Javascript Array: How To Use Arrays in

Arrays were an add-on afterthought to javascript; for that reason you can find various ways of using them listed on programming websites, including references to an **Array** object.

The 'Array' object is however basically obsolete now; in this tutorial we'll look at current best practices.

The code below won't work on very old browsers; however, javascript has been stable enough for long enough now that I suggest disregarding any browsers too old to run the following code. Your user base is unlikely to be affected much. Those who are using browsers too old to run such basic javascript, really need to upgrade their browsers.

Creating and Initializing Arrays In Javascript

You can create an array in javascript as follows (in this example you can see an entire .html file; later on we'll just look at the javascript part):

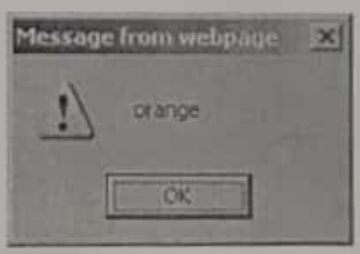
```
<html>

<head>
<title>Basic Javascript Array Example</title>

<script language="javascript">
// Empty array
var empty = [];

// Array containing initial elements.
var fruits = ['apple', 'orange', 'kiwi'];

alert(fruits[1]);
</script>
</head>
<body>
</body>
</html>
```



To add elements to an array in javascript, just define a new element at the end of the array.

You can also use **push()** to 'push' a new element onto the end of the array, or **unshift()** to add elements to the start of an array:

```
// Array containing initial elements.
var fruits = ['apple', 'orange', 'kiwi'];
// Adding new elements to the end of
// the array:
fruits[3] = 'banana';
fruits[4] = 'pear';

// Or you can just do this ...
fruits[fruits.length] = 'pineapple';
fruits[fruits.length] = 'cherry';

// Or this:
fruits.push('grape');
fruits.push('raspberry');

// Or you can add elements to the START
// of the array like this:
fruits.unshift('raspberry');
fruits.unshift('blackberry');
```

Multidimensional Arrays in Javascript

You can also create multidimensional arrays. The sub-arrays do not have to be all the same length.

```
var stuff = [
    [1, 2, 3],
    ['one', 'two', 'three'],
    ['apple', 'orange', 'banana', 'kiwi']
];

alert(stuff[2][1]);
```

Iterating Through Arrays in Javascript

You can iterate through an array in javascript using a **for** loop with a counter that runs from zero to one less than the length of the array.

Here we create an array called 'fruits'; we then pass the array to a function that iterates through the array and writes the array members to the HTML document body, one per line:

```
function show_array(array) {
    for(var i=0; i<array.length; i++) {
        document.write(array[i]);
        document.write('<br/>');
    }
}

var fruits = ['apple', 'orange', 'banana'];

show_array(fruits);
```

```
apple
orange
banana
```

You can also use **for** in a 'for each' kind of a way, meaning that we could have chosen to write the above function, ***showarray()*** *as follows (here I've also modified show*array() so that it shows the array contents in an alert box rather than writing to the document):

```
function show_array(array) {

    var text = '';
```

```
    for(var i in array) {
        text += array[i];
        text += 'n';
    }

    alert(text);
}
```

Note that i is still the item index, not the item itself.

For genuine 'for each'-type loops in javascript, you can use one of the several popular javascript libraries; for instance, jQuery defines an 'each' iterator that gets you the elements of an array one by one.

Javascript Array Sort: Sorting Arrays in Javascript

To sort an array in javascript, use the **sort()** function. You can only use sort() by itself to sort arrays in ascending alphabetical order; if you try to apply it to an array of numbers, they will get sorted alphabetically.

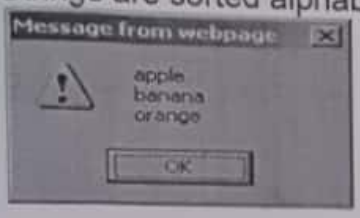
```
var fruits = ['apple', 'orange', 'banana'];
var numbers = [10, 20, 2, 3, 0, 500];

// This works.
fruits.sort();

// This sorts alphabetically, not numerically!
numbers.sort();

// We defined show array earlier.
show_array(fruits);
show_array(numbers);
```

Strings are sorted alphabetically:



But so are numbers:

To sort numbers, or to sort strings non-alphabetically, or to sort an array containing something else altogether, you must supply your own sort algorithm. This is done by passing the name of a function containing a sort algorithm to sort(). You can also just define the sort algorithm function 'inline', as in the following example.

The sort algorithm function receives two arguments; these are array elements that must be compared. If the first element is bigger than the second (in other words, comes later in the sort order), return 1. If the first element is smaller than the second (comes earlier in the sort order), return -1. If the elements are equal, return 0.

```
var fruits = ['apple', 'orange', 'banana'];
var numbers = [10, 20, 2, 3, 0, 500];

// Sort in reverse alphabetical order.
fruits.sort(function(a, b) {
    if(a > b) {
        return -1;
    }
    else if(a < b) {
        return 1;
    }
    else {
        return 0;
    }
});
```

```
// Sort in ascending numerical order.
numbers.sort(function(a, b) {
    if(a > b) {
        return 1;
    }
    else if(a < b) {
        return -1;
    }
    else {
        return 0;
    }
});

// We defined show array earlier.
show_array(fruits);
show_array(numbers);
```

The fruits array is now sorted in reverse alphabetical order:

The numbers array is sorted numerically:

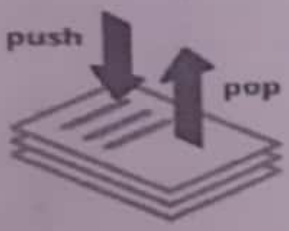
It supports two operations:

`push` adds an element to the end.

`pop` takes an element from the end.

So new elements are added or taken always from the "end".

A stack is usually illustrated as a pack of cards: new cards are added to the top or taken from the top:



For stacks, the latest pushed item is received first, that's also called LIFO (Last-In-First-Out) principle. For queues, we have FIFO (First-In-First-Out).

Arrays in JavaScript can work both as a queue and as a stack. They allow to add/remove elements both to/from the beginning or the end.

Types

JavaScript arrays come in different forms and this post will explain what the difference is between each array type. This post will look at the following array types;

- Homogeneous arrays
- Heterogeneous arrays
- Multidimensional arrays
- Jagged arrays

Homogeneous Arrays

As the name may suggest a homogeneous array is an array that stores a single data type(string, int or Boolean values).

```
var array = ["Matthew", "Simon", "Luke"];

var array = [27, 24, 30];

var array = [true, false, true];
```

Heterogeneous Arrays

A heterogeneous array is the opposite to a homogeneous array. Meaning it can store mixed data types.

```
var array = ["Matthew", 27, true];
```

Multidimensional Arrays

Also known as an array of arrays, multidimensional arrays allow you to store arrays within arrays, a kind of "array-ception".

```
var array = [["Matthew", "27"], ["Simon", "24"], ["Luke", "30"]];
```

Jagged Arrays

Jagged arrays are similar to multidimensional array with the exception being that a jagged array does not require a uniform set of data.

```
var array = [

    ["Matthew", "27", "Developer"],

    ["Simon", "24"],

    ["Luke"]

];
```

as you can see, each array within the overall array is not equal, the first array has three items stored in it, while the second has two and the third has just one. Share

OBJECTS

In JavaScript, almost "everything" is an object.

- Booleans can be objects (if defined with the **new** keyword)
- Numbers can be objects (if defined with the **new** keyword)
- Strings can be objects (if defined with the **new** keyword)
- Dates are always objects
- Maths are always objects
- Regular expressions are always objects
- Arrays are always objects
- Functions are always objects
- Objects are always objects

JavaScript values, except primitives, are objects.

JavaScript Primitives

A **primitive value** is a value that has no properties or methods.

A **primitive data type** is data that has a primitive value.

JavaScript defines 5 types of primitive data types:

- string
- number
- boolean
- null
- undefined

Primitive values are immutable (they are hardcoded and therefore cannot be changed).

JavaScript Objects

You have already learned that JavaScript variables are containers for data values.

This code assigns a **simple value** (Fiat) to a **variable** named car:

```
var car = "Fiat";
```

Objects are variables too. But objects can contain many values.

This code assigns **many values** (Fiat, 500, white) to a **variable** named car:

```
var car = {type:"Fiat", model:"500", color:"white"};
```

The values are written as **name:value** pairs (name and value separated by a colon).

JavaScript objects are containers for **named values**.

Object Properties

The name:values pairs (in JavaScript objects) are called **properties**.

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

Property	Property Value
firstName	John

lastName	Doe
age	50
eyeColor	blue

Object Methods

Methods are **actions** that can be performed on objects.

Methods are stored in properties as **function definitions**.

Property	Property Value
firstName	John
lastName	Doe
age	50
eyeColor	blue
fullName	function() {return this.firstName + " " + this.lastName;}

JavaScript objects are containers for named values called properties or methods.

Object Definition

You define (and create) a JavaScript object with an object literal:

Example

```
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
```

Spaces and line breaks are not important. An object definition can span multiple lines:

Example

```
var person = {
    firstName:"John",
    lastName:"Doe",
    age:50,
    eyeColor:"blue"
};
```

Accessing Object Properties

You can access object properties in two ways:

objectName.propertyName

or

objectName["propertyName"]

Example1

```
person.lastName;
```

Example2

```
person["lastName"];
```

Accessing Object Methods

You access an object method with the following syntax:

objectName.methodName()

Example

```
name = person.fullName();
```

If you access a method **without ()**, it will return the **function definition**:

Example

```
name = person.fullName;
```