Functions: In Objects, and in functions

Introduction

This document just expands the range of what you can do with functions. In JS, as you have seen, functions can be assigned to variables. This opens up a whole world of things that can be done with functions that we are going to look at here. Some of this can look a little complex, but take your time, ask questions, and have a go at the challenges associated with these.

Functions in objects

Firstly, objects can have functions as the value of their keys. Let's have a look:

```
// Defining one object with three keys, one that has a string value, and
the other two with functions as values.
const directionsObj = {
    info: "This directions object has two functions as values",
    left: function() {
        console.log("You have moved LEFT")
    },
    right: function() {
        console.log("You have moved RIGHT")
    }
}

// Logging out the value in the info key
console.log(directionsObj.info)

// Here we are assigning but NOT running the funciton at the key 'left'
const leftFn = directionsObj.left

console.log("Call LEFT")
// Here is the actual call of the 'left' function.
leftFn()

// Here we are directly calling the function in the 'right' key
directionsObj.right()
```

What you can see from these examples (and from your experience with functions prior to this), is that you can pass functions around much like any other object.

Especially note that we are accessing the functions through the key, and that we can both assign that function to a constant and then later call that function using its new name (`leftFn()`), or we can call it directly (`directionsObj.right()`) through the function.

The flexibility of the way functions can be passed around leads to other benefits. For example, we can pass functions in and out of functions as arguments and return values.

```
// Simple function that just logs something out.
const firstFunc = function() {
    console.log("Hi, firstFunc being run now.")
}

// This function takes an argument that it expects to be a function. It
logs something out, and then calls the function argumemt.
const secondFunc = function(fn) {
    console.log("secondFunc is running")
    fn()
}

// Here we are calling this second function, and sending as the argument
the first function.
secondFunc(firstFunc)
```

Here we can see that we have a very normal firstFunc function that just `console.log`s something out. The second function takes an argument, which must be a function (otherwise you will get an error when you try `fn()`). The function just `console.log`s something, and then runs the function.

Nothing is called though, until we hit the line `secondFunc(firstFunc)`. Here we are calling the second function, and calling it with firstFunction as its argument. `secondFunc` then runs, and in the process it calls `firstFunc` (although at the time it does this it is `fn()`).

We could add a little to this, and show how this might be used for multiple functions. This is getting quite advanced, so just do your best:

```
// Simple function that just logs something out.
const firstFunc = function() {
    console.log("Hi, firstFunc being run now.")
}

// This function takes an argument that it expects to be a function. It
logs something out, and then calls the function argumemt.
const secondFunc = function(fn1, fn2) {
    console.log("secondFunc is running")
    fn()
}

// Here we are calling this second function, and sending as the argument
the first function.
secondFunc(firstFunc)
```

Scope??

JS Prototype System

`this`