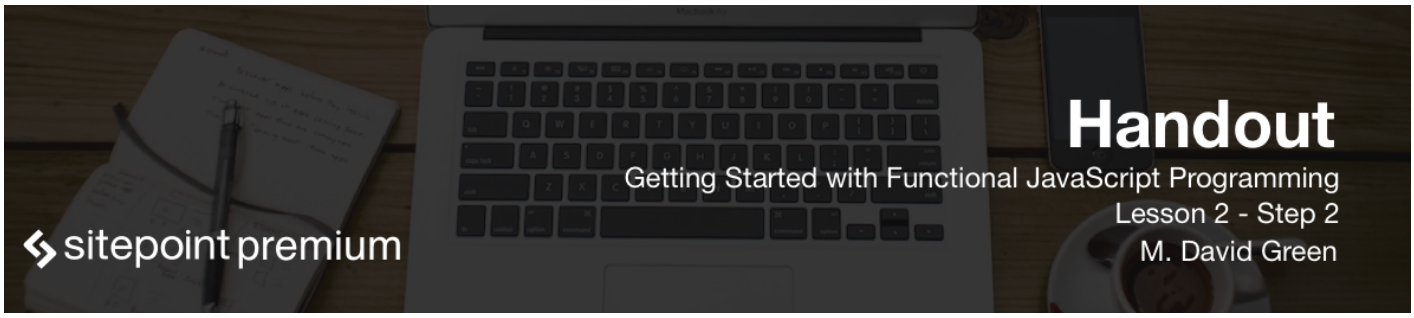

At the end of the last section, we left off with a `while` loop. And the reason I brought you two while loops is because they are good way to conceptualize how recursion works. So, let's think about a `while` loop and let's think about ways we can iterate without looping.

One way to think about this is to try to reproduce the effect of a `while` loop but without using a `while` loop. Here is the simplest `while` loop that you can imagine:

```
let counter = 10;
while(counter > 0) {
  console.log(counter--);
}
```

Iif you're like me, you're starting to think about the state that's being maintained in that counter and whether we could avoid having something that maintains state in our program. Let's see if we can reproduce the functionality of a `while` loop without creating all that state. One of the ways to do this is by calling a function from within its own definition:

```
const countdown = value => {
  if (value > 0) {
    console.log(value);
    return countdown(value - 1);
  } else {
    return value;
  }
};
countdown(10);
```

In order to understand this, it's good to know what the elements are of a recursive function. The most important element when defining a recursive function is a **terminal condition**. Without a terminal condition that functional just keep executing forever.

So let's see if we can do our factorial using recursion.

```
const factorial = number => {
  if (number <= 0) {
    return 1;
  }
  return (number * factorial(number - 1));
}

console.log(factorial(6));
```

Part of the value of using recursive functions is the clean code that you get. We're performing a stateless operation. We're not changing the value of any variables along the way. We're having no side effects outside of the functions, so nothing outside is being changed. We're testing for the terminal condition first. And that allows us to exit quickly and cleanly from our recursive function.