Array

The JavaScript **`Array`** object is a global object that is used in the construction of arrays; which are high-level, list-like objects.

Create an Array

```
var fruits = ['Apple', 'Banana'];

console.log(fruits.length);
```

Access (index into) an Array item

```
var first = fruits[0];


var last = fruits[fruits.length - 1];
```

Loop over an Array

```
fruits.forEach(function(item, index, array) {
  console.log(item, index);
});
```

Add to the end of an Array

```
var newLength = fruits.push('Orange');
```

Remove from the end of an Array

```
var last = fruits.pop();
```

Remove from the front of an Array

```
var first = fruits.shift();
```

Add to the front of an Array

```
var newLength = fruits.unshift('Strawberry')
```

Find the index of an item in the Array

```
fruits.push('Mango');


var pos = fruits.indexOf('Banana');
```

Remove an item by index position

```
var removedItem = fruits.splice(pos, 1);
```

Remove items from an index position

```
var vegetables = ['Cabbage', 'Turnip', 'Radish', 'Carrot'];
console.log(vegetables);


var pos = 1, n = 2;

var removedItems = vegetables.splice(pos, n);



console.log(vegetables);
```

```
console.log(removedItems);
```

Copy an Array

```
var shallowCopy = fruits.slice();
```

Syntax

```
[element0, element1, ..., elementN]
new Array(element0, element1[, ...[, elementN]])
new Array(arrayLength)
```

Parameters

`elementN`

A JavaScript array is initialized with the given elements, except in the case where a single argument is passed to the `Array` constructor and that argument is a number (see the arrayLength parameter below). Note that this special case only applies to JavaScript arrays created with the `Array` constructor, not array literals created with the bracket syntax.

`arrayLength`

If the only argument passed to the `Array` constructor is an integer between 0 and 2^{32}-1 (inclusive), this returns a new JavaScript array with its `length` property set to that number (**Note:** this implies an array of `arrayLength` empty slots, not slots with actual `undefined` values). If the argument is any other number, a `RangeError` exception is thrown.

Description

Arrays are list-like objects whose prototype has methods to perform traversal

and mutation operations. Neither the length of a JavaScript array nor the types of its elements are fixed. Since an array's length can change at any time, and data can be stored at non-contiguous locations in the array, JavaScript arrays are not guaranteed to be dense; this depends on how the programmer chooses to use them. In general, these are convenient characteristics; but if these features are not desirable for your particular use, you might consider using typed arrays.

Arrays cannot use strings as element indexes (as in an associative array), but must use integers. Setting or accessing via non-integers using bracket notation (or dot notation) will not set or retrieve an element from the array list itself, but will set or access a variable associated with that array's object property collection. The array's object properties and list of array elements are separate, and the array's traversal and mutation operations cannot be applied to these named properties.

Accessing array elements

JavaScript arrays are zero-indexed: the first element of an array is at index `0`, and the last element is at the index equal to the value of the array's `length` property minus 1.

```
var arr = ['this is the first element', 'this is the second element'];
console.log(arr[0]);
console.log(arr[1]);
console.log(arr[arr.length - 1]);
```

Array elements are object properties in the same way that `toString` is a property, but trying to access an element of an array as follows throws a syntax error, because the property name is not valid:

```
console.log(arr.0);
```

There is nothing special about JavaScript arrays and the properties that cause this. JavaScript properties that begin with a digit cannot be referenced with dot notation; and must be accessed using bracket notation. For example, if you had an object with a property named `'3d'`, it can only be referenced using bracket notation. E.g.:

```
var years = [1950, 1960, 1970, 1980, 1990, 2000, 2010];
console.log(years.0);
console.log(years[0]);

renderer.3d.setTexture(model, 'character.png');
renderer['3d'].setTexture(model, 'character.png');
```

Note that in the `3d` example, `'3d'` had to be quoted. It's possible to quote the JavaScript array indexes as well (e.g., `years['2']` instead of `years[2]`), although it's not necessary. The `2` in `years[2]` is coerced into a string by the JavaScript engine through an implicit `toString` conversion. It is for this reason that `'2'` and `'02'` would refer to two different slots on the `years` object and the following example could be `true`:

```
console.log(years['2'] != years['02']);
```

Similarly, object properties which happen to be reserved words(!) can only be accessed as string literals in bracket notation(but it can be accessed by dot notation in firefox 40.0a2 at least):

```
var promise = {
  'var'  : 'text',
  'array': [1, 2, 3, 4]
};
```

```
console.log(promise['var']);
```

Relationship between `length` and numerical properties

A JavaScript array's `length` property and numerical properties are connected. Several of the built-in array methods (e.g., `join`, `slice`, `indexOf`, etc.) take into account the value of an array's `length` property when they're called. Other methods (e.g., `push`, `splice`, etc.) also result in updates to an array's `length` property.

```
var fruits = [];
fruits.push('banana', 'apple', 'peach');

console.log(fruits.length);
```

When setting a property on a JavaScript array when the property is a valid array index and that index is outside the current bounds of the array, the engine will update the array's `length` property accordingly:

```
fruits[5] = 'mango';
console.log(fruits[5]);
console.log(Object.keys(fruits));
console.log(fruits.length);
```

Increasing the `length`.

```
fruits.length = 10;
console.log(Object.keys(fruits));
console.log(fruits.length);
```

Decreasing the `length` property does, however, delete elements.

```
fruits.length = 2;
console.log(Object.keys(fruits));
console.log(fruits.length);
```

This is explained further on the `Array.length` page.

Creating an array using the result of a match

The result of a match between a regular expression and a string can create a JavaScript array. This array has properties and elements which provide information about the match. Such an array is returned by `RegExp.exec`, `String.match`, and `String.replace`. To help explain these properties and elements, look at the following example and then refer to the table below:

```
var myRe = /d(b+)(d)/i;
var myArray = myRe.exec('cdbBdbsbz');
```

The properties and elements returned from this match are as follows:

Property/Element	Description	Example
`input`	A read-only property that reflects the original string against which the regular expression was matched.	cdbBdbsbz
`index`	A read-only property that is the zero-based index of the match in the string.	1
`[0]`	A read-only element that specifies the last matched characters.	dbBd
`[1], ...[n]`	Read-only elements that specify the parenthesized substring matches, if included in the regular expression. The number of possible parenthesized	[1]: bB [2]: d

	substrings is unlimited.	

Properties

`Array.length`

The `Array` constructor's length property whose value is 1.

`get Array[@@species]`

The constructor function that is used to create derived objects.

`Array.prototype`

Allows the addition of properties to all array objects.

Methods

`Array.from()`

Creates a new `Array` instance from an array-like or iterable object.

`Array.isArray()`

Returns true if a variable is an array, if not false.

`Array.of()`

Creates a new `Array` instance with a variable number of arguments, regardless of number or type of the arguments.

`Array` instances

All `Array` instances inherit from `Array.prototype`. The prototype object of the `Array` constructor can be modified to affect all `Array` instances.

Properties

Methods

Mutator methods

Accessor methods

Iteration methods

Several methods take as arguments functions to be called back while processing the array. When these methods are called, the `length` of the array is sampled, and any element added beyond this length from within the callback is not visited. Other changes to the array (setting the value of or deleting an element) may affect the results of the operation if the method visits the changed element afterwards. While the specific behavior of these methods in such cases is well-defined, you should not rely upon it so as not to confuse others who might read your code. If you must mutate the array, copy into a new array instead.

`Array.prototype.entries()`

Returns a new `Array Iterator` object that contains the key/value pairs for each index in the array.

`Array.prototype.every()`

Returns true if every element in this array satisfies the provided testing function.

`Array.prototype.filter()`

Creates a new array with all of the elements of this array for which the provided filtering function returns true.

`Array.prototype.find()`

Returns the found value in the array, if an element in the array satisfies the provided testing function or `undefined` if not found.

`Array.prototype.findIndex()`

Returns the found index in the array, if an element in the array satisfies the provided testing function or -1 if not found.

`Array.prototype.forEach()`

Calls a function for each element in the array.

`Array.prototype.keys()`

Returns a new `Array Iterator` that contains the keys for each index in

the array.

`Array.prototype.map()`

Creates a new array with the results of calling a provided function on every element in this array.

`Array.prototype.reduce()`

Apply a function against an accumulator and each value of the array (from left-to-right) as to reduce it to a single value.

`Array.prototype.reduceRight()`

Apply a function against an accumulator and each value of the array (from right-to-left) as to reduce it to a single value.

`Array.prototype.some()`

Returns true if at least one element in this array satisfies the provided testing function.

`Array.prototype.values()`

Returns a new `Array Iterator` object that contains the values for each index in the array.

`Array.prototype[@@iterator]()`

Returns a new `Array Iterator` object that contains the values for each index in the array.

`Array` generic methods

Array generics are non-standard, deprecated and will get removed in the near future.

Sometimes you would like to apply array methods to strings or other array-like objects (such as function arguments). By doing this, you treat a string as an array of characters (or otherwise treat a non-array as an array). For example, in order to check that every character in the variable *str* is a letter, you would write:

```
function isLetter(character) {
  return character >= 'a' && character <= 'z';
}

if (Array.prototype.every.call(str, isLetter)) {
  console.log("The string '" + str + "' contains only letters!");
}
```

This notation is rather wasteful and JavaScript 1.6 introduced a generic shorthand:

```
if (Array.every(str, isLetter)) {
  console.log("The string '" + str + "' contains only letters!");
}
```

Generics are also available on `String`.

These are **not** part of ECMAScript standards and they are not supported by non-Gecko browsers. As a standard alternative, you can convert your object to a proper array using `Array.from()`; although that method may not be supported in old browsers:

```
if (Array.from(str).every(isLetter)) {
  console.log("The string '" + str + "' contains only letters!");
}
```

Examples

Creating an array

The following example creates an array, `msgArray`, with a length of 0, then assigns values to `msgArray[0]` and `msgArray[99]`, changing the length of the array to 100.

```
var msgArray = [];
msgArray[0] = 'Hello';
msgArray[99] = 'world';

if (msgArray.length === 100) {
  console.log('The length is 100.');
}
```

Creating a two-dimensional array

The following creates a chess board as a two dimensional array of strings. The first move is made by copying the 'p' in (6,4) to (4,4). The old position (6,4) is made blank.

```
var board = [
  ['R','N','B','Q','K','B','N','R'],
  ['P','P','P','P','P','P','P','P'],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' ',' '],
  ['p','p','p','p','p','p','p','p'],
  ['r','n','b','q','k','b','n','r'] ];

console.log(board.join('\n') + '\n\n');


board[4][4] = board[6][4];
board[6][4] = ' ';
console.log(board.join('\n'));
```

Here is the output:

```
R,N,B,Q,K,B,N,R
P,P,P,P,P,P,P,P
 , , , , , , ,
 , , , , , , ,
```

```
 , , , , , , ,
 , , , , , , ,
p,p,p,p,p,p,p,p
r,n,b,q,k,b,n,r

R,N,B,Q,K,B,N,R
P,P,P,P,P,P,P,P
 , , , , , , ,
 , , , , , , ,
 , , , ,p, , ,
 , , , , , , ,
p,p,p,p, ,p,p,p
r,n,b,q,k,b,n,r
```

Using an array to tabulate a set of values

```
values = [];
for (var x = 0; x < 10; x++){
 values.push([
  2 ** x,
  2 * x ** 2
 ])
};
console.table(values)
```

Results in

```
0       1       0
1       2       2
2       4       8
3       8       18
4       16      32
5       32      50
6       64      72
7       128     98
8       256     128
9       512     162
```

(First column is the (index))

Specifications

Specification	Status	Comment
ECMAScript 1st Edition (ECMA-262)	Standard	Initial definition.
ECMAScript 5.1 (ECMA-262) The definition of 'Array' in that specification.	Standard	New methods added: `Array.isArray`, `indexOf`, `lastIndexOf`, `every`, `some`, `forEach`, `map`, `filter`, `reduce`, `reduceRight`
ECMAScript 2015 (6th Edition, ECMA-262) The definition of 'Array' in that specification.	Standard	New methods added: `Array.from`, `Array.of`, `find`, `findIndex`, `fill`, `copyWithin`
ECMAScript Latest Draft (ECMA-262) The definition of 'Array' in that specification.	Living Standard	New method added: `Array.prototype.includes()`

Browser compatibility

1. Available in Firefox Nightly only due to compatibility issues.

2. The `@@iterator` symbol is implemented.

3. A placeholder property named `@@iterator` is used.

4. Supported as `@@iterator`.

5. Supported as `iterator`.

See also

- JavaScript Guide: “Indexing object properties”

- JavaScript Guide: “Predefined Core Objects: `Array` Object”
- Array comprehensions
- Polyfill for JavaScript 1.8.5 Array Generics and ECMAScript 5 Array Extras
- Typed Arrays

Was this article helpful?

Thank you!