Data Structures and Object Oriented Design

This chapter describes a more formalized approach to building web applications. So far, we've examined web apps where the HTML on the page was also the primary component of the page. For example, in the Shopping List example in the previous chapter, the actual list was an HTML element (a <ul> list), including all of its child elements. Whenever we altered the list (for example, adding an item, removing one or selecting an item), we did so by manipulating the HTML mark-up.

This turns out to be a very fragile way to work with large HTML documents or apps. It is based on the premise that ALL of an application will be rendered on the screen, and for some types of application, this will never happen. For example, consider an address-book app – it may have an appearance that is similar to a shopping list, in that all of the names in your address book would appear in a list. However, typically, the name element in an address book is no more than the *key* to the rest of the information pertaining to that person. If I tap on the name "John Smith" in my phone's address book, I'm shown a page that tells me John's address, telephone numbers, email address, possibly his photograph, maybe his birthday, the company he works for, his work telephone number etc. in a mobile app, it would not be feasible to display a single list of all of these details for every entry – too much information, and so the app needs to link the list of names in the address book to a separate page of data for each entry. If you have 1000 entries in your address book, that's a lot of pages.

In a web app, we get around this by building data structures – Javascript objects that contain more than one piece of data – sometimes a lot of related pieces (name, phone, email etc.), and sometimes a list of complex items (all of John Smith's info, all of Fred Bloggs's info etc.). Javascript provides a couple of very simple but powerful ways for organising data.

Data Structures and Object Models

In software development terms a data structure is a group of collection of individual data items that are considered as a single, complex item. We've already seen how we can define a data structure in Javascript:

```
var person = {
 name: "Fred Bloggs",
 email: "fred@bloggo.com",
 dob: "22/06/1990",
 mobile: "07702 123456"
};
```

Listing 6.1: An item of structured data

Listing 6.1 shows an example of data aggregation in Javascript. This is essentially a way of collecting individual items of data to create a single 'collection'. Javascript gives us a few ways to group individual objects into a single object to make a collection:

- Arrays: an array is normally organised as an indexed list of similar types of object. The index is an integer, in the range 0 to 1 less than the number of elements in the array
- Objects: an object, as we saw in the previous chapter, is simply a group of name + value pairs. We can use this to create a ***map*** or ***mapping***, which is a collection of objects (the values) that are associated with a collection of ***keys*** (typically strings)

- Custom Objects: we can create objects which have methods added specifically to manage collections of other objects. This gives us the opportunity to improve how Javascript deals with collections in some way – added security, easier access, aggregated objects sorted in some way or other specialized object-management features

Arrays and Object mappings are built in to the Javascript language, and so these would be the default approaches we would use to forming collections. However, the option of creating a customized collection is often a powerful one that we should consider.

Javascript Arrays

An array is a group of objects organized by an index number. JS gives us two ways to create an array:

1. var a = new Array();
2. var a = [];

Both of these methods are equivalent. What version 1 shows is that a JS array is just another type of object. Array() is the constructor function and methods associated with this constructor apply to any array object. If you were to create an array in this way and then query it in a Console window, you would get the following results:

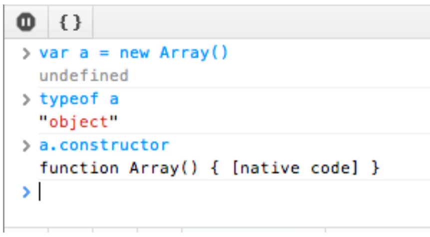

Figure 6.1: Querying an array object in a console window.

The second method is normally to be preferred. This is because it is always possible for someone (maybe you, more likely someone who has created code for or added code to an application) to re-define how the Array() constructor works, and here be dragons. Given the right set of circumstances, every item you add to an array using some unknown constructor could just be getting sent back to some scamming site: imagine the consequences if you were adding BankAccount passwords to the array.

It is not possible to subvert the [] method for declaring an array (at least, not within JS code), and so this method is much safer. If we wanted to create an array containing 4 integers, we would use:

```
var a  = [1, 79, 42, 36, 11];
```

Now, using the standard indexing, we can access any element of the array and apply array methods to it, as the following sequence of commands entered into the Chrome console shows:

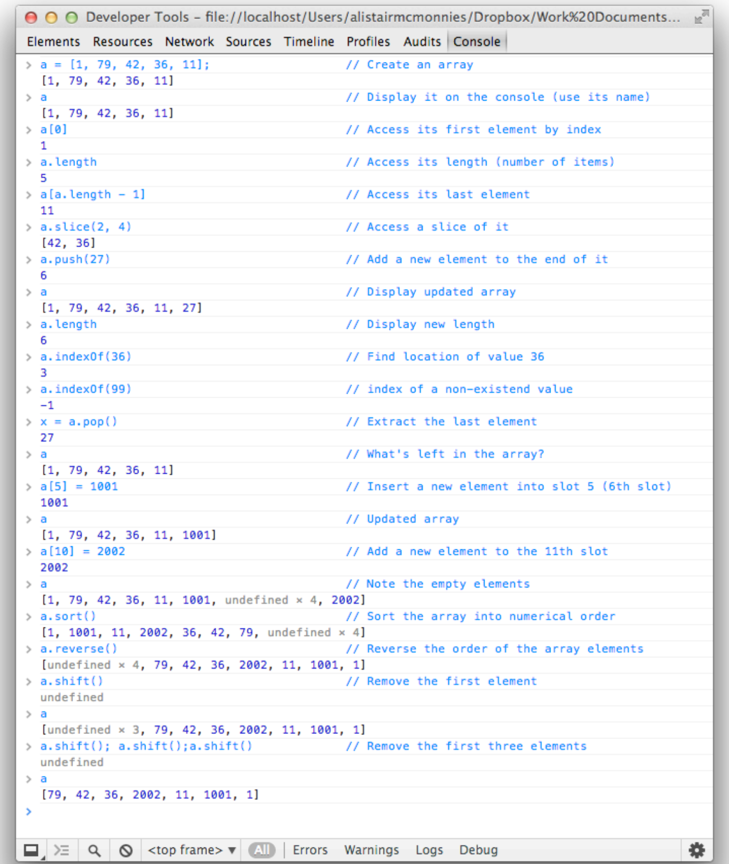

Figure 6.2: An array in the Console

Figure 6.2 shows an array being created and manipulated in the Chrome browser's Console window. I've added comments to indicate what each operation is doing with the array, so it should be fairly clear what is happening. In addition to creating the array and accessing individual elements:

- the **length** property returns the number of slots (i.e. locations where there could be elements) in the array
- the **length** property can be used to work out the index of the last element (the index of the last element in an array is always one less than length because the first index is 0)
- the **slice()** method is used to extract a sequence of elements (starting at index 2 and going up to, but not including, index 4)
- the **sort()** method has been used to sort it into ascending numerical order
- the **reverse()** method has reversed the order of all of the elements in the array
- the **push()** method adds a new element to the end of the array, and returns the new length
- the **indexOf()** method find an element and returns its position in the array (or -1 if it does not find it)
- the **pop()** method removes the last element of the array and returns it
- adding an element directly to a specific index, e.g. **a[7] = 105**, inserts the element at the given location, and extends the array up to this location if it is too short. Added elements up to the inserted one have the value ***undefined***
- the **shift()** method removes and returns the first element in the array; use it multiple times to remove several elements from the beginning

Note that the array indexes are 0-based. i.e., a[0] is the first element, and a[a.length-1] is the last. JS arrays have several more capabilities, but the most intriguing of them is that, JS arrays are ***polymorphic***. i.e. an array does not have to contain objects which are all of the same type. Another way of putting this is that a JS array is an array of Objects, with no restriction on the composition of the objects.

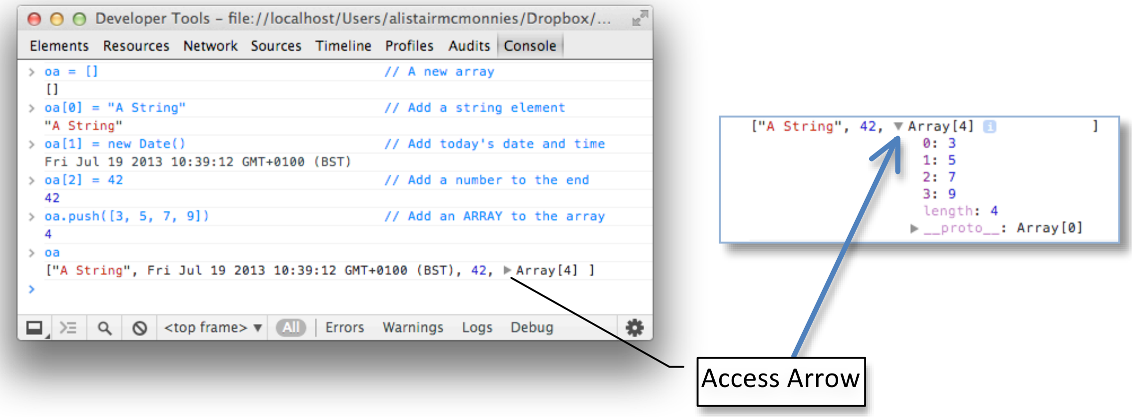

Figure 6.3: A polymorphic array

In figure 6.3, you can see a String, a Number, a Date object and another Array being added to an array (I've called it **oa** to indicate it is an Object Array). When asked to report on the contents, the Chrome Console displays the elements appropriately (note that the last element is shown as **Array[4]**, meaning a 4 element array). Click on the little access arrow to its left, and the Console will show the composition of the array element inside the main array object.

To access an element of the array within the array, simply use a second index in brackets. E.g. **oa[3][1]** is **5**, since 5 is the second element of the fourth element of the main array. It would now be possible to add this whole array into another array as an element, meaning that some of the contents would need three indices to access them. We could also add functions to an array as elements.

Given that we can create an array that contains any combination of elements, you might think there is little reason for having the Object type in Javascript. However, Objects have one further trick up their sleeve, in that instead of an index, an element (which is either a property or a method) can have a name. It would be quite awkward to use arrays as objects, since you would always need to associate a particular property or method with its index number; the code would be difficult to write and to read. Giving elements names makes them easier to identify.

Objects as Mappings (or Maps)

In computing terms, a map or mapping is a particular way of organising data: in a map there is a one-to-one correspondence between a given value (an object), and the key that can be used to access it (usually a String). We say that a value maps to a specific key. The name Associative Array is also used, since a map is like an array except that instead of a numeric index, values are accessed by a key that a programmer typically associates with the object or value being stored.

Javascript Objects are maps, in that each of the properties or methods within an object (all of which are also objects) has a name, which is the key to access that property or method. We can experiment with some literal object notation to demonstrate this:

Figure 6.4: A literal object – querying it as a mapping

In the console session in figure 6.4, you can see a block of statements at the top that defines a person object. The person has a name, address, telephone number, email address, date of birth, and an array of children[]. So far, so object like. It also contains methods to add a child, return the number of children and each individual child.

The code that follows this is almost like the code to access an array, except that instead of integer numbers, the index values are strings inside quote marks. If we try to access an element by number (e.g. person[0]), the console returns *undefined* as its value. Note that we can also use standard object dot notation (e.g. to access the addChild() method, I've used this), and that we can use either syntax for any member (e.g. ***person.child(0)*** is exactly equal to ***person["child"](0)***).

In this example, we're getting some insight into how Javascript manages objects and arrays. An array is simply an object that uses integers for indexes, and an object is simply a map of other objects (which can include functions). One limitation of an object map is that it does not have a length property. If you need to know how many items are it in you need to either keep track of the objects added and removed from it (you could use a numerical member to do that), or count the contents in a **for..in** loop:

```
var count = 0;
```

```
for(object in mapping){
 count += 1;
}
```

Listing 6.2: Counting the number of objects in a mapping

Because this is done in a Javascript loop rather than in native code, it is slow to count the number of items in a map, and will have to be done every time you need to know the number if it is possible that objects have been added or removed since the last time. For this reason, a true array is a better bet. Alternatively, you could create a custom collection that does this for you.

Custom Collections 1

We can make use of the object/map notation to create clever collections that are able to do useful work with their contents. For example, lets say we're creating a website that allows users to perform statistical calculations on data they enter or upload to the site. It would be sensible to encapsulate this functionality into an object type that dealt with the calculations, leaving us free to develop the appearance of the website and its user-interface without needing to consider how the statistics operate (two different skill-sets, that might be best done by two collaborating programmers – one a Web User-Interface expert and the other a maths wiz. Here's a simple object definition to get the statistics calculations going:

```
/**
 * A constructor and code for an object containing a set of statistical calculations.
 */
var Stats = function(){
    this.values = [];                             // We'll work on an array of numbers.
};

Stats.prototype.insertValue = function(v){
    this.values.push(v);                          // Adding a value to the array
};

Stats.prototype.insertValues = function(vals){
    this.values = this.values.concat(vals);       // Adding an array of values to the array
};

Stats.prototype.getValue = function (index){
        return this.values[index];
};

Stats.prototype.count = function(){
    return this.values.length;                    // How many values are there?
};

Stats.prototype.average = function(){             // To calculate the average...
    var total = 0, index, max;
    for(index = 0, max = this.values.length; index < max; index += 1){
        total += this.values[index];              // ...add all the values together...
    }
    return total / max;                           // ...and divide by their number.
};

// Stats object definition – continues over...
Stats.prototype.variance = function(){
    // The Variance is the average of the squared differences between the values and the
```

```
    // mean value.  To calculate this, square the difference between each value and the
    // average value, and add all these squares together.  Finally, divide by the number of
    // values...
    var avg = this.average(), sumDiffsSquared = 0, index, max, d;
    for(index=0, max = this.values.length; index<max; index +=1){
        d = (avg-this.values[index]);
        sumDiffsSquared += d*d;
    }
    return sumDiffsSquared / max;
};

Stats.prototype.standardDeviation = function() {
    // Standard Deviation is simply the square root of the variance…
    return Math.sqrt(this.variance());
};

// It would be very sensible to test this code on a known set of values.  I can have worked
// out the expected results on a scientific calculator or Excel to check...
var testStats = function(){
    var s = new Stats();
    s.insertValues([600, 470, 170, 430, 300]);
    console.log("Second value = " + s.getValue(1));             // Should be 470
    console.log("Count = " + s.count());                        // Should be 5
    console.log("Average = " + s.average());                    // Should be 394
    console.log("Variance = " + s.variance());                  // Should be 21704
    console.log("Standard Deviation = " + s.standardDeviation()); // Should be ~147.3
};

window.onload = testStats;   // Run the tests when the page loads.
```

Listing 6.3: A definition of a Stats object, which encapsulates some simple statistics

What we have gained from the definition of a statistics class is that we can incorporate statistical calculations on collections of numbers without even remembering how to do it – just include the JS code in a web application, and add a function call or several to insert the numerical collection that we want to analyse into the object. In this respect, we could consider our statistics object to be a set of utility functions, but it works only because of the customized collection we based it around.

We could easily augment the Stats collection object by incorporating functions that let us draw graphs and charts based on the values in it. We'll look into graphics in JS and HTML 5 in a later chapter.

Custom Collections 2

There are certain facilities that can be considered to make a collection class easier to use – being able to access individual elements, being able to find if an element exists in the collection, being able to add, remove one and remove all elements. Programmers with a background in all sorts of languages would expect these facilities to exist in a custom collection. In fact, once we had a collection that provided these features, we could use it as a base for any customised collection – in Javascript we simply need to add functions to the core object. Let's create this base collection, starting with a precise definition of how we expect it to operate:

Property or Method	**Example use**	**Description**
Constructor	var c = new Collection()	Constructor for a base collection

add() method	c.add(key, item);
	
	c.add(key, item, replace);
		1. Add an item and key pair to the collection
		
		2. Add an item:key pair. If the key exists and replace is true, replace the existing item; otherwise don't

count property	var x = c.count	Find out how many objects are in the collection
exists() method	if(c.exists("xyz"){ …	Returns true if an object with this key is in the collection already
item() method	var o = c.item("xyz");	Returns an item with the given key, or null if there is no such object
remove() method	c.remove("xyz");	Remove an item with the given key
removeAll() method	c.removeAll();	Removes everything from the collection
keys property	var k = c.keys	Returns an array of keys in the collection

Table 6.1: Methods and properties for a core collection class

We'll base our collection on a map.

```
/**
 * Collection: an object definition for managing key:value pairs of objects.
 * @constructor Collection: sets up a Collection objects with no items and count == 0
 */
var Collection = function(){
   "use strict";
   this.coll = {};    // Using a map.
   this.length = 0;
};

/**
 * Collection.add() function: adds an item to a collection with a key
 * @param key {Object} - typically a string
 * @param item {Object} - an item to add to the collection
 * @param replace {Boolean} - if true, any existing item with the given key will be replaced
 *                    if false, any existing item with the given key will not be replaced
 * @return {Boolean} - true if the item was added, or if it replaced an existing item
 */
Collection.prototype.add = function(key, item, replace){
   "use strict";
   // If the key exists and we've opted to replace the item, replace it..
   if(this.coll[key] && replace){
      this.coll[key] = item;
      return true;
   }
   // If the key exists, don't replace the item...
   if(this.coll[key]){
      return false;
   } else {   // Add the item and increment count...
      this.coll[key] = item;
      this.length += 1;
      return true;
   }
};

/**
 * Collection.item(key) - returns an item with the given key, or null
 * @param key {Object} - typically a string that uniquely identifies an item
 * @return {*}  - the object at the given key, or null
 */
Collection.prototype.item = function(key){
   "use strict";
   if(this.coll[key]){
      return this.coll[key];
   } else {
      return null;
   }
};

/**
 * Collection.exists(key) - test whether an item is stored uder the given key
 * @param key {Object} - key to test
 * @return {Boolean} - true if an item exists at that key, false otherwise
 */
Collection.prototype.exists = function(key){
   "use strict";
   if(this.coll[key]){
      return true;
   } else {
      return false;
   }
};

/**
 * Collection.count() - return the number of items in the collection
 * @return {number} - the count of items
 */
Collection.prototype.count = function(){
   "use strict";
   return this.length;
};

/**
 * Collection.remove(key) - remove the item with the given key
 * @param key - the key for the item to remove
 * @return {number} - the new length of the collection
 */
Collection.prototype.remove = function(key){
   "use strict";
   if(this.exists(key)){
      delete this.coll[key];
      this.length -= 1;
   }
   return this.length;
};

/**
 * Collection.removeAll() - deletes all collection contents and sets length to 0.
 */
Collection.prototype.removeAll = function(){
   "use strict";
   this.coll = {};
   this.length = 0;
};

/**
 * Collection.toArray() - return a standard array of the collection's items
 * @return {Array} - a new array of items
 */
Collection.prototype.toArray = function(){
   "use strict";
   var arr = [], o;
   for(o in this.coll){
      arr.push(this.coll[o]);
   }
   return arr;
};

/**
 * Collection.keys() - return an array of item keys from the collection.
 * @return {Array} - an array of key values.
 */
Collection.prototype.keys = function(){
   "use strict";
   var k = [], o;
   for(o in this.coll){
      k.push(o);
   }
   return k;
};
```

Listing 6.4: A custom Collection based on definitions in Table 6.1

Any time you create an object definition, especially one that you might use in a number of different applications, you should write some code to test the object.

```
/**
 * Test suite for this object definition.
 */
var collTest = function(){
    "use strict";

    // Run tests on all methods..
    var c = new Collection();

    console.log("*** add() tests***");
    if(c.add("one", "First item")){
        console.log("Pass: added 'one'");
    } else {
        console.log("Fail: 'one' not added");
    }
    if(c.add("one", "New First item", true)){
        console.log("Pass: updated 'one'");
    } else {
        console.log("Fail: 'one' not updated");
    }
    if(c.add("one", "First item")){
        console.log("Fail: updated 'one' when shouldn't have");
    } else {
        console.log("Pass: correctly didn't update 'one'");
    }
    if(c.add("two", "Second item")){
        console.log("Pass: added 'two'");
    } else {
        console.log("Fail: 'two' not added");
    }
```

```
    if(c.add("three", "Third item", true)){
        console.log("Pass: added 'three'");
    } else {
        console.log("Fail: 'three' not added");
    }
    if(c.add("four", "Fourth item", false)){
        console.log("Pass: added 'four'");
    } else {
        console.log("Fail: 'four' not added");
    }


    console.log("***item() tests***");
    var key = "one";
    var item = c.item(key);
    if(item){
        console.log("Pass: retrieved "+key+":"+item);
    } else {
        console.log("Fail: could not retrieve "+key);
    }
    key = "none";
    var item = c.item(key);
    if(item){
        console.log("Fail: wrongly retrieved " +
                              key+":"+item);
    } else {
        console.log("Pass: did not retrieve "+key);
    }


    console.log("***exists() tests***");
    var key = "one";
          if(c.exists(key)){
        console.log("Pass: states " + key + " exists.");
    } else {
        console.log("Fail: states " + key +
                              " does not exist.");
    }
    var key = "none";
    if(c.exists(key)){
        console.log("Fail: states " + key + " exists.");
    } else {
        console.log("Fail: states " + key +
                              " does not exist.");
    }
    console.log("***toArray() tests***");
    var a = c.toArray();
    if(a){
        console.log("Pass: retrieved " + a.join(', '));
    } else {
        console.log("Fail: array not returned");
    }

    console.log("***keys() tests***");
    var k = c.keys();
    if(k){
        console.log("Pass: retrieved " + k.join(', '));
    } else {
        console.log("Fail: keys not returned");
    }

    console.log("***count() tests***");
    var n = c.count();
    if(n){
        console.log("Pass: count is "+n);
    } else {
        console.log("Fail: count not returned");
    }

    console.log("***remove() tests***");
    var n2 = c.remove("two");
    if((n - n2) === 1){
        console.log("Pass: item successfully removed");
    } else {
        console.log("Fail: item not removed");
    }
    console.log("Status - values: " +
                           c.toArray().join(', '));
    console.log("Status - keys: " +
                           c.keys().join(', '));
    console.log("Status - count" + c.count() +
                           " items in collection");

    console.log("***removeAll() tests***");
    c.removeAll();
    if(c.count() == 0){
        console.log("Pass: all items removed");
    } else {
        console.log("Fail: items still in collection");
    }
    console.log("Status - final collection state = [" +
                      c.toArray().join(', ') + "]");
    console.log("Status - final keys state = ["
                       + c.keys().join(', ') + "]");
};

window.onload = collTest;  // Run the tests when the page loads.
```

Listing 6.4: A test suite for the Collection object type

On loading an HTML page that includes the two scripts in listing 6.3 and 6.4, we should see the following output in the browser console:

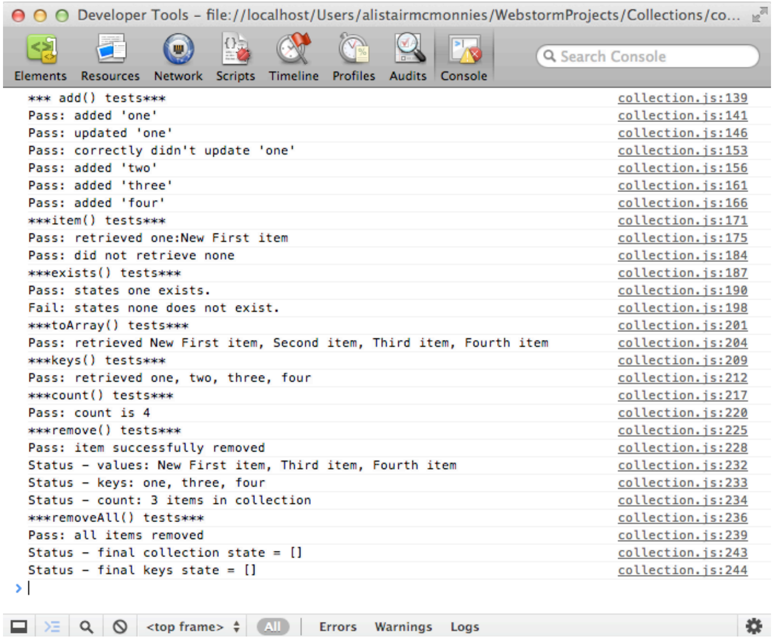

Figure 6.5: Test results shown in the Console window.

These results are fairly easy to interpret, although writing the test code was more effort that would ideally be necessary. We'll look at a software test tool that we can use with Javascript code to make testing code that we write easier (and therefore less likely to be omitted!) in a later chapter.

Using Data Structures in a jQuery Mobile App

One of the goals we've been working towards since early in this module is the separation of an applications user-interface (i.e. mark-up in HTML) and the code that makes the app do its job (i.e. the Javascript). There is nothing in the Javascript code in this chapter that is *specifically* for mobile apps – Javascript is pretty neutral in this respect, and so we could expect any of the code here to work exactly the same in a jQM application as it would in a normal web app.

We've not used any specific jQuery coding techniques either, but then jQuery is mainly about manipulating the DOM in a web-app, and jQuery mobile is mainly about rendering an app for small screens and the limited user-input methods available to phones and tablets, so this is no surprise either.

In a mobile app (or a normal web app for that matter), we use Javascript to model the Objects that the app is about (e.g. diary entries, notes, items in a game), and jQuery/jQM to manipulate the user-interface of the app to reflect these. This approach makes it easier to work on an application than if we tied all of the mark-up and code together. For example, the Shopping-List application from earlier in the module took this approach by having items that were `<li>..</li>` elements. If we decided to make shopping list elements more complex (e.g. each element containing a price and a quantity), we would certainly have got bogged down in a lot of detail about HTML mark-up that had little to do with the ***semantics*** of manipulating a list.

In a nutshell, the way to approach an application is to develop an object model separately in Javascript and then produce user-interface ***Views*** of the object model. This neatly separates the concerns of 'how it works' versus 'what it looks like' in a way that works well in code. In the next chapter, we'll go on to look at this approach in detail.

In this chapter we have discussed:

- How Javascript arrays are implemented and operated
- How the Object structure within Javascript uses a mapping structure to provide access to elements
- How to create custom collection classes that add some useful functionality to a simple collection

Web Sources

Advanced Javascript: Objects, Arrays and Array-like Objects - http://nfriedly.com/techblog/2009/06/advanced-javascript-objects-arrays-and-array-like-objects/

W3CSchools: Javascript Array Object - http://www.w3schools.com/jsref/jsref_obj_array.asp

A re-introduction to Javascript - https://developer.mozilla.org/en/A_re-introduction_to_JavaScript

Create your own collection object in Javascript: http://www.techrepublic.com/article/create-your-own-collection-objects-in-javascript/1044659

Exercises

1. Try to think of one good examples of where you could use an array in a program (an indexed list of similar objects) and another for using an object mapping (a list of objects indexed by name).
2. Assume you have need for a bookshelf object in a program. It will be used to store a collection of book objects. Think up a list of properties and methods you think a bookshelf should have – for example, the first property I'd think of would be ***count***, to indicate the number of book in the bookshelf.

Try out these exercises in a console window of a browser:

3. Create an array called ***emails*** with zero elements in it.
4. Add three elements – joe@bloggo.com, fred@smiddy.com and your own email address – to the array using the push() method
5. Enter an expression to show the second element in the array
6. Enter an expression to show how many elements are in the array
7. Add a new email (wilma@bedrock.com) to the array *at index location 5*
8. Display the whole array in the console
9. Enter an if() statement followed by an expression to show either a specific element (e.g. emails[3]) or "Empty" if there is no element there. Note the if() and expression can be all on a single line – it will also need an else part (to show "Empty")
10. Sort the array elements into alphabetical order and re-display it
11. Remove the last (empty) element using pop() and re-display the array
12. Write an expression to show the current index of joe@bloggo.com.