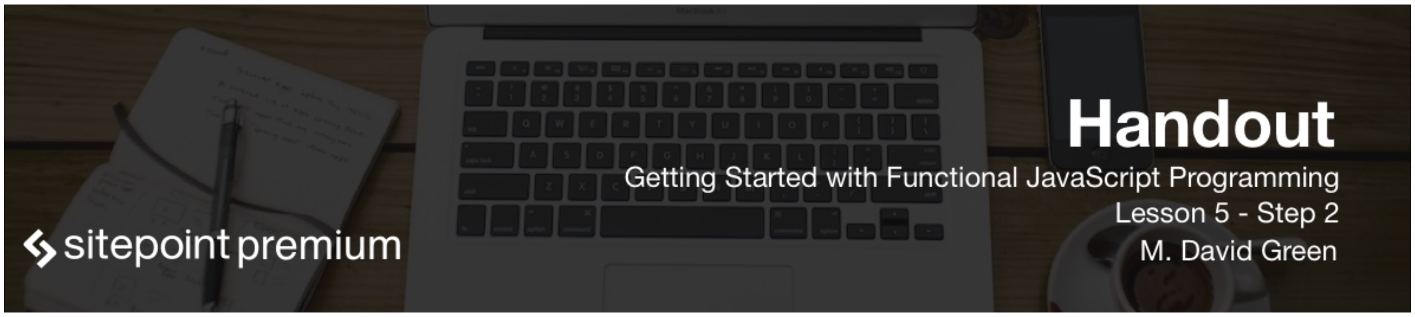

Since this is a course in functional programming, the first thing that might occur to you after seeing how JavaScript can nest all of those functions, is manual composition, defining new functions based on nested versions of existing functions. So you could define a new function that behaves exactly like a set of nested function, and use smaller pure functions as components.

That might look something like this:

```
const addOne = x => x + 1;
const timesTwo = x => x * 2;
const addOneTimesTwo = x => {
  let holder = x;
  holder = addOne(holder);
  holder = timesTwo(holder);
  return holder;
}
console.log(addOneTimesTwo(3)); //8
console.log(addOneTimesTwo(4)); //10
```

So we've defined a new function that allows us to quickly and easily call this nested function that we were creating before, without a bunch of extras parentheses and a bunch of extra nesting. But along the way, we've created an intermediate local variable. And if you've been following this course from the beginning, you know that we don't like state in functional programming. We try to avoid intermediate variables anywhere we can. So let's see if we can figure out a way that we can eliminate that intermediate variable.

Since we've already been talking about nesting, let's try using that instead.

```
const addOne = x => x + 1;
const timesTwo = x => x * 2;
const addOneTimesTwo = x => timesTwo(addOne(holder));
console.log(addOneTimesTwo(3)); //8
console.log(addOneTimesTwo(4)); //10
```

Doing this provides us with much more compact code, and it does eliminate that `holder` variable. But our code risks getting more complex as we add more levels of nesting. So we probably want to stick with the longer syntax in order to make it easier to manage and maintain our code in the future.

That way we can create a number of variations using this longer cleaner syntax. We just mix and match smaller functions as we need them.

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

const addOneTimesTwo = x => {
  let holder = x;
  holder = addOne(holder);
  holder = timesTwo(holder);
  return holder;
};

console.log(addOneTimesTwo(3)); //8
console.log(addOneTimesTwo(4)); //10

const timesTwoAddOne = x => {
  let holder = x;
  holder = timesTwo(holder);
  holder = addOne(holder);
  return holder;
};

console.log(timesTwoAddOne(3)); //7
console.log(timesTwoAddOne(4)); //9
```

So we're able to create as many variations as we like, but we're kind of trading complexity for configurability. We still have that intermediate variable every time, and there is a lot of repetitive code between these two functions. What we really want is a compose function. And a compose function, which is not currently native to JavaScript, could take two functions and then nest them producing a new function.

That new function would be the result of applying the compose function, and then passing in the first function and the second function. And then the tricky part is the order in which the operations would be performed. Because of course, as we know, when you're nesting functions the order is very critical.

So let's talk a little bit about what a compose utility might look like.