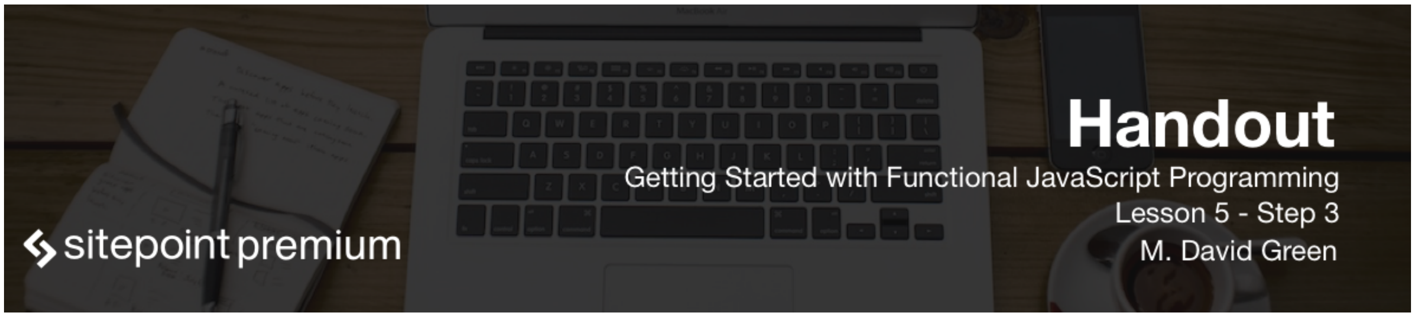

Handout

Getting Started with Functional JavaScript Programming
Lesson 5 - Step 3
M. David Green

Based on what we've seen already, it's not too difficult to conceive of what a compose utility would look like, and how you might construct one. So, let's create our own basic `compose` function. It's going to need to take two functions as arguments, and then return a composed function that nests them. The function that it returns is going to have to take a value that we can pass in.

```
const compose = (f1, f2) => {
  return value => {
    return f1(f2(value));
  };
};
```

So in defining our `compose` function, we've made some decisions. For example, we decided that the second function passed in to our compose function was going to be the inner function, the one that's going to be performed first. That's typical of the way that you do things in functional programming.

The last argument you pass in is the one that's most likely to change, so it should be the first one that operates on the value. The result of that will be passed to the outer function, which is the first one that we pass in. And although we've structured it this way for clarity, we could have also written our compose function as a one liner.

So let's test our compose function, and see just how it works.

```
const addOne = x => x + 1;
const timesTwo = x => x * 2;

const compose = (f1, f2) => {
  return value => {
    return f1(f2(value));
  };
};

const addOneTimesTwo = compose(timesTwo, addOne);
console.log(addOneTimesTwo(3)); //8
console.log(addOneTimesTwo(4)); //10
```

In fact, we can create a lot of variations using just one line for each one, without writing a lot of redundant code.

```
const addOneTimesTwo = compose(timesTwo, addOne);
console.log(addOneTimesTwo(3)); //8
console.log(addOneTimesTwo(4)); //10

const timesTwoAddOne = compose(addOne, timesTwo);
console.log(timesTwoAddOne(3)); //7
console.log(timesTwoAddOne(4)); //9
```

The point here is not that you're going to want to add one and times two all the time in your code. You're going to be wanting to do much more complex things that are more relevant to your own application, but I'm trying to keep the concepts here simple so you can focus on how `compose` is working. You're passing in functions and composing them in a consistent order. So you can always be sure what kind of result you're going to get, and you're not having to use a lot of redundant code or build with a lot of deeply nested parentheses.

So, things to keep in mind when you're composing, of course the order matters. The order of the functions being passed in is relevant to the results that you're going to get. Keep track of the return types that you're working with as well because every inner function is returning a value that's going to be operated on by the outer functions.

Those outer functions need to be expecting the return value that comes from the inner functions. Don't try to use compose with impure functions. You want to make sure that your functions always return the same value regardless of the state outside of them, and that they don't make any changes to the state outside of themselves.

Think about also who needs to understand your code, because somebody is going to be inheriting your code. Probably yourself in six months as I've said before. And you want to be sure that it's clear and understandable. And honestly, you're probably going to want to use a library to bring in a robust and well maintained compose function, instead of using one that you've built yourself.