

MDN Plus now available in [your](#) country! Support MDN [and](#) make it your own. [Learn more](#) ✨

Array

The **Array** object, as with arrays in other programming languages, enables [storing a collection of multiple items under a single variable name](#), and has members for [performing common array operations](#).

Description

In JavaScript, arrays aren't [primitives](#) but are instead **Array** objects with the following core characteristics:

- **JavaScript arrays are resizable** and **can contain a mix of different [data types](#)**. (When those characteristics are undesirable, use [typed arrays](#) instead.)
- **JavaScript arrays are not associative arrays** and so, [array elements cannot be accessed using strings as indexes](#), but must be accessed using integers as indexes.
- **JavaScript arrays are [zero-indexed](#)** : the first element of an array is at index `0` , the second is at index `1` , and so on — and the last element is at the value of the array's [length](#) property minus `1` .
- **JavaScript [array-copy operations](#) create [shallow copies](#)**. (All standard built-in copy operations with *any* JavaScript objects create shallow copies, rather than [deep copies](#)).

Constructor

[Array\(\)](#)

Creates a new **Array** object.

Static properties

[get Array\[\[@@species\]\(#\)\]](#)

Returns the **Array** constructor.

Static methods

[Array.from\(\)](#)

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

[Array.isArray\(\)](#)

Returns `true` if the argument is an array, or `false` otherwise.

[Array.of\(\)](#)

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

Instance properties

[Array.prototype.length](#)

Reflects the number of elements in an array.

[Array.prototype\[@@unscopables\]](#)

Contains property names that were not included in the ECMAScript standard prior to the ES2015 version and that are ignored for [with](#) statement-binding purposes.

Instance methods

[Array.prototype.at\(\)](#)

Returns the array item at the given index. Accepts negative integers, which count back from the last item.

[Array.prototype.concat\(\)](#)

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

[Array.prototype.copyWithin\(\)](#)

Copies a sequence of array elements within an array.

[Array.prototype.entries\(\)](#)

Returns a new [array iterator](#) object that contains the key/value pairs for each index in an array.

[Array.prototype.every\(\)](#)

Returns `true` if every element in the calling array satisfies the testing function.

[Array.prototype.fill\(\)](#)

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

[Array.prototype.filter\(\)](#)

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

[Array.prototype.find\(\)](#)

Returns the found `element` in the calling array, if some element in the array satisfies the testing function, or `undefined` if not found.

[Array.prototype.findIndex\(\)](#)

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

[Array.prototype.flat\(\)](#)

Returns a new array with all sub-array elements concatenated into it recursively up to the specified depth.

[Array.prototype.flatMap\(\)](#)

Returns a new array formed by applying a given callback function to each element of the calling array, and then flattening the result by one level.

[Array.prototype.forEach\(\)](#)

Calls a function for each element in the calling array.

[Array.prototype.groupBy\(\)](#)

Groups the elements of an array into an object according to the strings returned by a test function.

[Array.prototype.groupByToMap\(\)](#)

Groups the elements of an array into a [Map](#) according to values returned by a test function.

[`Array.prototype.includes\(\)`](#)

Determines whether the calling array contains a value, returning `true` or `false` as appropriate.

[`Array.prototype.indexOf\(\)`](#)

Returns the first (least) index at which a given element can be found in the calling array.

[`Array.prototype.join\(\)`](#)

Joins all elements of an array into a string.

[`Array.prototype.keys\(\)`](#)

Returns a new [`array iterator`](#) that contains the keys for each index in the calling array.

[`Array.prototype.lastIndexOf\(\)`](#)

Returns the last (greatest) index at which a given element can be found in the calling array, or `-1` if none is found.

[`Array.prototype.map\(\)`](#)

Returns a new array containing the results of invoking a function on every element in the calling array.

[`Array.prototype.pop\(\)`](#)

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

[`Array.prototype.push\(\)`](#)

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

[`Array.prototype.reduce\(\)`](#)

Executes a user-supplied "reducer" callback function on each element of the array (from left to right), to reduce it to a single value.

[`Array.prototype.reduceRight\(\)`](#)

Executes a user-supplied "reducer" callback function on each element of the array (from right to left), to reduce it to a single value.

[`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.slice\(\)`](#)

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

[`Array.prototype.some\(\)`](#)

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

[`Array.prototype.sort\(\)`](#)

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

[`Array.prototype.splice\(\)`](#)

Adds and/or removes elements from an array.

[Array.prototype.toLocaleString\(\)](#)

Returns a localized string representing the calling array and its elements. Overrides the [Object.prototype.toLocaleString\(\)](#) method.

[Array.prototype.toString\(\)](#)

Returns a string representing the calling array and its elements. Overrides the [Object.prototype.toString\(\)](#) method.

[Array.prototype.unshift\(\)](#)

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

[Array.prototype.values\(\)](#)

Returns a new [array iterator](#) object that contains the values for each index in the array.

[Array.prototype\[@@iterator\]\(\)](#)

Returns the [values\(\)](#) function by default.

Examples

This section provides some examples of common array operations in JavaScript.

Note: If you're not yet familiar with array basics, consider first reading [JavaScript First Steps: Arrays](#), which [explains what arrays are](#), and includes other examples of common array operations.

Create an array

This example shows three ways to create new array: first using [array literal notation](#), then using the [Array\(\)](#) constructor, and finally using [String.prototype.split\(\)](#) to build the array from a string.

```
// 'fruits' array created using array literal notation.
const fruits = ['Apple', 'Banana'];
console.log(fruits.length);
// 2

// 'fruits' array created using the Array() constructor.
const fruits = new Array('Apple', 'Banana');
console.log(fruits.length);
// 2

// 'fruits' array created using String.prototype.split().
const fruits = 'Apple, Banana'.split(', ');
console.log(fruits.length);
// 2
```

Create a string from an array

This example uses the [join\(\)](#) method to create a string from the `fruits` array.

```
const fruits = ['Apple', 'Banana'];
const fruitsString = fruits.join(', ');
console.log(fruitsString);
// "Apple, Banana"
```

Access an array item by its index

This example shows how to access items in the `fruits` array by specifying the index number of their position in the array.

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

// The index of an array's first element is always 0.
fruits[0]; // Apple

// The index of an array's second element is always 1.
fruits[1]; // Banana

// The index of an array's last element is always one
// less than the length of the array.
fruits[fruits.length - 1]; // Banana

// Using a index number larger than the array's length
// returns 'undefined'.
fruits[99]; // undefined
```

Find the index of an item in an array

This example uses the [indexOf\(\)](#) method to find the position (index) of the string "Banana" in the `fruits` array.

```
const fruits = ['Apple', 'Banana'];
console.log(fruits.indexOf('Banana'));
// 1
```

Check if an array contains a certain item

This example shows two ways to check if the `fruits` array contains "Banana" and "Cherry": first with the [includes\(\)](#) method, and then with the [indexOf\(\)](#) method to test for an index value that's not `-1`.

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


fruits.includes('Banana'); // true
fruits.includes('Cherry'); // false

// If indexOf() doesn't return -1, the array contains the given item.
fruits.indexOf('Banana') !== -1; // true
fruits.indexOf('Cherry') !== -1; // false
```

Append an item to an array

This example uses the [push\(\)](#) method to append a new string to the `fruits` array.

```
const fruits = ['Apple', 'Banana'];
const newLength = fruits.push('Orange');
```

 [mdn web docs](#)

```
// 3
```

Remove the last item from an array

This example uses the [pop\(\)](#) method to remove the last item from the `fruits` array.

```
const fruits = ['Apple', 'Banana', 'Orange'];
const removedItem = fruits.pop();
console.log(fruits);
// ["Apple", "Banana"]
console.log(removedItem);
// Orange
```

Note: `pop()` can only be used to remove the last item from an array. To remove multiple items from the end of an array, see the next example.

Remove multiple items from the end of an array

This example uses the [splice\(\)](#) method to remove the last 3 items from the `fruits` array.

```
const fruits = ['Apple', 'Banana', 'Strawberry', 'Mango', 'Cherry'];
const start = -3;
const removedItems = fruits.splice(start);
console.log(fruits);
// ["Apple", "Banana"]
console.log(removedItems);
// ["Strawberry", "Mango", "Cherry"]
```

Truncate an array down to just its first N items

This example uses the [splice\(\)](#) method to truncate the `fruits` array down to just its first 2 items.

```
const fruits = ['Apple', 'Banana', 'Strawberry', 'Mango', 'Cherry'];
const start = 2;
const removedItems = fruits.splice(start);
console.log(fruits);
// ["Apple", "Banana"]
console.log(removedItems);
// ["Strawberry", "Mango", "Cherry"]
```

Remove the first item from an array

This example uses the [shift\(\)](#) method to remove the first item from the `fruits` array.

```
const fruits = ['Apple', 'Banana'];
const removedItem = fruits.shift();
console.log(fruits);
// ["Banana"]
console.log(removedItem);
// Apple
```

Note: `shift()` can only be used to remove the first item from an array. To remove multiple items from the beginning of an array, see the next example.

Remove multiple items from the beginning of an array

This example uses the [splice\(\)](#) method to remove the first 3 items from the `fruits` array.

```
const fruits = ['Apple', 'Strawberry', 'Cherry', 'Banana', 'Mango'];
const start = 0;
const deleteCount = 3;
const removedItems = fruits.splice(start, deleteCount);
console.log(fruits);
// ["Banana", "Mango"]
console.log(removedItems);
// ["Apple", "Strawberry", "Cherry"]
```

Add a new first item to an array

This example uses the [unshift\(\)](#) method to add, at index 0, a new item to the `fruits` array — making it the new first item in the array.

```
const fruits = ['Banana', 'Mango'];
const newLength = fruits.unshift('Strawberry');
console.log(fruits);
// ["Strawberry", "Banana", "Mango"]
console.log(newLength);
// 3
```

Remove a single item by index

This example uses the [splice\(\)](#) method to remove the string "Banana" from the `fruits` array — by specifying the index position of "Banana".

```
const fruits = ['Strawberry', 'Banana', 'Mango'];
const start = fruits.indexOf('Banana');
const deleteCount = 1;
const removedItems = fruits.splice(start, deleteCount);
console.log(fruits);
// ["Strawberry", "Mango"]
console.log(removedItems);
// ["Banana"]
```

Remove multiple items by index

This example uses the [splice\(\)](#) method to remove the strings "Banana" and "Strawberry" from the `fruits` array — by specifying the index position of "Banana", along with a count of the number of total items to remove.

```
const fruits = ['Apple', 'Banana', 'Strawberry', 'Mango'];
const start = 1;
const deleteCount = 2;
const removedItems = fruits.splice(start, deleteCount);
console.log(fruits);
// ["Apple", "Mango"]
console.log(removedItems);
// ["Banana", "Strawberry"]
```

Replace multiple items in an array

This example uses the [splice\(\)](#) method to replace the last 2 items in the `fruits` array with new items.

```
const fruits = ['Apple', 'Banana', 'Strawberry'];
const start = -2;
const deleteCount = 2;
```

```
const removedItems = fruits.splice(start, deleteCount, 'Mango', 'Cherry');
console.log(fruits);
// ["Apple", "Mango", "Cherry"]
console.log(removedItems);
// ["Banana", "Strawberry"]
```

Iterate over an array

This example uses a [for...of](#) loop to iterate over the `fruits` array, logging each item to the console.

```
const fruits = ['Apple', 'Mango', 'Cherry'];
for (const fruit of fruits) {
  console.log(fruit);
}
// Apple
// Mango
// Cherry
```

But `for...of` is just one of many ways to iterate over any array; for more ways, see [Loops and iteration](#), and see the documentation for the [every\(\)](#), [filter\(\)](#), [flatMap\(\)](#), [map\(\)](#), [reduce\(\)](#), and [reduceRight\(\)](#) methods — and see the next example, which uses the [forEach\(\)](#) method.

Call a function on each element in an array

This example uses the [forEach\(\)](#) method to call a function on each element in the `fruits` array; the function causes each item to be logged to the console, along with the item's index number.

```
const fruits = ['Apple', 'Mango', 'Cherry'];
fruits.forEach(function(item, index, array) {
  console.log(item, index);
});
// Apple 0
// Mango 1
// Cherry 2
```

Merge multiple arrays together

This example uses the [concat\(\)](#) method to merge the `fruits` array with a `moreFruits` array, to produce a new `combinedFruits` array. Notice that `fruits` and `moreFruits` remain unchanged.

```
const fruits = ['Apple', 'Banana', 'Strawberry'];
const moreFruits = ['Mango', 'Cherry'];
const combinedFruits = fruits.concat(moreFruits);
console.log(combinedFruits);
// ["Apple", "Banana", "Strawberry", "Mango", "Cherry"]

// The 'fruits' array remains unchanged.
console.log(fruits);
// ["Apple", "Banana", "Strawberry"]

// The 'moreFruits' array also remains unchanged.
console.log(moreFruits);
// ["Mango", "Cherry"]
```

Copy an array

This example shows three ways to create a new array from the existing `fruits` array: first by using [spread syntax](#), then by using the [from\(\)](#) method, and then by using the [slice\(\)](#) method.

```
const fruits = ['Strawberry', 'Mango'];

// Create a copy using spread syntax.
const fruitsCopy = [...fruits];
// ["Strawberry", "Mango"]

// Create a copy using the from() method.
const fruitsCopy = Array.from(fruits);
// ["Strawberry", "Mango"]

// Create a copy using the slice() method.
const fruitsCopy = fruits.slice();
// ["Strawberry", "Mango"]
```

All built-in array-copy operations ([spread syntax](#), [Array.from\(\)](#), [Array.prototype.slice\(\)](#), and [Array.prototype.concat\(\)](#)) create [shallow copies](#). If you instead want a [deep copy](#) of an array, you can use [JSON.stringify\(\)](#) to convert the array to a JSON string, and then [JSON.parse\(\)](#) to convert the string back into a new array that's completely independent from the original array.

```
const fruitsDeepCopy = JSON.parse(JSON.stringify(fruits));
```

You can also create deep copies using the [structuredClone\(\)](#) method, which has the advantage of allowing [transferable objects](#) in the source to be *transferred* to the new copy, rather than just cloned.

Finally, it's important to understand that assigning an existing array to a new variable doesn't create a copy of either the array or its elements. Instead the new variable is just a reference, or alias, to the original array; that is, the original array's name and the new variable name are just two names for the exact same object (and so will always evaluate as [strictly equivalent](#)). Therefore, if you make any changes at all either to the value of the original array or to the value of the new variable, the other will change, too:

```
const fruits = ['Strawberry', 'Mango'];
const fruitsAlias = fruits;
// 'fruits' and 'fruitsAlias' are the same object, strictly equivalent.
fruits === fruitsAlias // true
// Any changes to the 'fruits' array change 'fruitsAlias' too.
fruits.unshift('Apple', 'Banana');
console.log(fruits);
// ['Apple', 'Banana', 'Strawberry', 'Mango']
console.log(fruitsAlias);
// ['Apple', 'Banana', 'Strawberry', 'Mango']
```

Grouping the elements of an array

The [Array.prototype.groupBy\(\)](#) methods can be used to group the elements of an array, using a test function that returns a string indicating the group of the current element.

Here we have a simple inventory array that contains "food" objects that have a `name` and a `type`.

```
const inventory = [
  { name: 'asparagus', type: 'vegetables' },
  { name: 'bananas', type: 'fruit' },
  { name: 'goat', type: 'meat' },
  { name: 'cherries', type: 'fruit' },
```

```
{ name: 'fish', type: 'meat' }
];
```

To use `groupBy()`, you supply a callback function that is called with the current element, and optionally the current index and array, and returns a string indicating the group of the element.

The code below uses a arrow function to return the `type` of each array element (this uses [object destructuring syntax for function arguments](#) to unpack the `type` element from the passed object). The result is an object that has properties named after the unique strings returned by the callback. Each property is assigned an array containing the elements in the group.

```
let result = inventory.groupBy( ({ type }) => type );
console.log(result.vegetables)
// expected output: Array [Object { name: "asparagus", type: "vegetables" }]
```

Note that the returned object references the *same* elements as the original array (not [deep copies](#)). Changing the internal structure of these elements will be reflected in both the original array and the returned object.

If you can't use a string as the key, for example, if the information to group is associated with an object that might change, then you can instead use [Array.prototype.groupByToMap\(\)](#). This is very similar to `groupBy` except that it groups the elements of the array into a [Map](#) that can use an arbitrary value ([object](#) or [primitive](#)) as a key.

Other examples

Creating a two-dimensional array

The following creates a chessboard 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.

```
const 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

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

Results in

```
// The first column is the index
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
```

Notes

Array objects 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.

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.

```
const 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
```

In the `3d` example, `'3d'` *had* to be quoted (because it begins with a digit). But it's also possible to quote the 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 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.

```
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(fruits); // ['banana', 'apple', 'peach', empty x 2, 'mango', empty x 4]
console.log(Object.keys(fruits)); // ['0', '1', '2', '5']
console.log(fruits.length); // 10
console.log(fruits[8]); // undefined
```

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 that has properties and elements which provide information about the match. Such an array is returned by [RegExp.exec\(\)](#) and [String.match\(\)](#).

To help explain these properties and elements, see the following 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
<code>input</code> <small>Read only</small>	The original string against which the regular expression was matched.	"cdbBdbsbz"
<code>index</code> <small>Read only</small>	The zero-based index of the match in the string.	1
<code>[0]</code> <small>Read only</small>	The last matched characters.	"dbBd"
<code>[1], ...[n]</code> <small>Read only</small>	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"

Specifications

Specification
ECMAScript Language Specification # sec-array-objects

Browser compatibility

[Report problems with this compatibility data on GitHub](#)

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android
Array	Chrome 1	Edge 12	Firefox 1	Internet Explorer 4	Opera 4	Safari 1	WebView 3 Android
Array().constructor	Chrome 1	Edge 12	Firefox 1	Internet Explorer 4	Opera 4	Safari 1	WebView 3 Android

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android
at	Chrome 92	Edge 92	Firefox 90	Internet Explorer No	Opera 78	Safari 15.4	WebView 9 Android
concat	Chrome 1	Edge 12	Firefox 1	Internet Explorer 5.5	Opera 4	Safari 1	WebView Android
copyWithin	Chrome 45	Edge 12	Firefox 32	Internet Explorer No	Opera 32	Safari 9	WebView 4 Android
entries	Chrome 38	Edge 12	Firefox 28	Internet Explorer No	Opera 25	Safari 8	WebView 3 Android
every	Chrome 1	Edge 12	Firefox 1.5	Internet Explorer 9	Opera 9.5	Safari 3	WebView 3 Android
fill	Chrome 45	Edge 12	Firefox 31	Internet Explorer No	Opera 32	Safari 8	WebView 4 Android
filter	Chrome 1	Edge 12	Firefox 1.5	Internet Explorer 9	Opera 9.5	Safari 3	WebView 3 Android
find	Chrome 45	Edge 12	Firefox 25	Internet Explorer No	Opera 32	Safari 8	WebView 4 Android
findIndex	Chrome 45	Edge 12	Firefox 25	Internet Explorer No	Opera 32	Safari 8	WebView 4 Android
findLast	Chrome 97	Edge 97	Firefox No	Internet Explorer No	Opera 83	Safari 15.4	WebView 9 Android
findLastIndex	Chrome 97	Edge 97	Firefox No	Internet Explorer No	Opera 83	Safari 15.4	WebView 9 Android
flat	Chrome 69	Edge 79	Firefox 62	Internet Explorer No	Opera 56	Safari 12	WebView 6 Android
flatMap	Chrome 69	Edge 79	Firefox 62	Internet Explorer No	Opera 56	Safari 12	WebView 6 Android
forEach	Chrome 1	Edge 12	Firefox 1.5	Internet Explorer 9	Opera 9.5	Safari 3	WebView 3 Android
from	Chrome 45	Edge 12	Firefox 32	Internet Explorer No	Opera 32	Safari 9	WebView 4 Android

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android
groupBy	Chrome No	Edge No	Firefox Nightly	Internet Explorer No	Opera No	Safari No	WebView No Android
groupByToMap	Chrome No	Edge No	Firefox Nightly	Internet Explorer No	Opera No	Safari No	WebView No Android
includes	Chrome 47	Edge 14	Firefox 43	Internet Explorer No	Opera 34	Safari 9	WebView 4 Android
indexOf	Chrome 1	Edge 12	Firefox 1.5	Internet Explorer 9	Opera 9.5	Safari 3	WebView 3 Android
isArray	Chrome 5	Edge 12	Firefox 4	Internet Explorer 9	Opera 10.5	Safari 5	WebView Android
join	Chrome 1	Edge 12	Firefox 1	Internet Explorer 5.5	Opera 4	Safari 1	WebView Android
keys	Chrome 38	Edge 12	Firefox 28	Internet Explorer No	Opera 25	Safari 8	WebView 3 Android
lastIndexOf	Chrome 1	Edge 12	Firefox 1.5	Internet Explorer 9	Opera 9.5	Safari 3	WebView 3 Android
length	Chrome 1	Edge 12	Firefox 1	Internet Explorer 4	Opera 4	Safari 1	WebView 3 Android
map	Chrome 1	Edge 12	Firefox 1.5	Internet Explorer 9	Opera 9.5	Safari 3	WebView 3 Android
of	Chrome 45	Edge 12	Firefox 25	Internet Explorer No	Opera 26	Safari 9	WebView 3 Android
pop	Chrome 1	Edge 12	Firefox 1	Internet Explorer 5.5	Opera 4	Safari 1	WebView Android
push	Chrome 1	Edge 12	Firefox 1	Internet Explorer 5.5	Opera 4	Safari 1	WebView Android
reduce	Chrome 3	Edge 12	Firefox 3	Internet Explorer 9	Opera 10.5	Safari 5	WebView 3 Android
reduceRight	Chrome 3	Edge 12	Firefox 3	Internet Explorer 9	Opera 10.5	Safari 5	WebView 3 Android

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android
reverse	Chrome 1	Edge 12	Firefox 1	Internet Explorer 5.5	Opera 4	Safari 1	WebView Android
shift	Chrome 1	Edge 12	Firefox 1	Internet Explorer 5.5	Opera 4	Safari 1	WebView Android
slice	Chrome 1	Edge 12	Firefox 1	Internet Explorer 4	Opera 4	Safari 1	WebView Android
some	Chrome 1	Edge 12	Firefox 1.5	Internet Explorer 9	Opera 9.5	Safari 3	WebView 3 Android
sort	Chrome 1	Edge 12	Firefox 1	Internet Explorer 5.5	Opera 4	Safari 1	WebView Android
Stable sorting	Chrome 70	Edge 79	Firefox 3	Internet Explorer No	Opera 57	Safari 10.1	WebView 7 Android
splice	Chrome 1	Edge 12	Firefox 1	Internet Explorer 5.5	Opera 4	Safari 1	WebView Android
toLocaleString	Chrome 1	Edge 12	Firefox 1	Internet Explorer 5.5	Opera 4	Safari 1	WebView 3 Android
locales parameter	Chrome 24	Edge 79	Firefox 52	Internet Explorer No	Opera 15	Safari 7	WebView 3 Android
options parameter	Chrome 24	Edge 79	Firefox 52	Internet Explorer No	Opera 15	Safari 7	WebView 3 Android
toSource	Chrome No	Edge No	Firefox 1–74	Internet Explorer No	Opera No	Safari No	WebView No Android
toString	Chrome 1	Edge 12	Firefox 1	Internet Explorer 4	Opera 4	Safari 1	WebView 3 Android
unshift	Chrome 1	Edge 12	Firefox 1	Internet Explorer 5.5	Opera 4	Safari 1	WebView Android
values	Chrome 66	Edge 12	Firefox 60	Internet Explorer No	Opera 53	Safari 9	WebView 6 Android
@@iterator	Chrome 38	Edge 12	Firefox 36	Internet Explorer No	Opera 25	Safari 10	WebView 3 Android

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android
@@species	Chrome 51	Edge 79	Firefox 48	Internet Explorer No	Opera 38	Safari 10	WebView 5 Android
@@unscopables	Chrome 38	Edge 12	Firefox 48	Internet Explorer No	Opera 25	Safari 10	WebView 3 Android

Full support

Partial support

In development. Supported in a pre-release version.

No support

Experimental. Expect behavior to change in the future.

Non-standard. Check cross-browser support before using.

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](#)
- [RangeError: invalid array length](#)

Last modified: Apr 15, 2022, [by MDN contributors](#)