•  [Creating arrays in JavaScript](http://www.dofactory.com/tutorial/javascript-arrays" \l "section1)

•  [Iterating over JavaScript arrays](http://www.dofactory.com/tutorial/javascript-arrays" \l "section2)

Creating arrays in JavaScript

1. // empty array with no elements
2. var empty = [];
3. // array with 2 string elements
4. var days = ["Sunday", "Monday"];
5. // array with elements of different types
6. var mixed = [false, 2.70, "Hello"];
7. // elements can be arbitrary expressions
8. var suffix = "tree";
9. var trees = ["Oak" + suffix, "Elm" + suffix];
10. // 2 dimensional array with object literals
11. var array2 = [[1,{x:1, y:2}], [2, {x:3, y:4}]];
12. // the 3rd element is undefined
13. var colors = ["Red", "Green", undefined];
14. // no value in the 1st element, it is undefined
15. var hobbies = [,"Art"];

Accessing array elements

1. var days = ["Sunday", "Monday"]; // an array with 2 elements
2. var firstDay = days[0]; // index 0 is converted to "0"
3. days[1] = "Ford";
4. days[2] = "Tuesday"; // writes element 3

Since an array is a subclass of object, there is nothing to stop you from adding properties on an array instance:

1. var cars = ["Toyota"];
2. cars["tires"] = 4;
3. alert(cars["tires"]); // => 4

Iterating over JavaScript arrays

Before reviewing iterating over an array we need to understand how to determine how many elements there are in an array. The closest we can get is JavaScript's length property.

The length of JavaScript arrays

JavaScript automatically keeps track of the length property that exists on each array. Unlike other languages, the length of a JavaScript array does not hold the array's upper bound.

1. var days = ["Sunday", "Monday"];
2. alert(days.length); // => 2
3. var cars = [];
4. cars[1] = "Honda";
5. cars[3] = "Fiat";
6. alert(cars.length); // => 4

[Run](javascript:void(0);)

In this example, the length property of the cars array is greater than the number of elements. The value length is 4, but the actual number of array elements is just 2. So, how does this work? The rule with length is that it is the last numerical index plus one. An array with gaps in its elements is called a *sparse* array.

Here is another unusual case: an element with the key six is added to the above script, but is ignored by the length property.

1. cars["six"] = "Volkswagen";
2. alert(cars.length); // => still returns 4

These examples also demonstrate that JavaScript arrays are dynamically sized, meaning that you will never see array out-of-bound errors.

Iterating over array elements

With an irregular array how do we iterate over all its elements? Let's first consider using for loops.

1. var cars = [];
2. cars[1] = "Ford";
3. cars[3] = "BMW";
4. cars["six"] = "Honda";
5. for (var i = 0; i < cars.length; i++) {
6. alert(cars[i]); // => undefined, Ford, undefined, BMW
7. }
8. // To skip missing and undefined elements, add this condition.
9. for (var i = 0; i < cars.length; i++) {
10. if (cars[i] === undefined) // skip undefined elements
11. continue;
12. alert(cars[i]); // => Ford, BMW
13. }

Notice that the array element with key six won't appear when iterating using a for-loop.

To see all index values we need to use a for-in loop, like so:

1. var cars = [];
2. cars[1] = "Ford";
3. cars[3] = "BMW";
4. cars["six"] = "Honda";
5. for (var index in cars) {
6. alert(cars[index]); // Ford, BMW, Honda
7. }

There are a couple of things to be aware of when using a for-in loop on arrays. First, if you are expecting the elements of your array to appear in numerical order, don't trust the for-in loop. They can appear in any order. Secondly, the for-in loop not only returns all array properties (elements), but it also returns the properties that are inherited through the prototype chain.

With both the length property and the for-in loop being somewhat peculiar, it is best to *always* use numeric index values starting with 0 going up. In that case arrays will behave similar to most other languages and the length property as well as the for-loop will work as you would expect. Here is an example:

1. var cars = [];
2. cars[0] = "Ford";
3. cars[1] = "BMW";
4. cars[2] = "Honda";
5. alert(cars.length); // => 3
6. for (var i = 0, len = cars.length; i < len; i++) {
7. alert(cars[i]); // => Ford, BMW, Honda
8. }

Multi-Dimensional Arrays

JavaScript does not natively support arrays of more than one dimension; you have to create them yourselves. Fortunately, they are easy to model with arrays of arrays. In the next example we create a 5 by 5 identity matrix with a value of 1 on diagonal elements and 0 everywhere else.

1. var twoDim = [];
2. for (var row = 0; row < 5; row++) {
3. var oneDim = [];
4. for (var col = 0; col < 5; col++) {
5. oneDim[col] = (row === col) ? 1 : 0; // 0 or 1 (diag)
6. }
7. twoDim[row] = oneDim;
8. }
9. alert(twoDim[4][2]); // => 0
10. alert(twoDim[3][3]); // => 1

The last two statements also demonstrate how you would access elements in a 2-dimensional array: simply use the [] operator twice, the first for the row and second for the column index.

Deleting array elements with delete

To remove an element from an array you can use the delete operator. Deleting an element does not affect the length property and the array becomes sparse. Also, elements with higher indexes to the right of the deleted element do not get shifted down to fill in the gap.

1. var days = ["Sunday", "Monday", "Tuesday", "Wednesday"];
2. delete days[2]; // => delete the element at index 2
3. alert(days[2]); // => undefined
4. alert(days.length); // => still 4

An alternative to delete is the built-in array method splice(). The difference with the delete operator is that splice() does not make the array sparse and shifts the elements to higher or lower positions as necessary making sure that no gap is left. This is discussed in the next section.