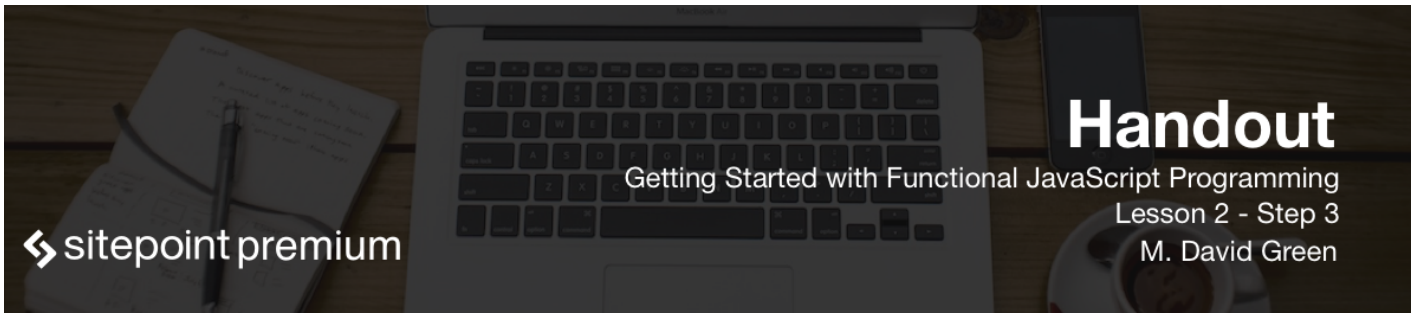

It would be great if I could tell you that recursion works flawlessly and that you don't have to worry about anything when you start using it. But unfortunately recursion does have a price and there are some things you need to consider. First of all the syntax may be a little bit confusing for people who are unfamiliar with the concept.

After all, remember what it sounded like to you when I first suggested that a function can call itself from within the definition of that function. Recursion also incurs a high memory price and depending on how deep the recursion goes that can cause an issue for your programs. There's a technique called proper **tail calls** and they are an option, but in order to use them effectively you have to make your recursive functions a little bit more complex. In addition, JavaScript engines can optimize for proper tail calls in order to avoid the deep memory issues. But tail call optimizations to take advantage of proper tail calls safely have not been broadly supported yet, and that's been slow to roll out. So in terms of these issues, the first thing I mentioned was that the code can be confusing to read.

In order to understand it better, it's a good idea to break down the code you're looking at into the return values that you're dealing with. The high memory use though, is much more difficult to deal with, because each successive recursion is stored in the stack.

So in our case, when we did `console.log(factorial(6))` we were running this code six times and that meant that each one of these was being stacked up in memory and maintained while the first one was running. So the values could gathered from them and then passed back to the original function before returned.

That's a lot of memory usage. In some cases it's possible to get around this memory issue with recursive functions by using a technique called proper tail calls. With proper tail calls, your recursive function can rewrite values to the last frame of the memory stack. And you do that by making sure that the recursive call itself is in a tail position in the function. That's done by making sure that that recursive call is returned without any additional calculations. If we take a look at the factorial function that we've been using, we can see that the final return of the function does return our factorial recursive, but it does so in the context of a calculation.

It's not being returned by itself. So our factorial does not implement a proper tail call and therefore it can't be optimized to take advantage of the last frame of the stack. So in order to make factorial optimizable, we might want to create a function and call it const `factorialPTC` for proper tail call.

```
const factorialPTC = number => factorIt(number, 1);

const factorIt = (number, accumulator) => {
  if (number <= 1) {
    return accumulator;
  }
  return factorIt(number -1, number * accumulator);
};

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

So what we've done is refractor our factorial and we've put our call to factorIt in the tail position. So that our recursive function implements a proper tail call and can be optimized if the JavaScript engine supports tail call optimization.

The tail call optimization that functional programming languages often have is supposed to be part of that ES2015. But as I said before, you need to check and make sure that the place where you're rolling out your code, and where users are gonna be touching it, is actually going to be able to support proper tail calls.

And as you can see here from late 2016, desktop browsers have been kind of getting there, but they're not really quite there yet. The point to keep in mind is that you need to consider the context that you're working in, the people who are going to be using your code.

And how you're going to be deploying, before you decide whether or not you want to use recursion.