mdn web docs_

Array.prototype.sort()

The **`sort()`** method sorts the elements of an array *in place* and returns the reference to the same array, now sorted. The default sort order is ascending, built upon converting the elements into strings, then comparing their sequences of UTF-16 code units values.

The time and space complexity of the sort cannot be guaranteed as it depends on the implementation.

Try it

JavaScript Demo: Array.sort()

```
const months = ['March', 'Jan', 'Feb', 'Dec'];
months.sort();
console.log(months);
// expected output: Array ["Dec", "Feb", "Jan", "

const array1 = [1, 30, 4, 21, 100000];
array1.sort();
console.log(array1);
// expected output: Array [1, 100000, 21, 30, 4]

```

Run ›   Reset

Syntax

```
// Functionless
sort()

// Arrow function
sort((a, b) => { /* … */ } )

// Compare function
sort(compareFn)

// Inline compare function
sort(function compareFn(a, b) { /* … */ })
```

Parameters

`compareFn` Optional

Specifies a function that defines the sort order. If omitted, the array elements are converted to strings, then sorted according to each character's Unicode code point value.

`a`

The first element for comparison.

`b`

The second element for comparison.

Return value

The reference to the original array, now sorted. Note that the array is sorted *in place* , and no copy is made.

Description

If `compareFunction` is not supplied, all non- `undefined` array elements are sorted by converting them to strings and comparing strings in UTF-16 code units order. For example, "banana" comes before "cherry". In a numeric sort, 9 comes before 80, but because numbers are converted to strings, "80" comes before "9" in the Unicode order. All `undefined` elements are sorted to the end of the array.

> **Note:** In UTF-16, Unicode characters above `\uFFFF` are encoded as two surrogate code units, of the range `\uD800` - `\uDFFF` . The value of each code unit is taken separately into account for the comparison. Thus the character formed by the surrogate pair `\uD855\uDE51` will be sorted before the character `\uFF3A` .

If `compareFunction` is supplied, all non- `undefined` array elements are sorted according to the return value of the compare function (all `undefined` elements are sorted to the end of the array, with no call to `compareFunction`).

`compareFunction(a, b)` return value	sort order
> 0	sort `a` after `b`
< 0	sort `a` before `b`
=== 0	keep original order of `a` and `b`

So, the compare function has the following form:

```
function compare(a, b) {
  if (a is less than b by some ordering criterion) {
    return -1;
  }
  if (a is greater than b by the ordering criterion) {
    return 1;
  }
  // a must be equal to b
  return 0;
}
```

More formally, the comparator is expected to have the following properties, in order to ensure proper sort behavior:

- *Pure*: The comparator does not mutate the objects being compared or any external state. (This is important because there's no guarantee *when* and *how* the comparator will be called, so any particular call should not produce visible effects to the outside.)
- *Stable*: The comparator returns the same result with the same pair of input.
- *Reflexive*: `compare(a, a) === 0`.
- *Symmetric*: `compare(a, b)` and `compare(b, a)` must both be `0` or have opposite signs.
- *Transitive*: If `compare(a, b)` and `compare(b, c)` are both positive, zero, or negative, then `compare(a, c)` has the same positivity as the previous two.

A comparator conforming to the constraints above will always be able to return all of `1`, `0`, and `-1`, or consistently return `0`. For example, if a comparator only returns `1` and `0`, or only returns `0` and `-1`, it will not be able to sort reliably because *symmetry* is broken. A comparator that always returns `0` will cause the array to not be changed at all, but is reliable nonetheless.

The default lexicographic comparator satisfies all constraints above.

To compare numbers instead of strings, the compare function can subtract `b` from `a`. The following function will sort the array in ascending order (if it doesn't contain `Infinity` and `NaN`):

```
function compareNumbers(a, b) {
  return a - b;
}
```

The `sort` method can be conveniently used with function expressions or arrow functions.

```
const numbers = [4, 2, 5, 1, 3];
numbers.sort(function (a, b) {
  return a - b;
});
console.log(numbers);
// [1, 2, 3, 4, 5]

// OR

const numbers2 = [4, 2, 5, 1, 3];
numbers2.sort((a, b) => a - b);
console.log(numbers2);
// [1, 2, 3, 4, 5]
```

Arrays of objects can be sorted by comparing the value of one of their properties.

```
const items = [
  { name: 'Edward', value: 21 },
  { name: 'Sharpe', value: 37 },
  { name: 'And', value: 45 },
  { name: 'The', value: -12 },
  { name: 'Magnetic', value: 13 },
  { name: 'Zeros', value: 37 }
];

// sort by value
```

```
items.sort((a, b) => a.value - b.value);

// sort by name
items.sort((a, b) => {
  const nameA = a.name.toUpperCase(); // ignore upper and lowercase
  const nameB = b.name.toUpperCase(); // ignore upper and lowercase
  if (nameA < nameB) {
    return -1;
  }
  if (nameA > nameB) {
    return 1;
  }

  // names must be equal
  return 0;
});
```

Examples

Creating, displaying, and sorting an array

The following example creates four arrays and displays the original array, then the sorted arrays. The numeric arrays are sorted without a compare function, then sorted using one.

```
const stringArray = ['Blue', 'Humpback', 'Beluga'];
const numberArray = [40, 1, 5, 200];
const numericStringArray = ['80', '9', '700'];
const mixedNumericArray = ['80', '9', '700', 40, 1, 5, 200];

function compareNumbers(a, b) {
  return a - b;
}

stringArray.join(); // 'Blue,Humpback,Beluga'
stringArray.sort(); // ['Beluga', 'Blue', 'Humpback']

numberArray.join(); // '40,1,5,200'
numberArray.sort(); // [1, 200, 40, 5]
numberArray.sort(compareNumbers); // [1, 5, 40, 200]

numericStringArray.join(); // '80,9,700'
numericStringArray.sort(); // ['700', '80', '9']
numericStringArray.sort(compareNumbers); // ['9', '80', '700']

mixedNumericArray.join(); // '80,9,700,40,1,5,200'
mixedNumericArray.sort(); // [1, 200, 40, 5, '700', '80', '9']
mixedNumericArray.sort(compareNumbers); // [1, 5, '9', 40, '80', 200, '700']
```

Sorting non-ASCII characters

For sorting strings with non-ASCII characters, i.e. strings with accented characters (e, é, è, a, ä, etc.), strings from languages other than English, use `String.localeCompare` . This function can compare those characters so they appear in the right order.

```
const items = ['réservé', 'premier', 'communiqué', 'café', 'adieu', 'éclair'];
items.sort((a, b) => a.localeCompare(b));

// items is ['adieu', 'café', 'communiqué', 'éclair', 'premier', 'réservé']
```

Sorting with map

The `compareFunction` can be invoked multiple times per element within the array. Depending on the `compareFunction`'s nature, this may yield a high overhead. The more work a `compareFunction` does and the more elements there are to sort, it may be more efficient to use `map()` for sorting. The idea is to traverse the array once to extract the actual values used for sorting into a temporary array, sort the temporary array, and then traverse the temporary array to achieve the right order.

```
// the array to be sorted
const data = ['delta', 'alpha', 'charlie', 'bravo'];

// temporary array holds objects with position and sort-value
const mapped = data.map((v, i) => {
  return { i, value: someSlowOperation(v) };
})

// sorting the mapped array containing the reduced values
mapped.sort((a, b) => {
  if (a.value > b.value) {
    return 1;
  }
  if (a.value < b.value) {
    return -1;
  }
  return 0;
});

const result = mapped.map((v) => data[v.i]);
```

There is an open source library available called mapsort which applies this approach.

sort() returns the reference to the same array

The `sort()` method returns a reference to the original array, so mutating the returned array will mutate the original array as well.

```
const numbers = [3, 1, 4, 1, 5];
const sorted = numbers.sort((a, b) => a - b);
// numbers and sorted are both [1, 1, 3, 4, 5]
sorted[0] = 10;
console.log(numbers[0]); // 10
```

In case you want `sort()` to not mutate the original array, but return a shallow-copied array like other array methods (e.g. `map()`) do, you can do a shallow copy before calling `sort()`, using the spread syntax or `Array.from()`.

```
const numbers = [3, 1, 4, 1, 5];
// [...numbers] creates a shallow copy, so sort() does not mutate the original
const sorted = [...numbers].sort((a, b) => a - b);
```

```
sorted[0] = 10;
console.log(numbers[0]); // 3
```

Sort stability

Since version 10 (or EcmaScript 2019), the specification dictates that `Array.prototype.sort` is stable.

For example, say you had a list of students alongside their grades. Note that the list of students is already pre-sorted by name in alphabetical order:

```
const students = [
  { name: "Alex",   grade: 15 },
  { name: "Devlin", grade: 15 },
  { name: "Eagle",  grade: 13 },
  { name: "Sam",    grade: 14 }
];
```

After sorting this array by `grade` in ascending order:

```
students.sort((firstItem, secondItem) => firstItem.grade - secondItem.grade);
```

The `students` variable will then have the following value:

```
[
  { name: "Eagle",  grade: 13 },
  { name: "Sam",    grade: 14 },
  { name: "Alex",   grade: 15 }, // original maintained for similar grade (stable sorting)
  { name: "Devlin", grade: 15 }  // original maintained for similar grade (stable sorting)
];
```

It's important to note that students that have the same grade (for example, Alex and Devlin), will remain in the same order as before calling the sort. This is what a stable sorting algorithm guarantees.

Before version 10 (or EcmaScript 2019), sort stability was not guaranteed, meaning that you could end up with the following:

```
[
  { name: "Eagle",  grade: 13 },
  { name: "Sam",    grade: 14 },
  { name: "Devlin", grade: 15 }, // original order not maintained
  { name: "Alex",   grade: 15 }  // original order not maintained
];
```

Sorting with non-well-formed comparator

If a comparing function does not satisfy all of purity, stability, reflexivity, symmetry, and transitivity rules, as explained in the description, the program's behavior is not well-defined.

For example, consider this code:

```
const arr = [3, 1, 4, 1, 5, 9];
const compare = (a, b) => a > b ? 1 : 0;
arr.sort(compare);
```

The `compare` function here is not well-formed, because it does not satisfy symmetry: if `a > b`, it returns `1`; but by swapping `a` and `b`, it returns `0` instead of a negative value. Therefore, the resulting array will be different across engines. For example, V8 (used by Chrome, Node.js, etc.) and JavaScriptCore (used by Safari) would not sort the array at all and return `[3, 1, 4, 1, 5, 9]`, while SpiderMonkey (used by Firefox) will return the array sorted ascendingly, as `[1, 1, 3, 4, 5, 9]`.

However, if the `compare` function is changed slightly so that it returns `-1` or `0`:

```
const arr = [3, 1, 4, 1, 5, 9];
const compare = (a, b) => a > b ? -1 : 0;
arr.sort(compare);
```

Then V8 and JavaScriptCore sorts it descendingly, as `[9, 5, 4, 3, 1, 1]`, while SpiderMonkey returns it as-is: `[3, 1, 4, 1, 5, 9]`.

Due to this implementation inconsistency, you are always advised to make your comparator well-formed by following the five constraints.

Specifications

Specification
[ECMAScript Language Specification](#) [# sec-array.prototype.sort](#)

Browser compatibility

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

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Chrome Android	Firefox for Android	Opera Android
`sort`	Chrome 1	Edge 12	Firefox 1	Internet Explorer 5.5	Opera 4	Safari 1	Chrome Android 18	Firefox for Android 4	Opera Androi
Stable sorting	Chrome 70	Edge 79	Firefox 3	Internet Explorer No	Opera 57	Safari 10.1	Chrome Android 70	Firefox for Android 4	Opera Androi

Full support No support

See also

- [Polyfill of `Array.prototype.sort` with modern behavior like stable sort in `core-js`](#)
- [`Array.prototype.reverse()`](#)

- `String.prototype.localeCompare()`
- About the stability of the algorithm used by V8 engine
- V8 sort stability
- Mathias Bynens' sort stability demo

Last modified: Aug 4, 2022, by MDN contributors