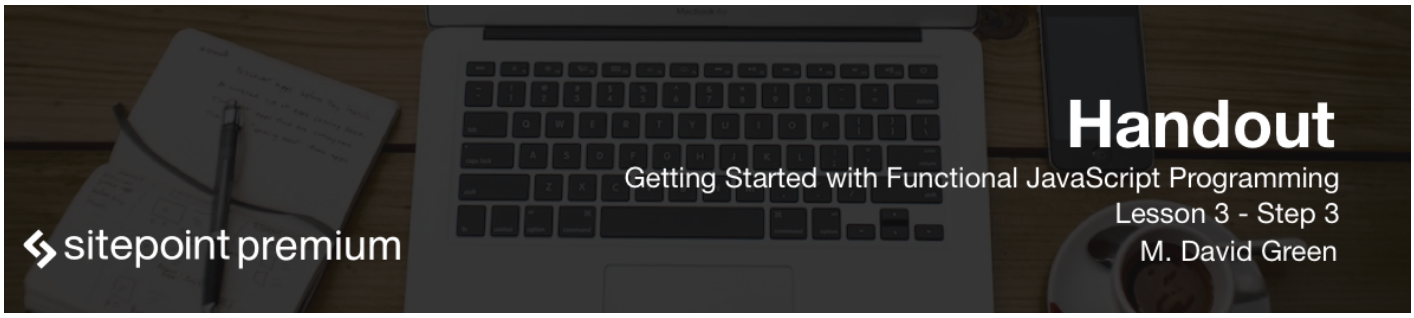

Now that we've been introduced to how currying works for a simple function which has a single argument being passed in. Let's imagine how we could do deep currying for a variadic function, a function with more than one argument to be passed in. Let's think about how we specific we can get with our greeting function.

What if we wanted the ability to customize every single element of our greeting from the name, to the punctuation, everything? So imagine we've got a greeting such as "Hello, Heidi". We've got the word hello, we've got a comma, we've got Heidi, and we've got a period. What if we wanted to be able to change the greeting, change the comma, change the name, and change the punctuation at the end?

Let's think about how we might typically do that with JavaScript. You can imagine writing a function called `greet`. And that greeting would take multiple arguments. It would take a greeting, it would take a separator, it might take an emphasis mark for the punctuation, and then, of course, it would take the name of the person that you're going to greet.

Notice that I put the name at the end. That's going to be important later. Anyway, our function would then just return the aggregate of all of those things. But the question is how could you curry this? How could you make it so that you can pass in the argument that you want, and then add all of the other arguments as you need them to define multiple functions that all provide a different type of greeting?

Technically in functional programming, the definition of currying limits us to working with one argument at a time. But what we've shown you is also applicable to a concept called partial application. Unlike currying which produces a series of chained functions each one taking a single argument. With partial application, we take a function and apply it to some of its arguments but not all of them, producing a new function in the process.

This is similar to the way that JavaScript uses `bind`, although bind requires passing in a context. Using a currying function for partial application allows us to just pass in the function that we want this applied to. And we can approach it the same way that we do currying.

Applying our currying technique to variadic functions, which take more than one argument. In this case, just to demonstrate how it would work, I'm going to leave out the type checking for brevity but you know from our previous example that it's easy to add that.

```
const greetDeeplyCurried = greeting => separator => emphasis => name => {
  return greeting + separator + name + emphasis;
};

const greetAwkwardly = greetDeeplyCurried("Hello")("...")("?");
console.log(greetAwkwardly("Heidi")); //"Hello...Heidi?"
console.log(greetAwkwardly("Eddie")); //"Hello...Eddie?"
```

So what we have here is a bunch of nested functions, sort of like matryoshka dolls, each one returns another function that takes a different argument, and passing in each one of those arguments independently allows us to customize every element.

The point is that we've got functions, returning functions, returning functions, returning functions, finally returning a value.

And you notice again in this case, the name was passed as the final function. And I told you that there was a reason we were doing that. And it's so that when we define our final function, `greatAwkwardly`, it can take the greeting, the separator, and the emphasis, and wait for the name at the end.

Because we've decided that the name is the thing that's most likely to change. So when we defined our Curried function, we defined it with the name as the final argument. That way, each of the subsequent functions can be defined with different greetings, different separators, different emphasis, and each one can take a new name every single time you run it.

We can also pass fewer arguments when we're creating our variations on the curried function. Any arguments that we don't pass when we're defining our functions can be passed when we call those functions. The important thing, again, is just to respect the order. So again using our `greetDeeplyCurried`, we could define another child function, we call it `sayHello`.

```
const sayHello = greetDeeplyCurried("Hello")(", ");
console.log(sayHello(".")("Heidi")); //"Hello, Heidi."
console.log(sayHello(".")("Eddie")); //"Hello, Eddie."
```

Defining our child functions this way gives us a lot of versatility, because we can retain a lot of the customization that we want if we can still keep all of the type checking or whatever it is that we want in our main curried function, in one place, so we're not repeating a lot of code.

And we can create as many new customizations as we want.

```
const sayHello = greetDeeplyCurried("Hello")(", ");
console.log(sayHello(".")("Heidi")); //"Hello, Heidi."
console.log(sayHello(".")("Eddie")); //"Hello, Eddie."

const askHello = sayHello("?");
console.log(askHello("Heidi")); //"Hello, Heidi?"
console.log(askHello("Eddie")); //"Hello, Eddie?"

const yellHello = sayHello("!");
console.log(yellHello("Heidi")); //"Hello, Heidi!"
console.log(yellHello("Eddie")); //"Hello, Eddie!"
```