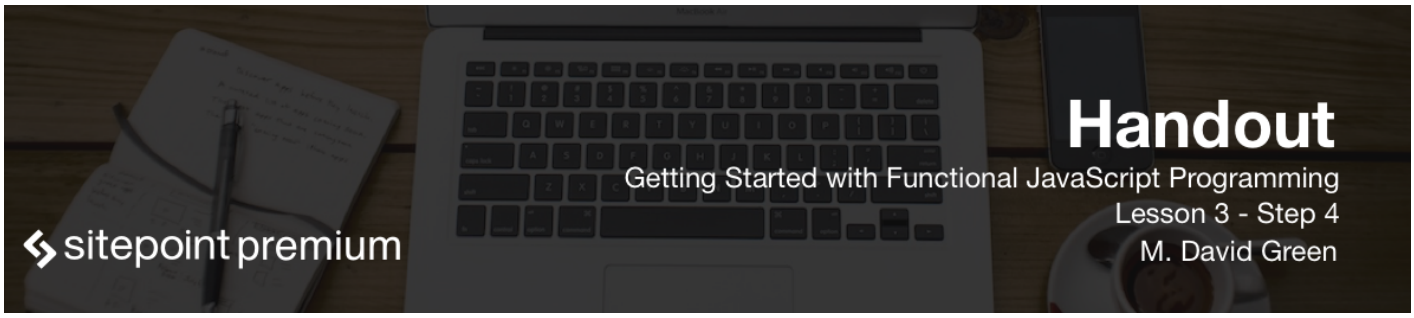

So we've been talking about currying in JavaScript, but currying as a concept comes from functional programming languages where there's more control over the signatures of the functions. It's a little bit more tricky to apply it in a JavaScript context. And because of that it's worth bringing up the concept of partial application, especially when dealing with variadic functions.

When we were putting together our currying examples in JavaScript, you noticed that we were calling our functions with multiple arguments by passing each argument as a separate set of parentheses. So each function only took one argument and then called the next function which only took one argument, which called the next function which only took one argument.

The nesting worked, but the syntax looked a little bit awkward in JavaScript. It was a lot of parentheses. When we're dealing with JavaScript we have the option of creating functions with multiple parameters by definition currying always returns a function that only has an arity of one, that is, a function that only takes one parameter with currying, you apply the arguments in a nested matter.

One argument at a time until all of the parameters have been supplied at which point the function returns. JavaScript syntax allows you to create functions that can take any number of arguments. And in some ways, this puts it into a different category from other functional programming languages. Which rely on specific and defined functions signatures.

It's not at all unusual to see a JavaScript function that might take four arguments. Some of which may be optional, and some of which may be mandatory. As a result currying is complicated because you never know in advance how many arguments its function may take. That's where the concept of **partial application** comes up.

With partial application you take a function, with multiple arguments, and you're returning a partially applied version of that function, which is waiting for additional arguments. And we can imagine how we might be able to create a partial utility that would be able to take an existing JavaScript function and return a version of that function that's waiting for our missing arguments without all of the nesting and passing each argument as an individual item.

For example, if we had our `greeter` function and it took a greeting, and a separator, and an emphasis, and a name what we would like to have is a partial utility function that could take that `greeter` function and pass perhaps the first three parameters and return a function that's waiting for the final parameter.

```
function greeter(greeting, separator, emphasis, name) {
  return (greeting + separator + name + emphasis);
}
const greetHello = partial(greeter, "Hello", ", ", ".");
console.log(greetHello("Heidi")); //"Hello, Heidi."
console.log(greetHello("Eddie")); //"Hello, Eddie."
```

If we did this correctly, our partial utility would not need to know the arity of the resulting partially applied function in advance. So let's think about what it would take to create a rudimentary partial application function. What it would

need to do is take a function with multiple arguments and it may have any number of arguments and return a partially applied version of that same function.

```
const partial = (variadic, ...args) => {
  return (...subargs) => variadic.apply(this, args.concat(subargs));
};
```

We're defining a new `partial` utility that's gonna take a variadic function of any kind typical JavaScript function, and all of the arguments that you wanna pass in, and it's gonna return a function that's going to take the rest of the arguments, the missing arguments, and apply those concatenated with the original arguments.

And with a little utility like this, we can create child functions that can accept a subset of the original arguments for the function, always starting from the end which is why the argument order does matter when you're using a technique like this. Now, if we just create a normal JavaScript function that takes multiple arguments, so we can make `greeter`.

```
const partial = (variadic, ...args) => {
  return (...subargs) => variadic.apply(this, args.concat(subargs));
};

const greeter = (greeting, separator, emphasis, name) => {
  return (greeting + separator + name + emphasis);
};

const greetHello = partial(greeter, "Hello", ", ", ".");
console.log(greetHello("Heidi"));
```

Note that the arguments are in a different order than the order in which the parameters are passed in. Because we're passing in the name the last since that's what we might wanna modify first.

When defining new functions, as long as we add the missing arguments in the order that they're expected. So we know already that if we were to copy or console.log(greetHello) and pass in a different name such as Eddie. And go over here and run that again, we would see Hello Eddie.

```
const greetGoodbye = partial(greeter, "Goodbye", ", ");
console.log(greetGoodbye(".", "Joe"));
```

So with our partial utility function, we can do partial application using variadic functions in JavaScript and get around some of the syntactic confusion that can come when you try to apply currying in a language like JavaScript.