Tutorial 3: Dealing with Lists of Objects

In labs 2 and 3, the timer application you have been building involves a collection – a list of appointments. In computing terms, a Collection is a structure that holds multiple things. A List is a specialised type of Collection in which all of the items are held in a specific order; this could be alphabetical, in order of some notional value or simply the order in which the items were added to the list.

In this tutorial, we'll look at ways of creating and manipulating lists of objects, leading up to a bigger design exercise. Initial exercises are to give you experience of using collections – later ones are to contribute to a bigger task.

Collections

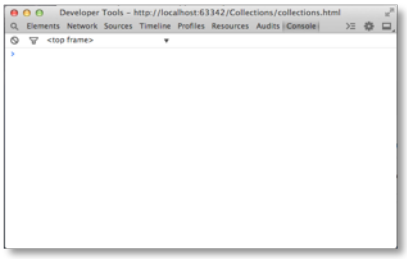
Javascript has two native types of collection:

> `Arrays: e.g. a = [];` or `a = new Array();` creates an empty array, `a.push(x);` adds x to the array etc. See http://www.w3schools.com/jsref/jsref_obj_array.asp for a complete reference
>
> Associative Arrays: e.g. `o = {};` or `o = new Object();` creates a new, empty object, to which items can be added using string values as keys (`o["one"] = 1; o["two"] = 2;` etc.). See http://www.w3schools.com/js/js_arrays.asp for a good description of the differences between arrays and associative arrays (a.k.a. Objects)

Exercise 1: Using simple collections

If you wish to complete the following exercises on a PC (recommended), start with a new HTML5 project, which has a simple html file (collections.html) and a Javascript file (collections.js) that the html file references. Open the html file in the Chrome browser and enter Ctrl+Shift+I to open up the developers' tools, and select Console from the set of developers tools along the top of the window. The developers' tool window should appear as shown, ready for you to enter commands:

Figure 1: A Developers' Tool window showing the Console view



Alternatively, write out the statements you think answer the following questions and test them the next time you have access to a PC:

a) Write two statements that create arrays (a1 and a2). Use the **`[]`** notation for the first, and the **`new Array()`** notation for the second. Verify using the console window that both are equivalent (use the `dir()` function – e.g. `dir(a1)` – dir stands for 'directory').

b) Add a single element (a string or a number) to `a1` using the `push()` method – use the `dir()` function again to see the result – note the result returned from the `push()` method

c) Add a new element to `a1`, but add it as element number 4 (`a1[4] = …`). Again, check the result using the `dir()` function

d) Write a `for()` loop to step through all of the elements in `a1`, printing their value on the console using `console.log()`. Note, you will need to enter the entire `for()` loop on a

single line, or start each new line using Ctrl+Enter

e) The result of part (d) should have shown you that you have a gap of three elements in the array (3 values equal to `undefined`). It would be useful to be able to remove these – the Array function `splice(index, howmany);` can be used to close gaps in an array – the value `index` indicates the first element you wish to remove, and the value `howmany` indicates the number you wish to remove. For example `arr.splice(10, 4);` would remove from the 11th (index counts from 0) to the 14th elements of `arr[]`. Write a statement to delete the three undefined elements, and use the `console.dir()` function to verify this

f) Use the .length property to display how many elements are in the array a1

g) Add three elements (strings or numbers) to array a2 (created at step a)

h) Use the Array.concat() function (see http://www.w3schools.com/jsref/jsref_concat_array.asp) to join a1 and a2 together into a single, new array, and then check how many elements are in the new array

The BIG Task: Make a collection of timers.

Since the last two tutorials were about managing timers, it seems sensible to continue with that, although you can build collections of any type of thing using exactly the same techniques. One issue about having multiple timers is that duration is no longer enough information - each timer ought to have at least a name (so we can recognise which one has just expired). We can do that easily:

```
var myTimer = { name:"Time for a coffee break", time: 3000};  // Assume 3000 is seconds.
```

This can now be implemented using the setTimeout() function:

```
setTimeout(function(){ alert(t.name);}, t.time);
```

Of course, if we want to deal with timers that have a regular format, the best approach is to use a Constructor function (see http://pivotallabs.com/javascript-constructors-prototypes-and-the-new-keyword/). Especially since the setTimeout() function call is so messy, involving an anonymous function.

Exercise 2: Constructing Timers

a) Write a suitable constructor function for creating a Timer object (recall the convention is that a Constructors function name starts with a capital letter) – enter this code in to the collections.js file. Carefully consider the data you would like each timer to hold

b) Add a prototype function, start, to the new Timer type, so that it displays the Timer's name in an alert box after the timeout period – it will contain a call to setTimeout()

c) Test your new Timer type (you will need to refresh the browser to make your new code 'live'). Try creating a timer with a short duration in the console, and then calling its start function

My guess is that in the test, you'll find that the message you want to display (the name of the timer) is not so easy to get into an alert box. We will need to discuss this (see box: anonymous function scope for info).

When a function is used as a parameter to another, the scope rules in Javascript cause a problem:

```
setTimeout(function() {
   alert(this.name);}, this.time
);
```

Inside the setTimeout() function, `this` refers to a Timer object. Inside the function that is its first parameter, `this` refers to the object that owns `setTimeout ()` which is a `Window` object). To fix this, we need to provide a new in-scope variable…

```
var message = this.name;
setTimeout(function() {
   alert(message);
}, this.time);
```

Exercise 3: Arrays of Timers

Finally, we get to the point of this tutorial. We'll try two different forms of collection:

a) Make sure your browser is refreshed up to date (with the fix to the Timer's start() prototype function mentioned in Anonymous Function Scope)

b) In a console window create a new "timers" array. The aim will be to add multiple timers to this array, so that each timer will pop up an identifying message at the end of its timeout

c) Add a couple of new Timer objects to the array – give them fairly short (but different) durations – less than 10 seconds

d) Start the timers (you'll need to use statements like (timers[0].start())). This will probably be quite awkward to do since one of your short-duration timers may well pop up while you are trying to start the other

e) Add the name of your timers array a to the collections.js file, and also add a new function – startTimers(), that will step through the array calling .start() on each element. Refresh the browser and then add some timers to the array and finally call this function

From here, we can start to improve of the Timer and collection code to make it easier to use in future. For example, you should consider:

1. Add an addTimer() function which will add a new timer to the array (it should have two parameters – name and duration)
2. Add a removeTimer(name) function which removes the timer which has a matching name

Exercise 4: Associative Arrays

Repeat all of the operations in Exercise 3, but this time using an associative array (map). The advantages of this will be that you can now refer to and manipulate timers by name, and iterate through the collection using the for..in syntax (e.g. for(t in timers){…}). ***Can you think of any disadvantages?***

A Timer type definition (slightly enhanced version of Ex 2)

```
var Timer = function (name, duration, actionFunc) {
    var me = this; // This provides a fixed reference to the new object
    this.name = name;
    this.duration = duration;           // Seconds
    this.actionFunc = actionFunc;       // Call this when the timer elapses
    this.timer = null;
};
Timer.prototype.action = function () {  // Having a defined action function
    if(this.actionFunc) {               // for this type of timer lets us
        this.actionFunc(this.name);     // say what happens when the timer
    } else {                            // times-out (e.g. could define
        alert(this.name);               // an action function to send an
    }                                   // email)
};
Timer.prototype.start = function () {
    var me = this;
    this.timer = setTimeout( function () {
        me.action();
    }, 1000 * this.duration);
}
Timer.prototype.cancel = function () {
    if (this.timer) {  // check it's not already null
        clearTimeout(this.timer);
        this.timer = null;
    }
};
```