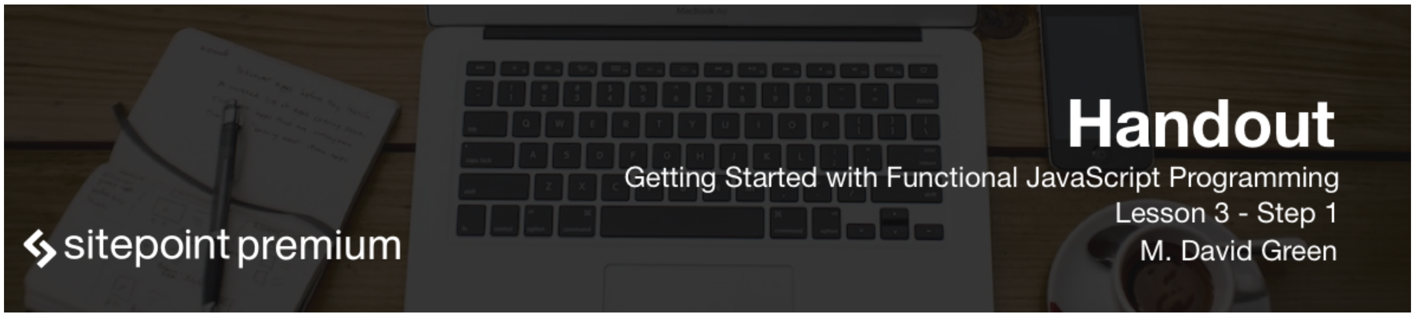

Welcome back to functional programming in JavaScript, lesson three. In this lesson we will discuss **currying**.

So what does currying do? Well, currying is a way to consolidate the functionality of multiple functions that may all share similar code, but may take different arguments. With currying, you can create that code in one place and then, pass in different arguments, and define separate versions of that function.

Each of those versions can take different arguments and be expecting different arguments, but they can all share the same base code functionality. That helps you avoid repeating yourself so your code doesn't become stale, and lets you make updates to a whole series of functions that may be related by making one change in one place.

Let me show you what this means. You might find yourself in the situation where you're calling a function with repetitive arguments. And every single time that you call that function you have to pass all of the arguments, even though most of these arguments, or some of these arguments or maybe even just one of these arguments, may be the same every single time that you call it.

Let's imagine that you're writing a function that's supposed to greet people, and every time that you want to greet somebody you pass in the greeting that you want to send and the name of the person that you want to greet:

```
function greet(greeting, name) {
  return (`${greeting}, ${name}`);
}
console.log(greet("Hello", "Heidi")); //"Hello, Heidi"
console.log(greet("Hello", "Eddie")); //"Hello, Eddie"
console.log(greet("Hello", "Barbara")); //"Hello, Barbara"
```

You can see we're getting into a situation where we're basically building up the same thing over and over. So look at this code and see what we're trying to do here. The first thing you may think is that you might want to get rid of that *hello* word.

```
function greetHello(name) {
  return (`Hello, ${name}`);
}
console.log(greetHello("Heidi")); //"Hello, Heidi"
console.log(greetHello("Eddie")); //"Hello, Eddie"
console.log(greetHello("Barbara")); //"Hello, Barbara"
```

But what do we actually gain by doing that? In this case we don't have to send as much in the arguments, but we've created this independent function that's less versatile than our original function. All that it can do is say hello to whoever's name we pass in. And versatility is one of the reasons that we use programming languages, so what if we wanted different types of greetings? We would have to write separate independent functions for each one.

```
function greetHi(name) {
  return (`Hi, ${name}`);
}
console.log(greetHi("Heidi")); //"Hi, Heidi"
console.log(greetHi("Eddie")); //"Hi, Eddie"
console.log(greetHi("Barbara")); //"Hi, Barbara"

function greetHowdy(name) {
  return (`Howdy, ${name}`);
}
console.log(greetHowdy("Heidi")); //"Howdy, Heidi"
console.log(greetHowdy("Eddie")); //"Howdy, Eddie"
console.log(greetHowdy("Barbara")); //"Howdy, Barbara"
```

Look at all of this repeated code we've got. In addition to being painful to look at, we're creating a maintenance nightmare out of this. We have so many different functions to maintain, just to be able to provide a few different greetings to a few different people. And in case I haven't convinced you yet, what if you wanted to add some type testing to your code so that you could make sure that the values being passed in are of the correct type.

```
function greet(greeting, name) {
  if (typeof(name) != "string" || typeof(greeting) != "string") {
    return ("Greetings");
  }
  return (`${greeting}, ${name}`);
}
console.log(greet("Hello", "Heidi")); //"Hello, Heidi"
console.log(greet(5)); //"Hello"

function greetHello(name) {
  if (typeof(name) != "string") {
    return ("Hello");
  }
  return (`Hello, ${name}`);
}
console.log(greetHello("Heidi")); //"Hello, Heidi"
console.log(greetHello(5)); //"Hello"

function greetHi(name) {
  if (typeof(name) != "string") {
    return ("Hi");
  }
  return (`Hi, ${name}`);
}
console.log(greetHi("Heidi")); //"Hi, Heidi"
console.log(greetHi(5)); //"Hi"

function greetHowdy(name) {
  if (typeof(name) != "string") {
    return ("Howdy");
  }
  return (`Howdy, ${name}`);
}
console.log(greetHowdy("Heidi")); //"Howdy, Heidi"
console.log(greetHowdy(5)); //"Howdy"
```

What a lot of extra code this is! All those extra tests, all that redundancy across these different functions where we're trying to do the exact same type checking in multiple places. There must be a better way.

Currying is going to come to the rescue. We can have multiple independent functions. We can set them up so that they take simple arguments. We can make our code customizable, and make sure that our functions all share the same template code while still staying just as versatile as we need them.

And in the next section I'm going to show you how.