Part 7

Arrays

An array is a set of variables (e.g., strings or numbers) that are grouped together and given a single name.

For example, an array could be used to hold a set of strings representing the names of a group of people. It might be called *people* and hold (say) four different strings, each representing the name of a person:

Sarah Patrick Jane Tim

Items are held in an array in a particular order, usually the order in which they were added to the array. However, one of the great advantages of arrays is that the ordering can be changed in various ways. For example, the array above could easily be sorted so that the names are arranged in alphabetical order.

Creating Arrays

To create an array, a new Array object must be declared. This can be done in two ways:

```
var myArray = new Array("Sarah","Patrick","Jane","Tim");
```

Or:

```
var myArray = ["Sarah","Patrick","Jane","Tim"];
```

In the first example, the new `Array` object is declared explicitly.

In the second example, the array is declared in much the same way a string might be declared, except that square brackets (`[]`) are used at the beginning and end instead of quote-marks. The square brackets indicate to the JavaScript interpreter that this sequence of characters is being declared as an array.

Arrays are often used to hold data typed-in or otherwise collected from a user. Therefore, it may be necessary to create the array first and add data to it later. An empty array may be created in the following way:

```
var myArray = new Array();
```

The array thus created has no elements at all, but elements can be added as necessary later.

If it is not known exactly what data will be stored in the array, but the number of items is known, it may be appropriate to create an array of a specific size. This may be done in the following way:

```
var myArray = new Array(4);
```

The array thus created has four elements, all of which are empty. These empty elements can be filled with data later on.

Viewing and Modifying Array Elements

Suppose an array has been created using the following code:

```
var demoArray = new Array("Sarah","Patrick","Jane","Tim");
```

The number of elements in the array can be determined using the `length` property. For example:

```
alert(demoArray.length);
```

Get No. of Elements Click here to see this example working.

The entire contents of the array can be viewed using the `valueOf()` method. For example:

```
alert(demoArray.valueOf())
```

This piece of code will display an alert showing the entire contents of the array `demoArray`, with the various elements separated by commas.

Display Contents of Array Click here to see this example working.

The value of a particular element in the array can be obtained by using its position in the array as an index. For example

```
var indexNumber = prompt ("Please enter a number between 0 and
3","");
alert("Element " + indexNumber + " = " + demoArray[indexNumber]);
```

This piece of code will prompt the user to enter a number between 0 and 3 (the elements in an array are numbered from zero, so the four elements in this array will be numbered 0, 1, 2 and 3). It will then display the corresponding element from the array.

Choose & display Elelent Click here to see this example working.

It is also possible to change the value of an array element using its position in the array as an index. For example:

```
var newValue = prompt("Please enter your name","");
demoArray[0] = newValue;
```

```
alert(demoArray.valueOf());
```

This piece of code will prompt the user to enter their name, then place this string into element 0 of the array, over-writing the string previously held in that element. It will then display the modified array using the `valueOf()` method described earlier.

Change Element 0 Click here to see this example working.

Adding and Removing Elements

The `length` property of an array can be altered as well as read:

- ***increasing*** the length property adds extra (empty) elements onto the end of the array.
- ***decreasing*** the length property removes some of the existing elements from the end of the array.

Consider the following examples:

(1)
```
var currentLength = demoArray.length;
demoArray[currentLength] = "Fred";
alert(demoArray.valueOf());
```

This piece of code first determines the number of elements in the array using the `length` property. It then uses this information to identify the next element position *after* the end of the existing array and places the string "Fred" into that position, thus creating a new element. Finally, it displays the modified array using the `valueOf()` method described earlier.

Note that it isn't necessary to add 1 to the value of `length` in order to identify the next position in the array. This is because `length` indicates the actual number of elements, even though the elements are numbered from zero. For example, if an array has two elements, the value of `length` will be 2; however, those two elements will be numbered 0 and 1. Therefore, if `length` is used as an index, it will indicate the *third* element in the array, not the second one.

Add Extra Element Click here to see this example working.

(2)
```
var currentLength = demoArray.length;
demoArray.length = currentLength - 1;
alert(demoArray.valueOf());
```

This piece of code first determines the number of elements in the array using the `length` property. It then resets `length` to one less than its previous value, thus removing the last element in the array. Finally, it displays the modified array using the `valueOf()` method described earlier.

Remove Last Element Click here to see this example working.

There are also several methods that add and remove elements directly, some of which are listed below. However, it should be noted that these methods only work with Netscape Navigator, so it is generally preferable to use the methods described above since they work with most browsers.

`push()` Adds one or more elements onto the end of an array. For example:

```
var lastElement = demoArray.push("Fred", "Lisa");
```

This piece of code would add two new elements, "Fred" and "Lisa" onto the end of the array. The variable `lastElement` would contain the value of the last element added, in this case "Lisa".

`pop()` Removes the last element from the end of an array. For example:

```
var lastElement = demoArray.pop();
```

This piece of code would remove the last element from the end of the array and return it in the variable `lastElement`.

`unshift()` Adds one or more new elements to the *beginning* of an array, shifting the existing elements up to make room. Operates in a similar fashion to `push()`, above.

`shift()` Removes the first element from the *beginning* of an array, shifting the existing elements down to fill the space. Operates in a similar fashion to `pop()`, above.

Splitting and Concatenating Arrays

Arrays can be split and concatenated using the following methods:

`slice(x,y)` Copies the elements between positions `x` and `y` in the source array into a new array. For example:

```
newArray = demoArray.slice(0,2);
alert(newArray.valueOf());
```

This piece of code will copy elements 0 and 1 from the array called `demoArray` into a new array called `newArray`. It will then display the contents of `newArray` using the `valueOf()` method described earlier.

Note that the slice starts at `x` but stops at the last position *before* `y` rather than at position `y` itself.

Copy Slice to New Array Click here to see this example working.

`concat(array)` Concatenates the specified array and the array to which it is applied into a new array. For example:

```
combinedArray = demoArray.concat(newArray);
alert(combinedArray.valueOf());
```

This piece of code will concatenate `demoArray` and the new array created in the last example (`newArray`) to form another array called `combinedArray`. It will then display the contents of `combinedArray` using the `valueOf()` method described earlier.

Concatenate Arrays Click here to see this example working.

Rearranging Array Elements

The order of the elements in an array can be modified using the following methods:

`reverse()` Reverses the order of the elements within an array. For example:

```
demoArray.reverse();
alert(demoArray.valueOf());
```

This piece of code will reverse the order of the elements in the array, then display the re-ordered array using the `valueOf()` method described earlier.

Reverse Array Order Click here to see this example working.

`sort()` Sorts the elements within the array. Unless otherwise specified, the elements will be sorted alphabetically. For example:

```
demoArray.sort();
alert(demoArray.valueOf());
```

This piece of code will sort the elements of the array into alphabetical order, then display the re-ordered array using the `valueOf()` method described earlier.

Sort Array Alphabetically Click here to see this example working.

Although the `sort()` method normally sorts arrays alphabetically, it can be modified to sort in other ways. This is done by creating a special function and passing the name of that function to the `sort()` method as a parameter, e.g.:

```
demoArray.sort(bylength);
```

The sorting function must be written in such a way that it takes two elements of the array as parameters, compares them, then returns one of the following values indicating what order they should be placed in:

- **-1** : The first element should be placed *before* the second.
- **0** : The two elements are the same so the ordering does not matter.
- **1** : The first element should be placed *after* the second.

For example, here is a sorting function that sorts array elements by length:

```
function bylength(item1, item2)
{
        if(item1.length < item2.length)
        {
                return -1;
        };
        if(item1.length == item2.length)
        {
                return 0;
        };
        if(item1.length > item2.length)
        {
                return 1;
        };
}
```

In accordance with the rules for sorting functions, this example accepts two parameters, `item1` and `item2`. Each of these parameters will be an element of the array, passed to it by the `sort()` method.

Since the demonstration array contains strings, each element will have a length property. The length properties of the two parameters are compared using three `if` statements, and either -1, 0 or 1 is returned depending upon their relative lengths.

The `sort()` method will apply this function to each pair of strings in the array in turn until all have been sorted.

Sort Array Elements by Length Click here to see this example working.

Multi-Dimensional Arrays

The arrays described so far are *One-Dimensional Arrays*. They are effectively just lists.

Sometimes, however, we need to store information which has more than one dimension, for example, the scores in a game:

Sarah	18
Patrick	16
Jane	12
Tim	13

To store data of this type we use multi-dimensional arrays. In JavaScript this is done by using arrays as elements of other arrays.

For example, the game scores could be stored in the following way:

- Each person's data would be stored in a separate, two-element array (one element for the name and one element for the score).
- These four arrays (one for each person) would then be stored in a four-element array.

We could create such an array in the following way:

```
var person1 = new Array("Sarah", 18);
var person2 = new Array("Patrick", 16);
var person3 = new Array("Jane", 12);
var person4 = new Array("Tim", 13);

var scores = new Array (person1,person2,person3,person4);
```

To identify the individual elements within a multi-dimensional array, we use two index values, one after the other. The first refers to an element in the outer array (`scores` in this example), and the second refers to an element in the inner array (`person1, person2, person3` or `person4` in this example). For example:

```
function displayMDarray()
{
    for(x = 0; x <=3; x++)
    {
        alert(scores[x][0] + " has " + scores[x][1] + " points");
    }
}
```

In this example, the array is accessed using pairs of values in square brackets, e.g.:

```
scores[x][0]
```

The value in the first pair of square-brackets indicates one of the four 'person' arrays. The value in the second pair of square-brackets indicates one of the elements within that 'person' array, either the name (0) or the score (1).

The variable called `x` is incremented from 0 to 3 using a `for` loop. Therefore, `x` will point to a different one of the four 'person' arrays each time through the loop. The second value, 0 or 1, then selects either the name or score from within that array.

View Elements of Multi-Dimensional Array Click here to see this example working.

Multi-dimensional arrays can be accessed and modified using the same methods as one-dimensional arrays.

For example, a multi-dimensional array can be sorted using the `reverse()` and `sort()` methods in just the same way as a one-dimensional array, e.g.:

```
scores.sort()
```

Sort Elements of Multi-Dimensional Array Click here to sort the array, then click on the "View Elements of Multi-Dimensional Array" button to see the effects.

Note that the array is sorted by the *first* element of each sub-array, i.e., by names rather than scores.