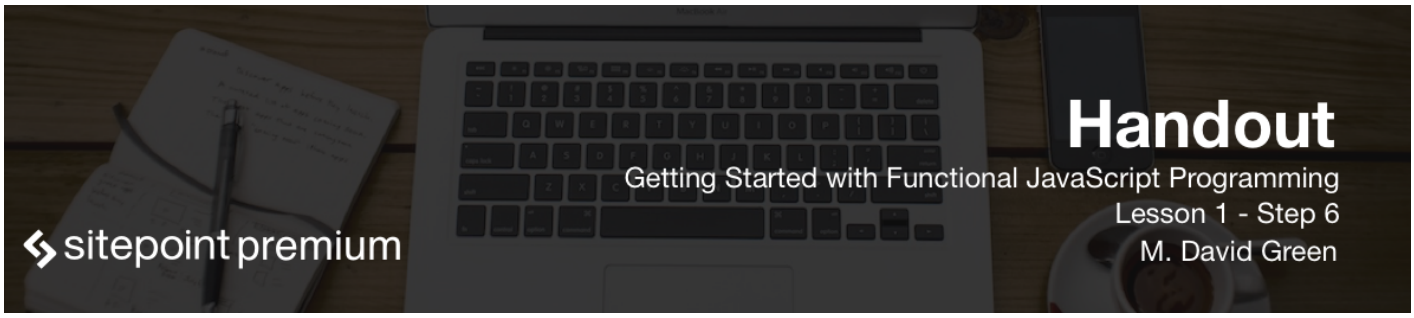

Another fundamental concept to understand when you're getting into functional programming is **Higher Order Functions**. So what is it that makes JavaScript functions different from functions in other languages? Some people say that JavaScript treats functions as first class citizens. What that really means is that the type of the function in JavaScript it's actually the same as the type of an object and as a result of that functions can be assigned to a variable just like a value.

And those functions and variables can be passed around as arguments to other functions. Similarly, functions can be returned by a function as a result, which gets us to the point of what is a higher order function. A higher order function is a function that takes another function as an argument or a function that returns a function as a result.

And of course, a higher order function could also accept and return a function. Now the idea of functions taking functions and returning functions may seem a little bit abstract, but you may be surprised to find out that if you've been doing JavaScript for a while, you've probably already used some higher order functions in your own work.

Now later in this course, I'm going to be introducing mapping and reducing and some of the other higher order functions that really demonstrate the power of higher order functions can provide. But first, I'd just like to demonstrate to you just how transparent higher order functions are in JavaScript.

For example, you may have used setInterval and setTimeout to control the timing of events:

```
const logTime = () => console.log(new Date().toLocaleTimeString());
const timer = setInterval(logTime, 1000);
setTimeout(() => clearInterval(timer), 3000);
```

We can see that `setInterval` and `setTimeout` are both Higher Order Functions in that they both take a function and then do something with it.

Now let's imagine you're doing some text processing on a document and it keeps on referring to the *millennials*, the *millennials*, *millennials* and you're getting tired of hearing about the *millennials*. And every time that you can counter the word millennials, you just want to replace it with the word *snake people*.

```
const snakify = text => text.replace(/millenials/ig, "Snake People");
console.log(snakify("The Millenials are always up to something."));
```

Now let's create another similar function:

```
const hippify = text => text.replace(/baby boomers/ig, "Aging Hippies");
console.log(hippify("The Baby Boomers just look the other way."));
```

We can see that our two functions are working properly. One of the things to notice though is that these functions have a lot of duplication and these are very short functions in this case. But you can imagine that you might have more

complicated functions. In that case, you might be duplicating a lot of code and violating the principle of don't repeat yourself when you're writing your code.

But when it comes to not repeating functionality across multiple functions, that's one of the places where returning functions as a result can really pay off. So if you wanted to reduce the duplication in your code, you might want to create a function with code that encapsulates the things that are repeated across those multiple functions and then have that function return a function.

They can be customized to do exactly what you want in different situations and I'll show you how that would look:

```
const attitude = (original, replacement) => {
  return function(source) {
    return source.replace(original, replacement);
  };
};

const snakify = attitude(/millenials/ig, "Snake People");
const hippify = attitude(/baby boomers/ig, "Aging Hippies");

console.log(snakify("The Millenials are always up to something."));
// The Snake People are always up to something.
console.log(hippify("The Baby Boomers just look the other way."));
// The Aging Hippies just look the other way.
```

In this case, the `snakify` and `hippify` functions, can be so simple because they're derived from the core functionality of `attitude` itself. And we've written this in a broken out way but it's possible to write this in an even more concise way, using more ECMAScript 2015 syntax.

In the process, we may have sacrificed a little bit of readability and I'm going to show you why I prefer to do it the other way.

```
const attitude = (original, replacement) => source => source.replace(original, replacement);
const snakify = attitude(/millenials/ig, "Snake People");
const hippify = attitude(/baby boomers/ig, "Aging Hippies");

console.log(snakify("The Millenials are always up to something."));
// The Snake People are always up to something.
console.log(hippify("The Baby Boomers just look the other way."));
// The Aging Hippies just look the other way.
```

I personally find it a little bit confusing when looking at something like this to try to understand what the higher order function of `attitude` is actually trying to accomplish? And I think it's because of those multiple arrows. I mean, that draws attention to the fact that there is a function returning a function. I find it harder to read and I think it makes the code a little bit more obscure. I personally find it easier to use the `function` keyword for a situation like this. You may choose to do it differently and it depends on what your team prefers. But keep in mind that the important thing about using these new syntax options is the ability to maintain your code and have it be understood by the next developer who's going to touch it.

What is the value of higher order functions? Now in this course one of the things that we're going to be focusing on is pure functions. So the value of pure higher order functions here is that they would provide you with callbacks that you can always rely on. Better encapsulation of the methods.

Pure higher order functions make it easier to combine functions, for more complex functionality without a lot of repetition. And that can be versatile for creating sets of related functions as well. And there's more, which we're about to learn in the next chapter.