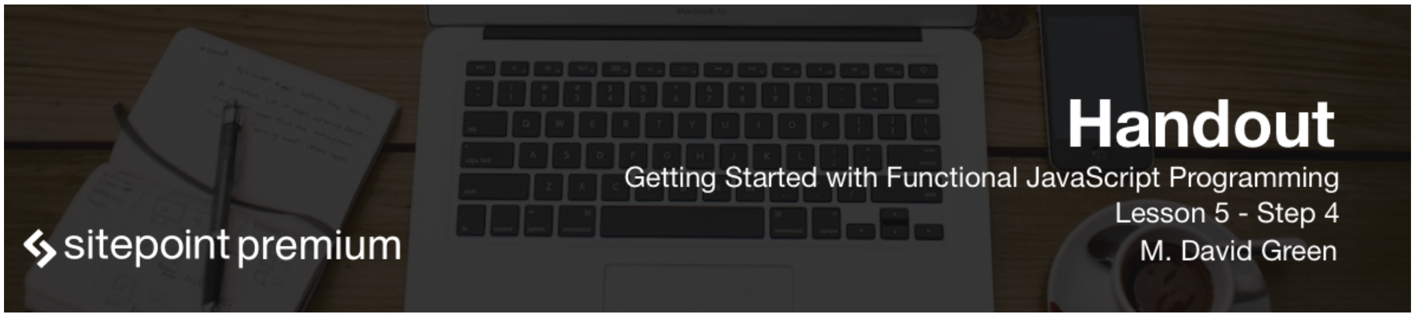

Handout

Getting Started with Functional JavaScript Programming
Lesson 5 - Step 4
M. David Green

At the end of the last section, I recommended using a library to bring in a `compose` function that's well maintained, instead of using the little one that we built ourselves. To be perfectly honest, there were some problems with the simple compose that we created. First of all, it doesn't handle nested functions that take multiple arguments, and the ability to support variadic functions is an important aspect of composing.

To demonstrate what I mean, let's take our `compose` function and our `addOne` function, which only takes one argument, and let's create a new function:

```
const compose = (f1, f2) => {
  return value => {
    return f1(f2(value));
  };
};
const addOne = x => x + 1;
const divide = (x, y) => x/y;
const divideAndAdd = compose(addOne, divide);

console.log(divideAndAdd(4,2)); // NaN
```

We get "not a number". JavaScript isn't able to deal with this, and it returns a value that's not actually a number.

If you look at the structure of how our compose was written, you can see that it's only designed to accept a single value. If we were trying to pass in multiple values, we'd have to do something more complex to the way that this is structured, and then we'd only be able to accept two values.

There are other problems with our simple `compose`: for example, it doesn't handle more than two functions.

```
const addOneThreeTimes = compose(addOne, addOne, addOne);
console.console.log(addOneThreeTimes(4)); //6
```

In this case, JavaScript hasn't returned an error, it simply returned the wrong value. This is because JS ignores extra arguments when you're creating a function. In our `compose` function, the last `addOne` was just ignored.

There are other problems with our compose as well. I'm a big fan of not reinventing the wheel as long as you understand exactly what you're inheriting. So now that you know how `compose` works, I would recommend adopting a functional programming library that includes a robust `compose` method that you're comfortable working with. For example, this is the `compose` method inside of Underscore:

```
const compose = (f1, f2) => {
  return () => {
    return f1.call(this, f2.apply(this, arguments));
  };
};
const addOne = x => x + 1;
const timesTwo = x => x * 2;
const divideAndAdd = compose(addOne, divide);
console.log(divideAndAdd(4,2)); // 3
```

And there are similar compose functions in other libraries. So whether you go with Ramda, or you go with Underscore or you go with Lodash, just consider the size, support, and how the other methods behave inside of a library.

Using a library is often as simple as just including it in your code, and then referencing it appropriately. And ultimately, once you’ve understood how functional programming works, and you know which utility methods you want, choosing a library is the best path to follow.