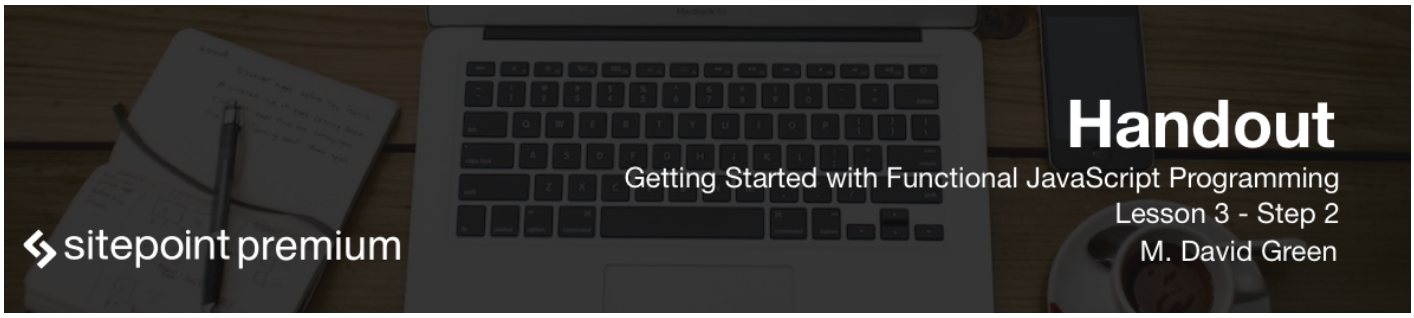

In the last section I gave you an example of the type of problem that currying is great at solving. We were trying to create a greeter that could send different types of greetings to different types of people, and yet maintain some consistency across multiple functions that tried to do the same thing.

So let's curry our `greet` function and see if we can provide a simple curried function that returns a function waiting for a missing argument.

```
function greetCurried(greeting) {
  return function(name) {
    if (typeof(greeting) != "string") {
      return ("Greetings!");
    } else if (typeof(name) != "string") {
      return (greeting);
    }
    return (`${greeting}, ${name}`);
  }
}
```

All we're doing is creating a function that asks for a greeting, and returning a function that asks for a name. All of these conditionals in the middle are just the type checking that we had in our previous versions.

Ultimately, the point of the internal function is to simply return "greeting, name". And to demonstrate how we can make use of our curried function, we can return a function waiting for a missing argument, and pass in the greeting, such as, Hello. In this case, we can define a function that will be a child function of our greetCurried.

```
const greetHowdy = greetCurried("Hello");
console.log(greetHello("Heidi")); // "Hello, Heidi"
console.log(greetHello(5)); // "Hello"
```

So what's cool about this is that we have all of our type checking in our `greetCurried`. And in order to create a custom version, `greetHello`, all we have to do is pass the string hello to `greetCurried`, and assign that to a new constant, and that becomes the function that we execute, with the name that we want to send a greeting to.

But if all we wanted to do was send the greeting hello, we wouldn't really be seeing the value of currying. The point is now it's easy for us to create a whole set of functions that will share this code, but they each can provide custom functionality.

```
const greetHi = greetCurried("Hi");
console.log(greetHi("Heidi")); // "Hi, Heidi"
```

```
console.log(greetHi(5)); // "Hi"

const greetHowdy = greetCurried("Howdy");
console.log(greetHowdy("Heidi")); // "Howdy, Heidi"
console.log(greetHowdy(5)); // "Howdy"
```

Now let's create a small function called `greetWrongType`:

```
const greetWrongType = greetCurried(5);
console.log(greetWrongType("Heidi")); // "Greetings"
console.log(greetWrongType(5)); // "Greetings"
console.log(greetWrongType(false)); // "Greetings"
console.log(greetWrongType(null)); // "Greetings"
```

We can see from this how easy it is to use currying to create a suite of related functions as long as each of those functions only takes one argument. But it's also possible to do deep currying with variadic functions that can take multiple arguments and I'm gonna show you that in the next section.