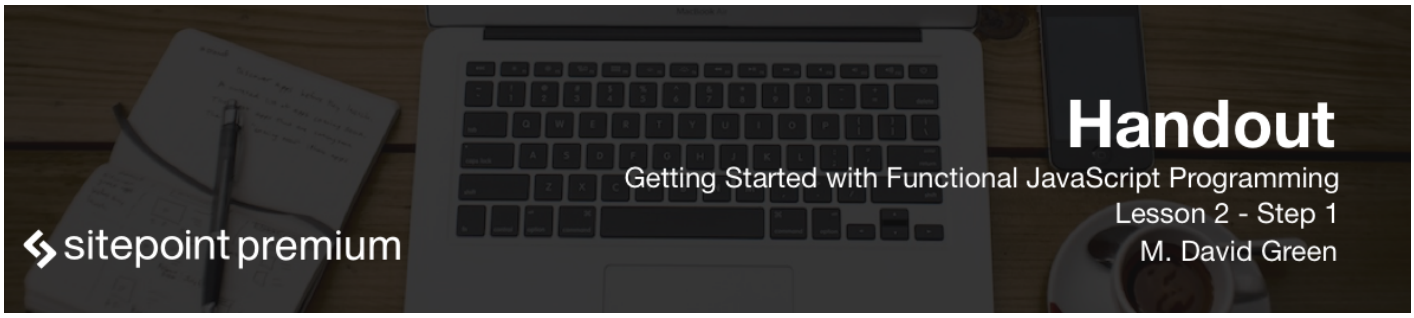

Welcome back to Functional Programming with JavaScript, Lesson 2. We are going to talk about **recursion**.

One of the functional programming techniques you might have heard of before is recursion, in which a function calls itself while it's executing. Of course a function calling itself is calling a function, which makes it a higher order function.

And if you follow the good practices of making pure functions and structure your recursion correctly, recursion can be a good solution for certain types of problems. Now to start with, I'm gonna show you how traditional looping works for a problem that might otherwise be better solved with recursion.

So what is recursion? Well, first of all, the definition of recursion means repeatedly calling a function from within itself, and then iterating until a completion state is reached. Ideally, you return to result without affecting anything outside of the function, and recursion can be a good alternative to a traditional looping structure.

So an example of a type of problem that might be best solved with recursion is the classic factorial. To perform a classic factorial operation, you multiply an integer by the previous integer. Then you multiply that product by the next previous integer and you keep on going backwards until you reach 1.

At that point you stop, and you return the result. So for example, to calculate the factorial of three, you take 3 multiplied by 3 minus 1, which is 2. And then multiplied by 2 minus 1, which is 1 and you'd get 6. Similarly, the factorial of six is 6 x 5 x 4 x 3 x 2 x 1, which gives us a result of 720.

That doesn't sound like a complicated problem to solve. I can show you how we would do it traditionally, using a `for` loop in JavaScript:

```
const factor = number => {
  let result = 1;
  for (let count = number; count > 1; count--) {
    result *= count;
  }
  return result;
};
console.log(factor(6)); // 720
```

We've actually created some problems for ourselves along the way that make the code a little bit messy. For example, we're creating a lot of local state. We've got that result that keeps changing values before we return it. State is being created and destroyed along the way. We keep on changing the value of `count` as we go, and then when we're done, we don't use it for anything. While the syntax is familiar, it's a little bit harder to read just because of the way that for loops manage their conditions.

We could also do the same factorial using a `while` loop:

```
const factor = number => {
```

```
  let result = 1;
  let count = number;
  while(count > 1) {
    result *= count;
    count--
  }
  return result;
};
console.log(factor(6)); // 720
```

Our code looks pretty similar but there are some differences. We have the same number of local variables that we're defining but this is slightly cleaner. The `count` is defined outside of our loop which makes it easier to reason about. And our terminal state, the point at which our loop is going to exit, is very clearly emphasized.

And that's going to be useful for us when we're starting to do actual recursion. So let's get to some basic recursion.