On this Page

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

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.

Common operations

Create an Array

```
let fruits = ['Apple', 'Banana']

console.log(fruits.length)
// 2
```

Access (index into) an Array item

```
let first = fruits[0]
// Apple

let last = fruits[fruits.length - 1]
// Banana
```

Loop over an Array

```
fruits.forEach(function(item, index, array) {
  console.log(item, index)
})
// Apple 0
// Banana 1
```

Add to the end of an Array

```
let newLength = fruits.push('Orange')
// ["Apple", "Banana", "Orange"]
```

Remove from the end of an Array

```
let last = fruits.pop() // remove Orange (from the end)
// ["Apple", "Banana"]
```

Remove from the front of an Array

```
let first = fruits.shift() // remove Apple from the front
// ["Banana"]
```

Add to the front of an Array

```
let newLength = fruits.unshift('Strawberry') // add to the front
// ["Strawberry", "Banana"]
```

Find the index of an item in the Array

```
fruits.push('Mango')
// ["Strawberry", "Banana", "Mango"]

let pos = fruits.indexOf('Banana')
// 1
```

Remove an item by index position

```
let removedItem = fruits.splice(pos, 1) // this is how to remove an item

// ["Strawberry", "Mango"]
```

Remove items from an index position

```
let vegetables = ['Cabbage', 'Turnip', 'Radish', 'Carrot']
console.log(vegetables)
// ["Cabbage", "Turnip", "Radish", "Carrot"]

let pos = 1
let n = 2

let removedItems = vegetables.splice(pos, n)
// this is how to remove items, n defines the number of items to be removed,
// starting at the index position specified by pos and progressing toward the end
of array.

console.log(vegetables)
// ["Cabbage", "Carrot"] (the original array is changed)

console.log(removedItems)
// ["Turnip", "Radish"]
```

Copy an Array

```
let shallowCopy = fruits.slice() // this is how to make a copy
// ["Strawberry", "Mango"]
```

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.

Using an invalid index number returns `undefined`.

```
let arr = ['this is the first element', 'this is the second element', 'this
is the last element']
console.log(arr[0])               // logs 'this is the first element'
```

```
console.log(arr[1])              // logs 'this is the second element'
console.log(arr[arr.length - 1]) // logs 'this is the last element'
```

Array elements are object properties in the same way that `toString` is a property (to be specific, however, `toString()` is a method). Nevertheless, 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) // a syntax error
```

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.

```
let years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
console.log(years.0)   // a syntax error
console.log(years[0])  // works properly
```

```
renderer.3d.setTexture(model, 'character.png')     // a syntax error
renderer['3d'].setTexture(model, 'character.png')  // works properly
```

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. As a result, `'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'])
```

Relationship between 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.

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

console.log(fruits.length) // 3
```

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])           // 'mango'
console.log(Object.keys(fruits)) // ['0', '1', '2', '5']
console.log(fruits.length)       // 6
```

Increasing the `length`.

```
fruits.length = 10
console.log(Object.keys(fruits)) // ['0', '1', '2', '5']
console.log(fruits.length)       // 10
```

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

```
fruits.length = 2
console.log(Object.keys(fruits)) // ['0', '1']
console.log(fruits.length)       // 2
```

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 `RegExp` 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, see this example and then refer to the table below:

```
// Match one d followed by one or more b's followed by one d
// Remember matched b's and the following d
// Ignore case

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

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

Property/Element	Description	Example
`input` [redacted] only	The original string against which the regular expression was matched.	`"cdbBdbsbz"`
`index` [redacted] only	The zero-based index of the match in the string.	`1`
`[0]` [redacted] only	The last matched characters.	`"dbBd"`
`[1], ...[n]` [redacted] only	Elements that specify the parenthesized substring matches (if included) in the regular expression. The number of possible parenthesized substrings is unlimited.	`[1]: "bB"` `[2]: "d"`

Constructor

`Array()`

Creates `Array` objects.

Static 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.

Static methods

`Array.from(arrayLike[, mapFn[, thisArg]])`

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

`Array.isArray(value)`

Returns `true` if *`value`* is an array, or `false` otherwise.

`Array.of(element0[, element1[, ...[, elementN]]])`

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

`Array` instances

`Array` instances inherit from `Array.prototype`. As with all constructors, you can change the constructor's `prototype` property to make changes to all `Array` instances. For example, you

can add new methods and properties to extend all `Array` objects. This is used for polyfilling, for example.

However, adding non-standard methods to the array object can cause issues later, either with your own code, or when adding features to JavaScript.

Little known fact: `Array.prototype` itself is an `Array`:

```
Array.isArray(Array.prototype) // true
```

Instance properties

`Array.prototype.constructor`
Specifies the function that creates an object's `prototype`.

`Array.prototype.length`
Reflects the number of elements in an array.

`Array.prototype[@@unscopables]`
A symbol containing property names to exclude from a `with` binding scope.

Instance methods

Mutator methods

These methods modify the array:

`Array.prototype.copyWithin(target[, start[, end]])`
Copies a sequence of array elements within the array.

`Array.prototype.fill(value[, start[, end]])`

Fills all the elements of an array from a `start` index to an `end` index with a static `value`.

`Array.prototype.pop()`

Removes the last element from an array and returns that element.

`Array.prototype.push(element1[, ...[, elementN]])`

Adds one or more elements to the end of an array, and returns the new `length` of the array.

`Array.prototype.reverse()`

Reverses the order of the elements of an array *in place*. (First becomes the last, last becomes first.)

`Array.prototype.shift()`

Removes the first element from an array and returns that element.

`Array.prototype.sort()`

Sorts the elements of an array in place and returns the array.

`Array.prototype.splice(start[, deleteCount[, item1[, item2[, ...]]]])`

Adds and/or removes elements from an array.

`Array.prototype.unshift(element1[, ...[, elementN]])`

Adds one or more elements to the front of an array, and returns the new `length` of the array.

Accessor methods

These methods do not modify the array, and return a new array (based on some representation of the original).

`Array.prototype.concat([value1[, value2[, ...[, valueN]]]])`

Returns a new array that is this array joined with other array(s) and/or value(s).

`Array.prototype.filter(callbackFn(element[, index[, array]])[, thisArg])`

Returns a new array containing all elements of the calling array for which the provided filtering `callbackFn` returns `true`.

`Array.prototype.includes(valueToFind[, fromIndex])`

Determines whether the array contains `valueToFind`, returning `true` or `false` as appropriate.

`Array.prototype.indexOf(searchElement[, fromIndex])`

Returns the first (least) index of an element within the array equal to `searchElement`, or `-1` if none is found.

`Array.prototype.join([separator])`

Joins all elements of an array into a string.

`Array.prototype.lastIndexOf(searchElement[, fromIndex])`

Returns the last (greatest) index of an element within the array equal to `searchElement`, or `-1` if none is found.

`Array.prototype.slice([begin[, end]])`

Extracts a section of the calling array and returns a new array.

`Array.prototype.toSource()`

Returns an array literal representing the specified array. You can use this value to create a new array. Overrides the `Object.prototype.toSource()` method.

`Array.prototype.toString()`

Returns a string representing the array and its elements. Overrides the `Object.prototype.toString()` method.

`Array.prototype.toLocaleString()`

Returns a localized string representing the array and its elements. Overrides the `Object.prototype.toLocaleString()` method.

Iteration methods

Several methods accept callback functions which are executed 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 a value 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(callbackFn(element[, index[, array]])[, thisArg])`
: Returns `true` if every element in this array satisfies the testing *`callbackFn`*.

`Array.prototype.find(callbackFn(element[, index[, array]])[, thisArg])`
: Returns the found *`element`* in the array if some element in the array satisfies the testing *`callbackFn`*, or `undefined` if not found.

`Array.prototype.findIndex(callbackFn(element[, index[, array]])[, thisArg])`
: Returns the found index in the array, if an element in the array satisfies the testing *`callbackFn`*, or `-1` if not found.

`Array.prototype.forEach(callbackFn(currentValue[, index[, array]])[, thisArg])`
: Calls a *`callbackFn`* 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(callbackFn(currentValue[, index[, array]])[, thisArg])`
: Returns a new array containing the results of calling *`callbackFn`* on every element in this array.

`Array.prototype.reduce(callbackFn(accumulator, currentValue[, index[, array]])[, initialValue])`
: Apply a *`callbackFn`* against an *`accumulator`* and each value of the array (from left-to-right) as to reduce it to a single value.

`Array.prototype.reduceRight(callbackFn(accumulator, currentValue[, index[, array]])[, initialValue])`

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

`Array.prototype.some(callbackFn(element[, index[, array]])[, thisArg])`

Returns `true` if at least one element in this array satisfies the provided testing *`callbackFn`*.

`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.

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`.

```
let 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 `board[6][4]` to `board[4][4]`. The old position at `[6][4]` is made blank.

```
let 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')

// Move King's Pawn forward 2
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 (let 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	Initial publication
ECMAScript Latest Draft (ECMA-262) The definition of 'Array' in that specification.	ECMAScript 1

Browser compatibility

Update compatibility data on GitHub

	Chr	Edg	Fire	Inte	Op	Saf	And	Chr	Fire	Op	Saf	Sar	Noc
Array	1	12	1	4	4	1	≤37	18	4	10.1	1	1.0	Yes
Array() constructor	1	12	1	4	4	1	≤37	18	4	10.1	1	1.0	Yes
concat	1	12	1	5.5	4	1	1	18	4	10.1	1	1.0	Yes
copyWithin	45	12	32	No	32	9	45	45	32	32	9	5.0	4.0.0
entries	38	12	28	No	25	8	38	38	28	25	8	3.0	0.12
every	1	12	1.5	9	9.5	3	≤37	18	4	10.1	1	1.0	Yes
fill	45	12	31	No	32	8	45	45	31	32	8	5.0	4.0.0
filter	1	12	1.5	9	9.5	3	≤37	18	4	10.1	1	1.0	Yes
find	45	12	25	No	32	8	45	45	4	32	8	5.0	4.0.0
findIndex	45	12	25	No	32	8	45	45	4	32	8	5.0	4.0.0
flat	69	79	62	No	56	12	69	69	62	48	12	10.0	11.0.0
flatMap	69	79	62	No	56	12	69	69	62	48	12	10.0	11.0.0
forEach	1	12	1.5	9	9.5	3	≤37	18	4	10.1	1	1.0	Yes
from	45	12	32	No	32	9	45	45	32	32	9	5.0	4.0.0
includes	47	14	43	No	34	9	47	47	43	34	9	5.0	6.0.0

indexOf	1	12	1.5	9	9.5	3	≤37	18	4	10.1	1	1.0	Yes
isArray	5	12	4	9	10.5	5	1	18	4	14	5	1.0	Yes
join	1	12	1	5.5	4	1	1	18	4	10.1	1	1.0	Yes
keys	38	12	28	No	25	8	38	38	28	25	8	3.0	0.12
lastIndexOf	1	12	1.5	9	9.5	3	≤37	18	4	10.1	1	1.0	Yes
length	1	12	1	4	4	1	≤37	18	4	10.1	1	1.0	Yes
map	1	12	1.5	9	9.5	3	≤37	18	4	10.1	1	1.0	Yes
observe	36 — 52	No	No	No	No	No	No	No	No	No	No	No	No
of	45	12	25	No	26	9	39	39	25	26	9	4.0	4.0.0
pop	1	12	1	5.5	4	1	1	18	4	10.1	1	1.0	Yes
push	1	12	1	5.5	4	1	1	18	4	10.1	1	1.0	Yes
reduce	3	12	3	9	10.5	4.1	≤37	18	4	14	4	1.0	Yes
reduceRight	3	12	3	9	10.5	4.1	≤37	18	4	14	4	1.0	Yes
reverse	1	12	1	5.5	4	1	1	18	4	10.1	1	1.0	Yes
shift	1	12	1	5.5	4	1	1	18	4	10.1	1	1.0	Yes
slice	1	12	1	4	4	1	1	18	4	10.1	1	1.0	Yes
some	1	12	1.5	9	9.5	3	≤37	18	4	10.1	1	1.0	Yes
sort	1	12	1	5.5	4	1	1	18	4	10.1	1	1.0	Yes
splice	1	12	1	5.5	4	1	1	18	4	10.1	1	1.0	Yes
toLocaleString	1	12	1	5.5	4	1	≤37	18	4	10.1	1	1.0	Yes
toSource	No	No	1 — 74	No	No	No	No	No	4	No	No	No	No
toString	1	12	1	4	4	1	≤37	18	4	10.1	1	1.0	Yes
unobserve	36 — 52	No	No	No	No	No	No	No	No	No	No	No	No
unshift	1	12	1	5.5	4	1	1	18	4	10.1	1	1.0	Yes

`values`	66	12	60	No	53	9	66	66	60	47	9	9.0	10.9.0
`@@iterator`	38	12	36	No	25	10	38	38	36	25	10	3.0	0.12
`@@species`	51	79	48	No	38	10	51	51	48	41	10	5.0	6.5.0
`@@unscopables`	38	12	48	No	25	10	38	38	48	25	10	3.0	0.12

What are we missing?

Full support

No support

Non-standard. Expect poor cross-browser support.

Deprecated. Not for use in new websites.

See implementation notes.

User must explicitly enable this feature.

Uses a non-standard name.

See also

- From the JavaScript Guide:
  - “Indexing object properties”
  - “Indexed collections: `Array` object”

Typed Arrays

Last modified: Mar 9, 2020, by MDN contributors

Related Topics

Standard built-in objects

`Array`

Properties

- `Array.length`
- `Array.prototype[@@unscopables]`

Methods

- `Array.from()`
- `Array.isArray()`
- `Array.observe()`
- `Array.of()`
- `Array.prototype.concat()`
- `Array.prototype.copyWithin()`
- `Array.prototype.entries()`
- `Array.prototype.every()`
- `Array.prototype.fill()`
- `Array.prototype.filter()`
- `Array.prototype.find()`
- `Array.prototype.findIndex()`
- `Array.prototype.flat()`
- `Array.prototype.flatMap()`
- `Array.prototype.forEach()`
- `Array.prototype.includes()`
- `Array.prototype.indexOf()`
- `Array.prototype.join()`
- `Array.prototype.keys()`
- `Array.prototype.lastIndexOf()`

- Array.prototype.map()
- Array.prototype.pop()
- Array.prototype.push()
- Array.prototype.reduce()
- Array.prototype.reduceRight()
- Array.prototype.reverse()
- Array.prototype.shift()
- Array.prototype.slice()
- Array.prototype.some()
- Array.prototype.sort()
- Array.prototype.splice()
- Array.prototype.toLocaleString()
- Array.prototype.toSource()
- Array.prototype.toString()
- Array.prototype.unshift()
- Array.prototype.values()
- Array.prototype[@@iterator]()
- Array.unobserve()
- get Array[@@species]

Inheritance:

`Function`

Properties

- Function.arguments
- Function.arity
- Function.caller
- Function.displayName
- Function.length
- Function.name

Methods

- Function.prototype.apply()
- Function.prototype.bind()
- Function.prototype.call()
- Function.prototype.isGenerator()
- Function.prototype.toSource()

- Function.prototype.toString()

Object

Properties

- Object.prototype.__count__
- Object.prototype.__noSuchMethod__
- Object.prototype.__parent__
- Object.prototype.__proto__
- Object.prototype.constructor

Methods

- Object.prototype.__defineGetter__()
- Object.prototype.__defineSetter__()
- Object.prototype.__lookupGetter__()
- Object.prototype.__lookupSetter__()
- Object.prototype.hasOwnProperty()
- Object.prototype.isPrototypeOf()
- Object.prototype.propertyIsEnumerable()
- Object.prototype.toLocaleString()
- Object.prototype.toSource()
- Object.prototype.toString()
- Object.prototype.unwatch()
- Object.prototype.valueOf()
- Object.prototype.watch()
- Object.setPrototypeOf()