started with Functional JavaScript Programming
Lesson 5 - Step 1
M. David Green

sitepoint premium

Welcome back, this is Lesson 5 of Functional Programming in JavaScript from SitePoint. In this lesson, we're going to talk about **composition**. Now we've already talked about a few different aspects of functional programming, but composition takes it to the next level, and looks at how you can build up new functionality using small pieces of existing functionality.

And in order to do that, you're going to want to nest your functions which is something that JavaScript allows you to do, and probably something that you've already been doing. In fact, it's something that's been hidden inside of some of the examples we've already talked about. We know that JavaScript allows you to use a function to call a function.

Higher order functions can take a function as an argument, or return a function. And as an example, we took a look at the `setTimeout` and `setInterval` functions. Each one of these can take either an anonymous inline function or a named function, plus a second argument to control when that function will be executed.

```
const logTime = () => console.log(new Date().toLocaleTimeString());
const timer = setInterval(logTime, 1000);
setTimeout(() => clearInterval(timer), 3000);
```

Function methods can also be changed, the way that we demonstrated in the last lesson about `map`, `reduce`, and `filter`. You can take a series of pure functions that return a value, and each one of them can perform its function independently and pass its return value on to the next one, so that the sequence of functions can be called all in a single line.

```
const animals = ["cat","dog","fish"];
let threeLetterAnimals = animals.filter(exactlyThree)
                                .map(capitalize)
                                                                .reduce(merged);
```

But, in addition, JavaScript allows you to pass functions to other functions using nested parentheses. And when a function followed by parentheses is being passed to another function, it's not passed as the function, but rather as the return value of performing that function. Let me demonstrate what that looks like.

```
const addOne = x => x + 1;
const timesTwo = x => x * 2;
console.log(addOne(timesTwo(3))); //7
```

Even in an extremely simple example like this, the order of the nesting matters. Compare the results from these examples:

```
const addOne = x => x + 1;
const timesTwo = x => x * 2;
console.log(addOne(timesTwo(3))); //7
console.log(timesTwo(addOne(3))); //8
```

So when it comes to nesting functions, as I said, the most important thing to keep in mind is that the order matters. The operation in the innermost parentheses is always going to be performed first. And nesting has no limits in JavaScript. What we've done here is essentially composed new functionality within JavaScript by nesting existing functions.

We can keep nesting just as far as we want, as long as we follow a few simple rules. We want each nested function to be a small pure function. We want to make sure that they pass their return values. And we want to make sure that we respect the order in which these operations are going to be performed. As long as we do that, we can nest to our heart's content:

```
console.log(timesTwo(addOne(addOne(timesTwo(3))))); //16
console.log(addOne(timesTwo(timesTwo(addOne(3))))); //17
```

JavaScript will let us nest our functions just as deeply as we want. But where does that get us? What can we do with that in a functional programming paradigm in order to make JavaScript's code a little bit cleaner? Maybe get rid of some of those extra parentheses.