Further JavaScript Object Topics: Unobtrusive JavaScript, "this", ECMAScript 6 Features

NOTE: This is a relatively advanced topic which is aimed at those of you finding the unit, and programming in general, relatively straightforward. You should only attempt this if you have completed **all** previous exercises and are confident with your answers. If you are struggling with the unit it is recommended that you catch up with previous topics first. You can then return to this topic later, when you are confident with all previous topics.

User input

Before we begin the main topics for this session, we just need to briefly talk about a "loose end" in AJAX development. When building an AJAX application it can be useful to gather ***user input*** in some cases. For example, imagine that HitTastic! had a fully AJAX-based review facility allowing a user to review songs after searching for them. How could this be done? There are a number of ways, some involving using external libraries, but here is one.

You can use a ***dialog box***. This is just a <div> which can be shown or hidden. You could create this using the DOM. When the page loads you could create a <div> with a label, a text area and a button, and give it an ID. You could show this dialog when the user chooses to review a song, and then hide it when the review has been submitted. For example, assuming the dialog is within a <div> with an ID of "dialog", this can be shown and hidden using DOM code such as:

```
document.getElementById('dialog').style.display='block
```

to show it as a block-level element, or

```
document.getElementById('dialog').style.display='none'
```

to hide it.

The dialog's button could be linked to a further JavaScript function to hide the dialog div when clicked

Unobtrusive JavaScript

Unobtrusive JavaScript is JavaScript in which event handlers are setup in code rather than the HTML. For example, rather than:

```
<input type="button" id="searchBtn" onclick="search()"/
```

you would setup the event handler in your init() function with ***addEventListener()***, such as:

```
function init()
{
    document.getElementById("searchBtn").addEventListen
}
```

"this" and callbacks

In the previous examples on JavaScript objects, you will have seen the use of the keyword ***this*** in the examples. What does ***this*** actually mean? It represents ***the current object***. So, if we revisit the Cat object:

```
function Cat(n,a)
{
    this.name=n;
    this.age=a;
}

Cat.prototype.displayDetails = function()
{
    alert(this.name + " " + this.age);
}
```

When we write the following code:

```
var tom=new Cat("Tom", 7);
```

we create a new Cat object referenced by the variable ***tom***. The constructor function is called and the ***name*** and ***age*** of the newly-created object (referenced by ***this***) will be assigned to the values of the parameters ***n*** and ***a***. Likewise, in the ***displayDetails()*** method which belongs to the prototype for cats, ***this*** will refer to ***whichever object the method is being called in the context of***. So if we do:

```
tom.displayDetails();
```

then ***this*** will refer to ***tom***, but if we do:

```
tigger.displayDetails();
```

then ***this*** will refer to ***tigger***.

Subtleties with "this"

There are a number of subtleties with JavaScript ***this*** which you need to understand to work effectively with objects, and which make its use less straightforward than the use of ***this*** in other languages such as Java and C++.

"this" and the window object

Firstly, what does ***this*** refer to if it is not being used in the context of an object? For example, if the function below was run, what would be displayed?

```
function thisTest()
{
    alert(this);
}
```

On Firefox it displays:

```
[object Window]
```

The output demonstrates what ***this*** refers to when out of the context of an object. It refers to a ***"default object"*** which in a web browser is the ***window*** object, representing the browser window as a whole. So the example below would work too:

```
function thisTest()
{
    window.alert("alert is actually a method of the wi
    this.alert("... and here, *this* refers to the wind
}
```

Both alerts are displayed. ***alert()*** is actually a method of the window object, but because the window object is the environment's "default" object, we can normally leave ***window*** out; it's included here just to illustrate the point. And because ***this*** also refers to the window object in this context, the second alert will be displayed too.

"this" and callbacks

The problem most commonly encountered with "this" is when it is being used in a ***callback function***. A ***callback function*** is a function that does not run immediately, but at a later time, when some event has occurred (such as the user clicking a button, or the response from an AJAX request is received). So imagine we had a Cat and we wanted it to meow if we pressed a button, e.g:

```
<html>
<head>
<title>"this" callback test</title>
<script type='text/javascript'>

function Cat(n,a)
{
    this.theName=n;
    this.age=a;
}

Cat.prototype.meow = function()// e is the event object
{
    alert(this.theName + " says Meow!");
}


// Runs when the page loads
function init()
{
    // Create the cat
    var tom = new Cat("Tom", 7);

    // When the user clicks the button, the meow metho
    document.getElementById("meowbtn").addEventListene
}
</script>
</head>
<body onload='init()'>
<input type='button' id='meowbtn' value='Meow!' />
</body>
</html>
```

If you try this out, and click the button, does the expected output:

```
Tom says Meow!
```

occur? No. Instead, you will get something like:

```
undefined says Meow!
```

Why is this happening?

- On a simple level, it is because ***callback functions do not keep track of the object that the method relates to.*** So when we click the button, the meow function will be called but the fact that it relates to the object ***tom*** is not retained. So when we try to display ***theName*** of the current object, we get ***undefined*** because the current object is not ***tom*** and hence has no ***theName*** property
- To explain in more depth: ***when we are using a callback function, the context of the function changes***. In this example, we are basically setting the ***click*** event handler of the button to be the meow method of the object ***tom***. But the object ***tom*** itself is ***not*** copied across. Instead, the code in ***meow()*** is copied across to the button object, and becomes the ***click*** method of the button object. So, when we click the button, ***this*** is actually the button object! Buttons do not have a ***theName*** property, so we get ***undefined***.
- A question: why did I use ***theName*** and not ***name*** in the above example?

Dealing with "this" issues in callbacks with bind()

Up to relatively recently, it was a little awkward to resolve the issue of keeping track of the correct object in callback functions. However, now it is actually very easy, as long as you are using up-to-date browsers. You can simply use the ***bind()*** method. ***bind()*** does what it says on the tin: it binds the callback function to the specified object, so that when the callback is called, ***this*** will refer to the correct object. So to rewrite our Cat example so they work correctly:

```
<html>
<head>
<title>"this" callback test</title>
<script type='text/javascript'>


function Cat(n,a)
{
    this.theName=n;
    this.age=a;
}

Cat.prototype.meow = function()
{
    alert(this.theName + " says Meow!");
}


// Runs when the page loads
function init()
{
    // Create the cat
    var tom = new Cat("Tom", 7);

    // When the user clicks the button, the meow metho
    document.getElementById("meowbtn").addEventListene
            (this, tom.meow.bind(tom));
}

</script>
</head>
<body onload='init()'>
<input type='button' id='meowbtn' value='Meow!' />
</body>
</html>
```

A more complex example

This more complex example shows a common case of the need to preserve "this". A fairly common approach to development is to have a so-called "application object" to represent the application as a whole. This prevents the need for global variables. For instance, in a mapping application, we can have the map object as a property of our app object. In the example below, the constructor is also linking a 'go to location' button to an event handler. The idea is that the user enters a latitude and longitude, and the map moves to that location when the user clicks this button.

```
function MappingApp()
{
    this.map = L.map("map1");
    this.tileLayer = L.tileLayer(...etc...);
    // rest of map setup omitted...

    // Link button to the move() method of the Mapping
    document.getElementById("gotoloc").addEventListene
}
```

We could create the application object in our init() method, to be run when the page loads:

```
// In our init() function, create the app object
function init()
{
    var app = new MappingApp();
}
```

The ***move()*** method, which runs when the user clicks the button, then moves to the location entered by the user. Note how ***move()*** attempts to access the application object using ***this***.

```
MappingApp.prototype.move = function()
{
    var lat = document.getElementById("lat").value;
    var lon = document.getElementById("lon").value;
    this.map.setView([lat,lon],14);
}
```

As we have seen, this will ***not*** work - however if we use ***bind*** when setting up the event listener, it will - as the context of ***this*** will be preserved. Here's how we would use bind():

```
document.getElementById("gotoloc").addEventListener("c
```

Arrow functions and "this"

Arrow functions are another mechanism of dealing with this problem; a key thing about arrow functions is that they preserve the context of "this". In an arrow function, "this" refers to the ***same object as it does outside the arrow function***. In the example below, the click event handler is implemented as an arrow function; just like the above example using bind(), the context of ***this*** will be preserved.

```
function MappingApp()
{
    this.map = L.map("map1");
    this.tileLayer = L.tileLayer(...etc...);
    // rest of map setup omitted...

    // Link button to the move() method of the Mapping
    document.getElementById("gotoloc").addEventListene
            () =>
                {
                    var lat = document.getElementById('
                    var lon = document.getElementById('
                    this.map.setView([lat,lon],14);
                } );
}
```

Arrow functions without the parameter brackets

When writing arrow functions, the brackets round the parameter ease the transition to arrow functions from regular functions. However, they are actually optional, for example we can write an AJAX callback as:

```
xhr2.addEventListener ("load", e => {

                var html = "";
                var data = JSON.parse(e.target.response
                for(var i=0; i<data.length; i++)
                {
                    html = html + "Song: " + data[i].so
                        data[i].artist + "<br />";
                }
                document.getElementById("response").inn
            });
```

Furthermore, with simple functions that perform a mathematical transformation, they syntax can be simplified still further so ***no return statement is required***. For example:

```
var getSquare = function(val) { return val*val};
```

can be simplified using an arrow function to:

```
var getSquare = val => { return val*val}
```

but in this case can be ***further*** simplified to:

```
var getSquare = val => val*val;
```

We create a variable ***getSquare*** representing our square function. Note the transformation from input to output using the arrow operator =>, so that, in this example, the function takes in ***val*** and returns ***val*val***. We would use it like a normal function, e.g:

```
alert(getSquare(1)); // shows 1
alert(getSquare(3)); // shows 9
```

Exercise

Convert the following previous exercises to use an application object and ***either*** bind() ***or*** arrow functions (your choice) to maintain the context of "this". ***Ensure you've done the corresponding exercises first!!!***

- Topic 12, Exercise 1 (geolocation)
- Topic 12, Exercise 2 (artist hometowns)

Other ECMAScript 6 features

Here are a few other interesting ECMAScript 6 features.

Backticks for output

Backticks avoid the need to manually construct strings. Variables are embedded in the backticks by using a dollar sign ***$*** followed by the variable wrapped in braces ***{ }***. Here is an example of a typical AJAX response with backticks:

```
function ajaxresponse(xmlHTTP)
{
    var data = JSON.parse(xmlHTTP.responseText);
    var html  = "";
    for(var i=0; i<data.length; i++)
    {
        html = html + `Song: ${data[i].song} Year ${dat
    }
    document.getElementById("response").innerHTML = dat
}
```

Default parameters

Functions can now take parameters which have ***default values***, e.g.:

```
function showGreeting(greeting='Hello')
{
    alert(greeting);
}

function test()
{
    showGreeting(); // shows 'Hello'
    showGreeting('Goodbye') // shows 'Goodbye'
}
```

The rest operator

The rest operator (...) allows us to combine multiple arguments sent to a function into an array. For example:

```
function restop(name, ...languages)
{
    console.log(`Hello ${name}, you know these language
    for(var i=0; i<languages.length; i++)
    {
        console.log(`${languages[i]} `);
    }
}

function test()
{
    // languages will contain just "C++"
    restop("Fred", "C++");

    // languages will contain John's four languages as
    restop("John", "C++", "Java", "JavaScript", "PHP")
}
```

The spread operator

The spread operator is the opposite of the rest operator in that it allows you to pass an array in as an argument to a function and the members of the array then get split into multiple parameters. It also uses the ... operator. For example:

```
function spreadop(firstname, lastname)
{
    console.log(`Hello ${firstname} ${lastname}!`);
}

function test()
{
    var name = ['James', 'Jones'];
    spreadop(...name);
}
```

Classes

You can actually use classes (in the same sense as Java) in ECMAScript 6. Some libraries and frameworks, such as React, use classes extensively. Here is an example of an ECMAScript 6 class:

```
class Cat
{
    constructor(n,a,e)
    {
        this.name = n;
        this.age = a;
        this.weight = w;
    }

    eat()
    {
        this.weight++;
    }

    walk()
    {
        this.weight--;
    }
}
```

Note the special ***constructor*** method. This is a constructor, but using different syntax to the prototype approach. To create the cat we would use very similar code to before:

```
var cat = new Cat("Tigger", 7,7);
cat.eat();
```

Note that ***classes use the standard prototype-based mechanism under the hood***; they are "syntactical sugar" (ref: Mozilla) rather than an entirely separate language construct.