Callbacks

Introduction

Callbacks can be a bit of a challenge, but like every topic we touch on, the best thing to do is get used to using them, and then worrying about how they work under the hood a bit later. You will see callbacks everywhere in JS, and you'll soon be using them without fear.

Let's start things off with an example:

`.forEach` loop

The `.forEach` function is built-in loop that operates on arrays in JS. Here is an example:

```
// create an array
let colours = ["red", "blue", "green", "yellow"]
// Here we are using a built in array method, `.forEach()`
// `.forEach` takes a callback function as an argument
colours.forEach(function(colour) {
    console.log(colour)
})
```

Here you can see we are setting up an array. Arrays have a built in JS method called `.forEach` that functions as a loop. The `.forEach` method takes a callback function as an argument. You can see here that the callback function is an anonymous function that appears to be a function definition, but is situated in the place where an argument should go. Seems odd, right?

It is odd, and it takes a little bit of getting used to. One thing to remember while we go through this, is that functions in JS are just values like everything else, and also that functions are just groupings of code that perform a particular job (which can be a simple job, or can be a complex job).

If you run the code, you will see that we are logging out each of the values in the array in turn. So that means that the argument that we here called `colour` is standing in for each element in the array as we pass through it.

The first issue that you might notice is that we aren't defining `.forEach`, and yet here it is. And also we get to decide the *name* for `colour`, but not *what* it is. For example, the following will work in the same way:

```
colours.forEach(function(yippio) {
    console.log(yippio)
})
```

There's also one more notable thing (at least): our function looks like a definition, but seems to be being run.

So many mysteries.

Let's have a look at another example and see if we can get rid of some of the mysteries:

```
// Defining a function that takes a function as an argument
const funcWithCallbackArg = function(callback) {
    console.log("This OUTER FUNCTION is RUNNING")
    callback("The callback is being sent this string")
}
// Now we are calling that function, and running it with a callback
function as the argument
// The callback is often like a way to modify the behaviour of the
function that calls it
funcWithCallbackArg(function(x) {
    console.log("In the callback..")
    console.log(x)
})
```

Here we are defining a function that takes a callback function as it's argument. We can see inside that it runs a `console.log`, and then *runs* the callback function. In this case we are passing a string to the callback function. We could pass nothing, or we could pass several arguments.

We can see the function being run next, and the callback function being defined in the argument space. It takes an argument. In it we run a `console.log` that will appear when the callback runs, and then we log out the argument.

The way it works is that the calling function is run, and it runs the code as you'd expect. You can see the callback being run after the `console.log`, and this is equivalent to running all the code in our callback.

I understand this is a lot of mentioning of callback and call and fucntion, so let's see another example to see if it makes things clearer:

```
// Defining a function that takes a string and a function as an argument
const sportCallbackArg = function(sport, callback) {
    console.log(`Looking forward to playing ${sport}`)
    callback(`${sport}! ${sport}! ${sport}!`)
}

// Here we are calling that function, and chosing 'rugby' as the argument.
sportCallbackArg('rugby', function(givenArg) {
    console.log("In the callback..")
    console.log(givenArg)
})
```

Still a bit tricky?

The first thing to know is that you can get very used to using them without understanding them well. For example, with this `.forEach` loop, we know that we are required to write a callback function as the argument (we will get an error if we don't). We also know (and can find in the docs) that the callback

function will be passed an argument (at least one), and that this argument will be each element in the array in turn.

Let's have a go at making our own version of this `.forEach` type loop, but as a stand-alone function (rather than a method on an object).

```
// We are making a function that takes an array and a callback, and
performs very similarly to the normal `.forEach` loop
const myArrLoop = function(arr, callback) {
    // Setting up the counter
    counter = 0
    // A while loop to loop through the array
    while (counter < arr.length) {
        // Here we are passing each element to the callback in turn
        // Because of the way we set up this loop, the callback will be
run as many times as there are elements in the array
        callback(arr[counter])
        // Increment
        counter += 1
    }
}
// Setting up an array
const arr = [5,6,7]
// Calling the function with the array and the callback as arguments
myArrLoop(arr, function(arrItem){
    // Just a console.log to show it working
    console.log(arrItem)
})
```

Have a look at the above code, and see if you can make sense of what is happening. If you have any trouble, please get in touch and let me know how I can help to explain things better.

Advanced: another example - prototypes and 'this'

This example is very much to show under the hood of JS, and is definitely advanced. If you don't understand it it's no worries, and won't get in the way of the things we are learning in this course. But some might like to get a sense of things under the hood, and it ties together several concepts from JS that we have discussed so far.

One of the first things to note is that Js has a prototype inheritance system. It's too complex to go into detail here, and you are welcome to have a look at the details (here is a good start: MDN - Inheritance and the Prototype Chain).

But one thing you can understand is that everything in JS inherits from an object. In this case we will be looking at arrays, and while an array appears to be a stand-alone structure, it infact inherits from the Array.prototype object (and is itself an object).

When we create an array, we create an 'instance' of this Array.prototype object, and that has the specific features of the array we are creating, along with all the built-in array methods and attributes. This is how, without doing any work, we have access to things like `.length` and `.forEach()`.

Let's get in to the code:

```
// Here we are making a new key on the Array.prototype object,
'myForEach'.
// The value of that key is a function, and the function takes a callback
(which here we will call 'cb')
Array.prototype.myForEach = function(cb) {
    // We are setting the counter to zero
    counter = 0
    // We start our loop
    // The constraint on the loop uses 'this'. Here, 'this' will refer to
whatever particular array is calling the .myForEach function.
    while (counter < this.length) {
        // We call the callback, and send as the argument each item in the
array in turn
        cb(this[counter])
        // Incrememnt the counter
        counter += 1
    }
}
// Let's make a new array
const arr2 = [9, 7, 5]
// Here we call the function we added to the prototype. We are calling
that function on a particular 'instance' of Array, `[9, 7, 5]`. Because we
are operating from that instance, when we access 'this' inside this
function, it will refer to this array instance.
// We have a callback as our argument
arr2.myForEach(function(element) {
    // A console.log to show the behaviour
    console.log(element)
})
```

Just to be clear, there is a lot going on here, and it's not something you will do often. But it does highlight several features we have talked about so far in JS (functions on objects, this, callbacks), and gives a peek under the hood of JS itself.

Summary

Here we discussed:

- Callbacks
- Passing arguments to the callback function
- Prototypes and 'this'