



Buy EPUB/PDF

[Home](#) → [The JavaScript language](#) → [Data types](#)

December 31, 2023

Array methods

Arrays provide a lot of methods. To make things easier, in this chapter, they are split into groups.

Add/remove items

We already know methods that add and remove items from the beginning or the end:

- `arr.push(...items)` – adds items to the end,
- `arr.pop()` – extracts an item from the end,
- `arr.shift()` – extracts an item from the beginning,
- `arr.unshift(...items)` – adds items to the beginning.

Here are a few others.

splice

How to delete an element from the array?

The arrays are objects, so we can try to use `delete`:

```
1 let arr = ["I", "go", "home"];
2
3 delete arr[1]; // remove "go"
4
5 alert( arr[1] ); // undefined
6
7 // now arr = ["I", , "home"];
8 alert( arr.length ); // 3
```



The element was removed, but the array still has 3 elements, we can see that `arr.length == 3`.

That's natural, because `delete obj.key` removes a value by the `key`. It's all it does. Fine for objects. But for arrays we usually want the rest of the elements to shift and occupy the freed place. We expect to have a shorter array now.

So, special methods should be used.

The `arr.splice` method is a Swiss army knife for arrays. It can do everything: insert, remove and replace elements.

The syntax is:

```
1 arr.splice(start[, deleteCount, elem1, ..., elemN])
```

It modifies `arr` starting from the index `start`: removes `deleteCount` elements and then inserts `elem1`, ..., `elemN` at their place. Returns the array of removed elements.

This method is easy to grasp by examples.

Let's start with the deletion:

```
1 let arr = ["I", "study", "JavaScript"];
2
3 arr.splice(1, 1); // from index 1 remove 1 element
4
5 alert( arr ); // ["I", "JavaScript"]
```

Easy, right? Starting from the index `1` it removed `1` element.

In the next example, we remove 3 elements and replace them with the other two:

```
1 let arr = ["I", "study", "JavaScript", "right", "now"];
2
3 // remove 3 first elements and replace them with another
4 arr.splice(0, 3, "Let's", "dance");
5
6 alert( arr ) // now ["Let's", "dance", "right", "now"]
```

Here we can see that `splice` returns the array of removed elements:

```
1 let arr = ["I", "study", "JavaScript", "right", "now"];
2
3 // remove 2 first elements
4 let removed = arr.splice(0, 2);
5
6 alert( removed ); // "I", "study" <-- array of removed elements
```

The `splice` method is also able to insert the elements without any removals. For that, we need to set `deleteCount` to `0`:

```
1 let arr = ["I", "study", "JavaScript"];
2
3 // from index 2
4 // delete 0
5 // then insert "complex" and "language"
6 arr.splice(2, 0, "complex", "language");
7
```

8

```
alert( arr ); // "I", "study", "complex", "language", "JavaScript"
```

Negative indexes allowed

Here and in other array methods, negative indexes are allowed. They specify the position from the end of the array, like here:

```
1 let arr = [1, 2, 5];
2
3 // from index -1 (one step from the end)
4 // delete 0 elements,
5 // then insert 3 and 4
6 arr.splice(-1, 0, 3, 4);
7
8 alert( arr ); // 1,2,3,4,5
```

slice

The method `arr.slice` is much simpler than the similar-looking `arr.splice`.

The syntax is:

```
1 arr.slice([start], [end])
```

It returns a new array copying to it all items from index `start` to `end` (not including `end`). Both `start` and `end` can be negative, in that case position from array end is assumed.

It's similar to a string method `str.slice`, but instead of substrings, it makes subarrays.

For instance:

```
1 let arr = ["t", "e", "s", "t"];
2
3 alert( arr.slice(1, 3) ); // e,s (copy from 1 to 3)
4
5 alert( arr.slice(-2) ); // s,t (copy from -2 till the end)
```

We can also call it without arguments: `arr.slice()` creates a copy of `arr`. That's often used to obtain a copy for further transformations that should not affect the original array.

concat

The method `arr.concat` creates a new array that includes values from other arrays and additional items.

The syntax is:

```
1 arr.concat(arg1, arg2...)
```

It accepts any number of arguments – either arrays or values.

The result is a new array containing items from `arr`, then `arg1`, `arg2` etc.

If an argument `argN` is an array, then all its elements are copied. Otherwise, the argument itself is copied.

For instance:

```
1 let arr = [1, 2];
2
3 // create an array from: arr and [3,4]
4 alert( arr.concat([3, 4]) ); // 1,2,3,4
5
6 // create an array from: arr and [3,4] and [5,6]
7 alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6
8
9 // create an array from: arr and [3,4], then add values 5 and 6
10 alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

Normally, it only copies elements from arrays. Other objects, even if they look like arrays, are added as a whole:

```
1 let arr = [1, 2];
2
3 let arrayLike = {
4   0: "something",
5   length: 1
6 };
7
8 alert( arr.concat(arrayLike) ); // 1,2,[object Object]
```

...But if an array-like object has a special `Symbol.isConcatSpreadable` property, then it's treated as an array by `concat`: its elements are added instead:

```
1 let arr = [1, 2];
2
3 let arrayLike = {
4   0: "something",
5   1: "else",
6   [Symbol.isConcatSpreadable]: true,
7   length: 2
8 };
9
10
```

```
alert( arr.concat(arrayLike) ); // 1,2,something,else
```

Iterate: forEach

The `arr.forEach` method allows to run a function for every element of the array.

The syntax:

```
1 arr.forEach(function(item, index, array) {  
2   // ... do something with an item  
3 });
```

For instance, this shows each element of the array:

```
1 // for each element call alert  
2 ["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

And this code is more elaborate about their positions in the target array:

```
1 ["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {  
2   alert(`${item} is at index ${index} in ${array}`);  
3 });
```

The result of the function (if it returns any) is thrown away and ignored.

Searching in array

Now let's cover methods that search in an array.

indexOf/lastIndexOf and includes

The methods `arr.indexOf` and `arr.includes` have the similar syntax and do essentially the same as their string counterparts, but operate on items instead of characters:

- `arr.indexOf(item, from)` – looks for `item` starting from index `from`, and returns the index where it was found, otherwise `-1`.
- `arr.includes(item, from)` – looks for `item` starting from index `from`, returns `true` if found.

Usually, these methods are used with only one argument: the `item` to search. By default, the search is from the beginning.

For instance:

```
1 let arr = [1, 0, false];
2
3 alert( arr.indexOf(0) ); // 1
4 alert( arr.indexOf(false) ); // 2
5 alert( arr.indexOf(null) ); // -1
6
7 alert( arr.includes(1) ); // true
```

Please note that `indexOf` uses the strict equality `===` for comparison. So, if we look for `false`, it finds exactly `false` and not the zero.

If we want to check if `item` exists in the array and don't need the index, then `arr.includes` is preferred.

The method `arr.lastIndexOf` is the same as `indexOf`, but looks for from right to left.

```
1 let fruits = ['Apple', 'Orange', 'Apple']
2
3 alert( fruits.indexOf('Apple') ); // 0 (first Apple)
4 alert( fruits.lastIndexOf('Apple') ); // 2 (last Apple)
```

The `includes` method handles `NaN` correctly

A minor, but noteworthy feature of `includes` is that it correctly handles `NaN`, unlike `indexOf`:

```
1 const arr = [NaN];
2 alert( arr.indexOf(NaN) ); // -1 (wrong, should be 0)
3 alert( arr.includes(NaN) ); // true (correct)
```

That's because `includes` was added to JavaScript much later and uses the more up-to-date comparison algorithm internally.

find and findIndex/findLastIndex

Imagine we have an array of objects. How do we find an object with a specific condition?

Here the `arr.find(fn)` method comes in handy.

The syntax is:

```
1 let result = arr.find(function(item, index, array) {
2   // if true is returned, item is returned and iteration is stopped
3   // for falsy scenario returns undefined
4 });
```

The function is called for elements of the array, one after another:

- `item` is the element.
- `index` is its index.
- `array` is the array itself.

If it returns `true`, the search is stopped, the `item` is returned. If nothing is found, `undefined` is returned.

For example, we have an array of users, each with the fields `id` and `name`. Let's find the one with `id == 1`:

```
1 let users = [  
2   {id: 1, name: "John"},  
3   {id: 2, name: "Pete"},  
4   {id: 3, name: "Mary"}  
5 ];  
6  
7 let user = users.find(item => item.id == 1);  
8  
9 alert(user.name); // John
```

In real life, arrays of objects are a common thing, so the `find` method is very useful.

Note that in the example we provide to `find` the function `item => item.id == 1` with one argument. That's typical, other arguments of this function are rarely used.

The `arr.findIndex` method has the same syntax but returns the index where the element was found instead of the element itself. The value of `-1` is returned if nothing is found.

The `arr.findLastIndex` method is like `findIndex`, but searches from right to left, similar to `lastIndexOf`.

Here's an example:

```
1 let users = [  
2   {id: 1, name: "John"},  
3   {id: 2, name: "Pete"},  
4   {id: 3, name: "Mary"},  
5   {id: 4, name: "John"}  
6 ];  
7  
8 // Find the index of the first John  
9 alert(users.findIndex(user => user.name == 'John')); // 0  
10  
11 // Find the index of the last John  
12 alert(users.findLastIndex(user => user.name == 'John')); // 3
```

filter

The `find` method looks for a single (first) element that makes the function return `true`.

If there may be many, we can use `arr.filter(fn)`.

The syntax is similar to `find`, but `filter` returns an array of all matching elements:

```
1 let results = arr.filter(function(item, index, array) {
2   // if true item is pushed to results and the iteration continues
3   // returns empty array if nothing found
4 });
```

For instance:

```
1 let users = [
2   {id: 1, name: "John"},
3   {id: 2, name: "Pete"},
4   {id: 3, name: "Mary"}
5 ];
6
7 // returns array of the first two users
8 let someUsers = users.filter(item => item.id < 3);
9
10 alert(someUsers.length); // 2
```

Transform an array

Let's move on to methods that transform and reorder an array.

map

The `arr.map` method is one of the most useful and often used.

It calls the function for each element of the array and returns the array of results.

The syntax is:

```
1 let result = arr.map(function(item, index, array) {
2   // returns the new value instead of item
3 });
```

For instance, here we transform each element into its length:

```
1 let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
2 alert(lengths); // 5,7,6
```

sort(fn)

The call to `arr.sort()` sorts the array *in place*, changing its element order.

It also returns the sorted array, but the returned value is usually ignored, as `arr` itself is modified.

For instance:

```
1 let arr = [ 1, 2, 15 ];
2
3 // the method reorders the content of arr
4 arr.sort();
5
6 alert( arr ); // 1, 15, 2
```

Did you notice anything strange in the outcome?

The order became `1, 15, 2`. Incorrect. But why?

The items are sorted as strings by default.

Literally, all elements are converted to strings for comparisons. For strings, lexicographic ordering is applied and indeed `"2" > "15"`.

To use our own sorting order, we need to supply a function as the argument of `arr.sort()`.

The function should compare two arbitrary values and return:

```
1 function compare(a, b) {
2   if (a > b) return 1; // if the first value is greater than the second
3   if (a == b) return 0; // if values are equal
4   if (a < b) return -1; // if the first value is less than the second
5 }
```

For instance, to sort as numbers:

```
1 function compareNumeric(a, b) {
2   if (a > b) return 1;
3   if (a == b) return 0;
4   if (a < b) return -1;
5 }
6
7 let arr = [ 1, 2, 15 ];
8
9 arr.sort(compareNumeric);
10
11 alert(arr); // 1, 2, 15
```

Now it works as intended.

Let's step aside and think about what's happening. The `arr` can be an array of anything, right? It may contain numbers or strings or objects or whatever. We have a set of *some items*. To sort it, we need an *ordering function* that knows how to compare its elements. The default is a string order.

The `arr.sort(fn)` method implements a generic sorting algorithm. We don't need to care how it internally works (an optimized [quicksort](#) or [Timsort](#) most of the time). It will walk the array, compare its elements using the provided function and reorder them, all we need is to provide the `fn` which does the comparison.

By the way, if we ever want to know which elements are compared – nothing prevents us from alerting them:

```
1 [1, -2, 15, 2, 0, 8].sort(function(a, b) {  
2   alert( a + " <> " + b );  
3   return a - b;  
4 });
```

The algorithm may compare an element with multiple others in the process, but it tries to make as few comparisons as possible.

i A comparison function may return any number

Actually, a comparison function is only required to return a positive number to say “greater” and a negative number to say “less”.

That allows to write shorter functions:

```
1 let arr = [ 1, 2, 15 ];  
2  
3 arr.sort(function(a, b) { return a - b; });  
4  
5 alert(arr); // 1, 2, 15
```

i Arrow functions for the best

Remember [arrow functions](#)? We can use them here for neater sorting:

```
1 arr.sort( (a, b) => a - b );
```

This works exactly the same as the longer version above.

Use `localeCompare` for strings

Remember `strings` comparison algorithm? It compares letters by their codes by default.

For many alphabets, it's better to use `str.localeCompare` method to correctly sort letters, such as `Ö`.

For example, let's sort a few countries in German:

```
1 let countries = ['Österreich', 'Andorra', 'Vietnam'];
2
3 alert( countries.sort( (a, b) => a > b ? 1 : -1) ); // Andorra, Vietn
4
5 alert( countries.sort( (a, b) => a.localeCompare(b) ) ); // Andorra, Ö
```

reverse

The method `arr.reverse` reverses the order of elements in `arr`.

For instance:

```
1 let arr = [1, 2, 3, 4, 5];
2 arr.reverse();
3
4 alert( arr ); // 5,4,3,2,1
```

It also returns the array `arr` after the reversal.

split and join

Here's the situation from real life. We are writing a messaging app, and the person enters the comma-delimited list of receivers: `John, Pete, Mary`. But for us an array of names would be much more comfortable than a single string. How to get it?

The `str.split(delim)` method does exactly that. It splits the string into an array by the given delimiter `delim`.

In the example below, we split by a comma followed by a space:

```
1 let names = 'Bilbo, Gandalf, Nazgul';
2
3 let arr = names.split(', ');
4
5 for (let name of arr) {
6   alert( `A message to ${name}.` ); // A message to Bilbo (and other na
7 }
```

The `split` method has an optional second numeric argument – a limit on the array length. If it is provided, then the extra elements are ignored. In practice it is rarely used though:

```
1 let arr = 'Bilbo, Gandalf, Nazgul, Saruman'.split(', ', 2);
2
3 alert(arr); // Bilbo, Gandalf
```

i Split into letters

The call to `split(s)` with an empty `s` would split the string into an array of letters:

```
1 let str = "test";
2
3 alert( str.split('') ); // t,e,s,t
```

The call `arr.join(glue)` does the reverse to `split`. It creates a string of `arr` items joined by `glue` between them.

For instance:

```
1 let arr = ['Bilbo', 'Gandalf', 'Nazgul'];
2
3 let str = arr.join(';'); // glue the array into a string using ;
4
5 alert( str ); // Bilbo;Gandalf;Nazgul
```

reduce/reduceRight

When we need to iterate over an array – we can use `forEach`, `for` or `for..of`.

When we need to iterate and return the data for each element – we can use `map`.

The methods `arr.reduce` and `arr.reduceRight` also belong to that breed, but are a little bit more intricate. They are used to calculate a single value based on the array.

The syntax is:

```
1 let value = arr.reduce(function(accumulator, item, index, array) {
2   // ...
3 }, [initial]);
```

The function is applied to all array elements one after another and “carries on” its result to the next call.

Arguments:

- `accumulator` – is the result of the previous function call, equals `initial` the first time (if `initial` is provided).
- `item` – is the current array item.
- `index` – is its position.
- `array` – is the array.

As the function is applied, the result of the previous function call is passed to the next one as the first argument.

So, the first argument is essentially the accumulator that stores the combined result of all previous executions. And at the end, it becomes the result of `reduce`.

Sounds complicated?

The easiest way to grasp that is by example.

Here we get a sum of an array in one line:

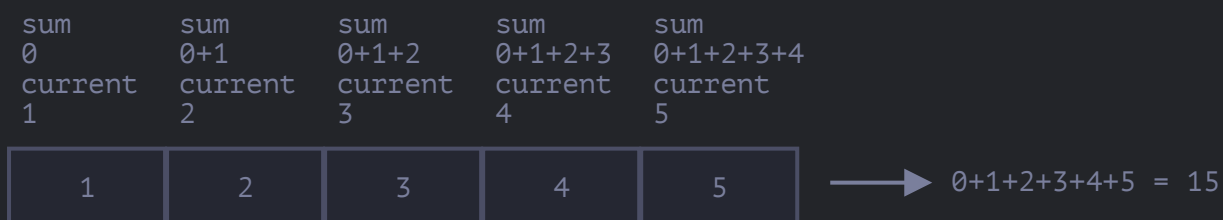
```
1 let arr = [1, 2, 3, 4, 5];
2
3 let result = arr.reduce((sum, current) => sum + current, 0);
4
5 alert(result); // 15
```

The function passed to `reduce` uses only 2 arguments, that's typically enough.

Let's see the details of what's going on.

1. On the first run, `sum` is the `initial` value (the last argument of `reduce`), equals `0`, and `current` is the first array element, equals `1`. So the function result is `1`.
2. On the second run, `sum = 1`, we add the second array element (`2`) to it and return.
3. On the 3rd run, `sum = 3` and we add one more element to it, and so on...

The calculation flow:



Or in the form of a table, where each row represents a function call on the next array element:

	sum	current	result
the first call	0	1	1
the second call	1	2	3
the third call	3	3	6
the fourth call	6	4	10
the fifth call	10	5	15

Here we can clearly see how the result of the previous call becomes the first argument of the next one.

We also can omit the initial value:

```
1 let arr = [1, 2, 3, 4, 5];
2
3 // removed initial value from reduce (no 0)
4 let result = arr.reduce((sum, current) => sum + current);
5
6 alert( result ); // 15
```

The result is the same. That's because if there's no initial, then `reduce` takes the first element of the array as the initial value and starts the iteration from the 2nd element.

The calculation table is the same as above, minus the first row.

But such use requires an extreme care. If the array is empty, then `reduce` call without initial value gives an error.

Here's an example:

```
1 let arr = [];
2
3 // Error: Reduce of empty array with no initial value
4 // if the initial value existed, reduce would return it for the empty array
5 arr.reduce((sum, current) => sum + current);
```

So it's advised to always specify the initial value.

The method `arr.reduceRight` does the same but goes from right to left.

Array.isArray

Arrays do not form a separate language type. They are based on objects.

So `typeof` does not help to distinguish a plain object from an array:

```
1 alert(typeof {}); // object
2 alert(typeof []); // object (same)
```

...But arrays are used so often that there's a special method for that: `Array.isArray(value)`. It returns `true` if the `value` is an array, and `false` otherwise.

```
1 alert(Array.isArray({})); // false
2
3 alert(Array.isArray([])); // true
```

Most methods support “thisArg”

Almost all array methods that call functions – like `find`, `filter`, `map`, with a notable exception of `sort`, accept an optional additional parameter `thisArg`.

That parameter is not explained in the sections above, because it's rarely used. But for completeness, we have to cover it.

Here's the full syntax of these methods:

```
1 arr.find(func, thisArg);
2 arr.filter(func, thisArg);
3 arr.map(func, thisArg);
4 // ...
5 // thisArg is the optional last argument
```

The value of `thisArg` parameter becomes `this` for `func`.

For example, here we use a method of `army` object as a filter, and `thisArg` passes the context:

```
1 let army = {
2   minAge: 18,
3   maxAge: 27,
4   canJoin(user) {
5     return user.age >= this.minAge && user.age < this.maxAge;
6   }
7 };
8
9 let users = [
10  {age: 16},
11  {age: 20},
12  {age: 23},
13  {age: 30}
14 ];
15
16 // find users, for who army.canJoin returns true
17 let soldiers = users.filter(army.canJoin, army);
18
19 alert(soldiers.length); // 2
20 alert(soldiers[0].age); // 20
21 alert(soldiers[1].age); // 23
```

If in the example above we used `users.filter(army.canJoin)`, then `army.canJoin` would be called as a standalone function, with `this=undefined`, thus leading to an instant error.

A call to `users.filter(army.canJoin, army)` can be replaced with `users.filter(user => army.canJoin(user))`, that does the same. The latter is used more often, as it's a bit easier to understand for most people.

Summary

A cheat sheet of array methods:

- To add/remove elements:
 - `push(...items)` – adds items to the end,
 - `pop()` – extracts an item from the end,
 - `shift()` – extracts an item from the beginning,
 - `unshift(...items)` – adds items to the beginning.
 - `splice(pos, deleteCount, ...items)` – at index `pos` deletes `deleteCount` elements and inserts `items`.
 - `slice(start, end)` – creates a new array, copies elements from index `start` till `end` (not inclusive) into it.
 - `concat(...items)` – returns a new array: copies all members of the current one and adds `items` to it. If any of `items` is an array, then its elements are taken.
- To search among elements:
 - `indexOf/lastIndexOf(item, pos)` – look for `item` starting from position `pos`, and return the index or `-1` if not found.
 - `includes(value)` – returns `true` if the array has `value`, otherwise `false`.
 - `find/filter(func)` – filter elements through the function, return first/all values that make it return `true`.
 - `findIndex` is like `find`, but returns the index instead of a value.
- To iterate over elements:
 - `forEach(func)` – calls `func` for every element, does not return anything.
- To transform the array:
 - `map(func)` – creates a new array from results of calling `func` for every element.
 - `sort(func)` – sorts the array in-place, then returns it.
 - `reverse()` – reverses the array in-place, then returns it.
 - `split/join` – convert a string to array and back.
 - `reduce/reduceRight(func, initial)` – calculate a single value over the array by calling `func` for each element and passing an intermediate result between the calls.
- Additionally:
 - `Array.isArray(value)` checks `value` for being an array, if so returns `true`, otherwise `false`.

Please note that methods `sort`, `reverse` and `splice` modify the array itself.

These methods are the most used ones, they cover 99% of use cases. But there are few others:

- `arr.some(fn)/arr.every(fn)` check the array.

The function `fn` is called on each element of the array similar to `map`. If any/all results are `true`, returns `true`, otherwise `false`.

These methods behave sort of like `||` and `&&` operators: if `fn` returns a truthy value, `arr.some()` immediately returns `true` and stops iterating over the rest of items; if `fn` returns a falsy value, `arr.every()` immediately returns `false` and stops iterating over the rest of items as well.

We can use `every` to compare arrays:

```
1 function arraysEqual(arr1, arr2) {  
2   return arr1.length === arr2.length && arr1.every((value, index) => \,  
3 }  
4  
5 alert( arraysEqual([1, 2], [1, 2])); // true
```

- `arr.fill(value, start, end)` – fills the array with repeating `value` from index `start` to `end`.
- `arr.copyWithin(target, start, end)` – copies its elements from position `start` till position `end` into *itself*, at position `target` (overwrites existing).
- `arr.flat(depth)/arr.flatMap(fn)` create a new flat array from a multidimensional array.

For the full list, see the [manual](#).

At first sight, it may seem that there are so many methods, quite difficult to remember. But actually, that's much easier.

Look through the cheat sheet just to be aware of them. Then solve the tasks of this chapter to practice, so that you have experience with array methods.

Afterwards whenever you need to do something with an array, and you don't know how – come here, look at the cheat sheet and find the right method. Examples will help you to write it correctly. Soon you'll automatically remember the methods, without specific efforts from your side.

✓ Tasks

Translate `border-left-width` to `borderLeftWidth`

importance: 5

Write the function `camelize(str)` that changes dash-separated words like “my-short-string” into camel-cased “myShortString”.

That is: removes all dashes, each word after dash becomes uppercased.

Examples:

```
1 camelize("background-color") == 'backgroundColor';  
2 camelize("list-style-image") == 'listStyleImage';  
3 camelize("-webkit-transition") == 'WebkitTransition';
```

P.S. Hint: use `split` to split the string into an array, transform it and `join` back.

[Open a sandbox with tests.](#)

[solution](#)

Filter range

importance: 4

Write a function `filterRange(arr, a, b)` that gets an array `arr`, looks for elements with values higher or equal to `a` and lower or equal to `b` and return a result as an array.

The function should not modify the array. It should return the new array.

For instance:

```
1 let arr = [5, 3, 8, 1];
2
3 let filtered = filterRange(arr, 1, 4);
4
5 alert( filtered ); // 3,1 (matching values)
6
7 alert( arr ); // 5,3,8,1 (not modified)
```

[Open a sandbox with tests.](#)

solution

Filter range "in place"

importance: 4

Write a function `filterRangeInPlace(arr, a, b)` that gets an array `arr` and removes from it all values except those that are between `a` and `b`. The test is: $a \leq arr[i] \leq b$.

The function should only modify the array. It should not return anything.

For instance:

```
1 let arr = [5, 3, 8, 1];
2
3 filterRangeInPlace(arr, 1, 4); // removed the numbers except from 1 to 4
4
5 alert( arr ); // [3, 1]
```

[Open a sandbox with tests.](#)

solution

Sort in decreasing order

importance: 4

```
1 let arr = [5, 2, 1, -10, 8];
2
3 // ... your code to sort it in decreasing order
4
5 alert( arr ); // 8, 5, 2, 1, -10
```

solution

Copy and sort array

importance: 5

We have an array of strings `arr`. We'd like to have a sorted copy of it, but keep `arr` unmodified.

Create a function `copySorted(arr)` that returns such a copy.

```
1 let arr = ["HTML", "JavaScript", "CSS"];
2
3 let sorted = copySorted(arr);
4
5 alert( sorted ); // CSS, HTML, JavaScript
6 alert( arr ); // HTML, JavaScript, CSS (no changes)
```

solution

Create an extendable calculator

importance: 5

Create a constructor function `Calculator` that creates “extendable” calculator objects.

The task consists of two parts.

1.

First, implement the method `calculate(str)` that takes a string like `"1 + 2"` in the format “NUMBER operator NUMBER” (space-delimited) and returns the result. Should understand plus `+` and minus `-`.

Usage example:

```
1 let calc = new Calculator;
2
3 alert( calc.calculate("3 + 7") ); // 10
```

2.

Then add the method `addMethod(name, func)` that teaches the calculator a new operation. It takes the operator `name` and the two-argument function `func(a,b)` that implements it.

For instance, let's add the multiplication `*`, division `/` and power `**`:

```
1 let powerCalc = new Calculator;
2 powerCalc.addMethod("*", (a, b) => a * b);
3 powerCalc.addMethod("/", (a, b) => a / b);
4 powerCalc.addMethod("**", (a, b) => a ** b);
5
6 let result = powerCalc.calculate("2 ** 3");
7 alert( result ); // 8
```

- No parentheses or complex expressions in this task.
- The numbers and the operator are delimited with exactly one space.
- There may be error handling if you'd like to add it.

[Open a sandbox with tests.](#)

[solution](#)

Map to names

importance: 5

You have an array of `user` objects, each one has `user.name`. Write the code that converts it into an array of names.

For instance:

```
1 let john = { name: "John", age: 25 };
2 let pete = { name: "Pete", age: 30 };
3 let mary = { name: "Mary", age: 28 };
4
5 let users = [ john, pete, mary ];
6
7 let names = /* ... your code */
8
9 alert( names ); // John, Pete, Mary
```

[solution](#)

Map to objects

importance: 5

You have an array of `user` objects, each one has `name`, `surname` and `id`.

Write the code to create another array from it, of objects with `id` and `fullName`, where `fullName` is generated from `name` and `surname`.

For instance:

```
1 let john = { name: "John", surname: "Smith", id: 1 };
2 let pete = { name: "Pete", surname: "Hunt", id: 2 };
3 let mary = { name: "Mary", surname: "Key", id: 3 };
4
5 let users = [ john, pete, mary ];
6
7 let usersMapped = /* ... your code ... */
8
9 /*
10 usersMapped = [
11   { fullName: "John Smith", id: 1 },
12   { fullName: "Pete Hunt", id: 2 },
13   { fullName: "Mary Key", id: 3 }
14 ]
15 */
16
17 alert( usersMapped[0].id ) // 1
18 alert( usersMapped[0].fullName ) // John Smith
```

So, actually you need to map one array of objects to another. Try using `=>` here. There's a small catch.

[solution](#)

Sort users by age

importance: 5

Write the function `sortByAge(users)` that gets an array of objects with the `age` property and sorts them by `age`.

For instance:

```
1 let john = { name: "John", age: 25 };
2 let pete = { name: "Pete", age: 30 };
3 let mary = { name: "Mary", age: 28 };
4
5 let arr = [ pete, john, mary ];
6
7 sortByAge(arr);
8
9 // now: [john, mary, pete]
10 alert(arr[0].name); // John
11 alert(arr[1].name); // Mary
12 alert(arr[2].name); // Pete
```

[solution](#)

Shuffle an array

importance: 3

Write the function `shuffle(array)` that shuffles (randomly reorders) elements of the array.

Multiple runs of `shuffle` may lead to different orders of elements. For instance:

```
1 let arr = [1, 2, 3];
2
3 shuffle(arr);
4 // arr = [3, 2, 1]
5
6 shuffle(arr);
7 // arr = [2, 1, 3]
8
9 shuffle(arr);
10 // arr = [3, 1, 2]
11 // ...
```

All element orders should have an equal probability. For instance, `[1,2,3]` can be reordered as `[1,2,3]` or `[1,3,2]` or `[3,1,2]` etc, with equal probability of each case.

[solution](#)

Get average age

importance: 4

Write the function `getAverageAge(users)` that gets an array of objects with property `age` and returns the average age.

The formula for the average is $(age1 + age2 + \dots + ageN) / N$.

For instance:

```
1 let john = { name: "John", age: 25 };
2 let pete = { name: "Pete", age: 30 };
3 let mary = { name: "Mary", age: 29 };
4
5 let arr = [ john, pete, mary ];
6
7 alert( getAverageAge(arr) ); // (25 + 30 + 29) / 3 = 28
```

[solution](#)

Filter unique array members

importance: 4

Let `arr` be an array.

Create a function `unique(arr)` that should return an array with unique items of `arr`.

For instance:

```
1 function unique(arr) {
2   /* your code */
3 }
4
5 let strings = ["Hare", "Krishna", "Hare", "Krishna",
6   "Krishna", "Krishna", "Hare", "Hare", ":-0"
7 ];
8
9 alert( unique(strings) ); // Hare, Krishna, :-0
```

[Open a sandbox with tests.](#)

solution

Create keyed object from array

importance: 4

Let's say we received an array of users in the form `{id:..., name:..., age:...}`.

Create a function `groupById(arr)` that creates an object from it, with `id` as the key, and array items as values.

For example:

```
1 let users = [
2   {id: 'john', name: "John Smith", age: 20},
3   {id: 'ann', name: "Ann Smith", age: 24},
4   {id: 'pete', name: "Pete Peterson", age: 31},
5 ];
6
7 let usersById = groupById(users);
8
9 /*
10 // after the call we should have:
11
12 usersById = {
13   john: {id: 'john', name: "John Smith", age: 20},
14   ann: {id: 'ann', name: "Ann Smith", age: 24},
```

```
15   pete: {id: 'pete', name: "Pete Peterson", age: 31},  
16   }  
17   */
```

Such function is really handy when working with server data.

In this task we assume that `id` is unique. There may be no two array items with the same `id`.

Please use array `.reduce` method in the solution.

[Open a sandbox with tests.](#)

[solution](#)



Previous lesson

Next lesson



Share



[Tutorial map](#)

Comments

- If you have suggestions what to improve - please [submit a GitHub issue](#) or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert few words of code, use the `<code>` tag, for several lines – wrap them in `<pre>` tag, for more than 10 lines – use a sandbox ([plnkr](#), [jsbin](#), [codepen](#)...)

333 Comments

1 Login ▾

G

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

♡ 32 • Share

Best Newest Oldest**Amit Gupta** ■

5 years ago edited

For removing Duplicates from Array

```
function removeDuplicates(array) {  
  return array.filter((a, b) => array.indexOf(a) === b)  
};
```

■ 30 ■ 0 Reply ↗

**Abdulrahman Elbasel** ■ → Amit Gupta

3 years ago edited

For those who don't understand this code:

indexOf will always only return the index of the first occurrence of "a" (the string) in the array and then you are comparing it with "b" (the index of the current item being iterated over)

Now, if the index of the first occurrence of the string === the index of the current item being compared, this means that the current item being iterated over is its own first occurrence in the array.

and subsequent comparisons for the same item will always be false as duplicate values will not have the same index as the first occurrence of the item.

■ 9 ■ 0 Reply ↗

**Sameer Izaj** ■ → Abdulrahman Elbasel

a year ago

i got it but is this only if we're using in the higher order function or like ?

■ 0 ■ 0 Reply ↗

**Betto Raite** ■ → Sameer Izaj

a year ago

Apparently it works in the same way.

```
function removeDuplicates(arr) {  
  if (!Array.isArray(arr)) {  
    console.error("An array data type is expected.");  
    return null;  
  }  
  const uniqueValues = [];  
  for (let i = 0; i < arr.length; ++i) {  
    const item = arr[i];  
    if (arr.indexOf(item) === i) {  
      uniqueValues.push(item);  
    }  
  }  
  return uniqueValues;  
  // return arr.filter((a, b) => arr.indexOf(a) === b);  
}
```

0 0 Reply



issam seghir → Amit Gupta

2 years ago

you can use set for this its **easiest** and **simplest**:

```
let array = [1, 1, 1, 2, 2, 3, 4, 5, 6, 6];  
let removeDuplicates = [...new Set(array)];  
// it's the same as : let removeDuplicates = Array.from(new Set(array));  
console.log(removeDuplicates);
```

3 0 Reply



victorxph → Amit Gupta

a year ago

I changed the arguments names and it make easier for me to understand

```
function unique(arr) {  
  
  return arr.filter((current, index) => arr.indexOf(current) === index);  
  
}
```

0 0 Reply



Chris → Amit Gupta

2 years ago

Doesn't array.indexOf() also loop through the array?

0 0 Reply



sapitando → Amit Gupta

3 years ago

```
const unique = (arr) =>
arr.filter(
function(item) { return this[item] ? false : this[item] = true; }, {});
```

0 0 Reply

B**Baxtiyor** → Amit Gupta

4 years ago

I don't get it.

Won't it be always:

```
array.filter((item, index) => array.indexOf(item) === index)
```

true?

What is the point of this code?

0 0 Reply

P**Perry** → Baxtiyor

4 years ago

It's the same as:

```
function removeDuplicates(array) {
return (
array.filter( (item, currentIndex) => {
let firstFoundIndex = array.indexOf(item);
return firstFoundIndex === currentIndex;
})
);
}
```

2 0 Reply

**Pigeon** → Baxtiyor

4 years ago edited

indexOf() always finds FIRST element that fulfills the condition

1 0 Reply

**ヒラヨキ** → Baxtiyor

4 years ago

If you run it, you'll see it in action. It's the inverse of checking for filtering out duplicates using .filter.

0 0 Reply

**James Shisiah**

5 years ago

One of the best articles to help one ace a JS interview test

24 1 Reply



علي
5 years ago

I solved the unique names task this way, and I'm really proud of this :

```
function unique(names) {  
  return names.filter((name,ind)=> names.indexOf(name,ind+1) == -1);  
}
```

6 0 Reply



Hào Nguyễn Đức
4 years ago edited

```
function unique(arr) {  
  return arr.filter((item, index) => !arr.includes(item, index + 1));  
}
```

I'm proud of this too

2 0 Reply



Marcos
5 years ago

Great solution. Pretty clever. It seems more efficient than using the arr.includes() method.

0 0 Reply



Farid
5 years ago

i think for this Lesson , some tasks is little hard :)

8 1 Reply



Shayan
2 years ago

Yes, some task are a bit too hard

1 0 Reply



adarsh vaidya
2 years ago

yup, found someone who responded 10 hours ago. What's your progress?

1 0 Reply



This comment was deleted.

**dracyoda** Guest

2 years ago

fluency in js but don't understand the need of generators yet.. yeah mid-level dev with milk tooth.. true imposter

0 3 Reply

**Naveen Kumar** adarsh vaidya

2 years ago

Hello

0 0 Reply

**Dawid Kopyś**

4 years ago edited

The requirement to use **.reduce** in the "Create keyed object from array" task seems counterintuitive.

It is much straightforward to do simply looping through the array with either **for..of** or **.forEach**:

```
function groupId(arr) {  
  let result = {};  
  arr.forEach( (item) => result[item.id] = item );  
  return result;  
}
```

The use of **.reduce** feels forced. It does not feel the method is the right tool for this task.

4 0 Reply

**Kuro** Dawid Kopyś

3 years ago

Definitely. It feels forced, I like your approach better.

0 0 Reply

**Barok Dagim** Kuro

2 years ago edited

I also had the same question but I think I realised why they asked us to do it. I was doing it like this at first:

```
function groupId(users) {  
  return users.reduce(  
    (accumulator, currentUser) => (accumulator[currentUser.id] = currentUser),  
    {}  
  );  
}
```

but I kept getting the wrong result. I had forgotten that a few lessons back we learned that every operator returns a value and that's also the case for the equal sign operator. In every iteration I was returning the value of

currentUser(which is what is returned by the "=" operator) instead of the accumulator itself.

To fix it I had to return the accumulator on a separate line like this:

```
function groupId(users) {
  return users.reduce((accumulator, current) => {
    accumulator[current.id] = current;
    return accumulator;
  }, {});
}
```

Maybe this is why they asked us to solve the problem using the reduce method even though there's an easier and more elegant way. Or maybe I'm wrong and this only applies to me 😊.

5 0 Reply ↗



Brahim Lte

4 years ago

Filter unique array members:

```
let strings = ["Hare", "Krishna", "Hare", "Krishna",
"Krishna", "Krishna", "Hare", "Hare", ":-0"
];
```

```
function unique(strings) {
  return strings
  .filter((item, index, str) => {
    return str.indexOf(item) === index;
  })
}
```

```
alert(unique(strings)); // Hare, Krishna, :-0
```

3 0 Reply ↗



Eva Lavinia Bucur

4 years ago

Regarding the unique array problem:

```
function unique(arr) {
  return arr.reduce((uniqueArr, item) => {
    if (!uniqueArr.includes(item)) {
      uniqueArr.push(item);
    }
  }, []);

  return uniqueArr;
}
```

Looks a little simpler to me.

3 0 Reply



Chindiana Jones → Eva Lavinia Bucur

a year ago

```
function unique(arr) {  
  return arr.filter((elem, index, array) =>  
    !array.slice(index+1).includes(elem));  
}
```

looks even simpler to me

1 0 Reply



Mustafa Husseiny

5 years ago

Arrays one of the most important data structure in web development sooner or later you will need really to master it and learn it in depth. And when it comes to sort remember to learn more about bubble and selection sort algorithms you will know about how sort works it's pure mathematics.

3 0 Reply



Михаил Петров

5 years ago

Don't we have to pass an argument to `array.canJoin` in this piece of code from the section about "thisArg"? How does it work? Can someone explain, please? [View](#) — uploads.disquscdn.com

3 0 Reply



Musaddiq manzoor → Михаил Петров

5 years ago

I really didn't get this part of the tutorial. Would it be OK to come back to it later?

0 0 Reply



the 1stgeek → Musaddiq manzoor

4 years ago

ohh it's been 7months now. How's your JS journey now?

0 0 Reply



NO NAME 7 → the 1stgeek

3 years ago

8 months later you're and now I'm writing, How's your JS journey now? :)



the 1stgeek → NO NAME 7

3 years ago

A lotttt has happened in 8months. I'm in medical school now so I'm on a break from programming but I did raise my skill to an okay level. I was able to solve programming challenges on codewars and looking forward to going fully into tech once I graduate.

1 0 Reply



Diego Machin → the 1stgeek

a year ago

well dude time has passed ¿what have you done? a dev now?

0 0 Reply



Diego Machin → NO NAME 7

a year ago

well 2 years has passed since then.. ¿what have you done?

0 0 Reply



Sreyas Suresh → Михаил Петров

5 years ago

Yes. We do need an argument but remember the first argument of the filter function should be a function and `array.canJoin` is a function that is already declared which does indeed takes an argument.

I hope this clears your doubt.

0 0 Reply



Stefan

5 years ago

This lesson is HARD!...does anyone know any other external sources with more in depth explanation of everything in this lesson? I did 5 out of 13 tasks on my own... I feel dumb...

2 0 Reply



Stefan → Stefan

5 years ago edited

Problem solved, this other resource helped me think in terms of object and arrays, check it out and give it a go if you struggled in this lesson

<https://watchandcode.com/>

1 0 Reply



G. ShadowBroker

5 years ago

I found some of these exercises really hard, I couldn't finish most of them even though I haven't skipped a single lesson so far. I feel like I'm doing something terribly wrong.

 2  0 Reply **The 25th night Bam** → G. ShadowBroker— 

5 years ago

Same. It makes me feel really dumb. But some of these are really fking hard. Like that calculator task. Why is that on this chapter? That's more about object than array method.

 2  0 Reply **Lyka**  → G. ShadowBroker— 

5 years ago

im crying 😭

 1  0 Reply **Coders Underdogs**  → G. ShadowBroker— 

5 years ago

Its ok . It seems pretty hard during the previous journey. As you keep on exploring the details and keep on understanding the key concept, the problem will be much easier.

 0  0 Reply **Marcin** — 

5 years ago

This is my fast solution for the last task. Do you find it efficient ot it's just a mess?

```
function unique(array) {  
  let obj = {};  
  array.forEach(item => obj[item] = item);  
  let arr = [];  
  for(let item in obj) {  
    arr.push(item);  
  }  
  return arr;  
}
```

 2  0 Reply **Shweta** → Marcin— 


2 years ago

If we are not using Typescript than we can also implement it as

```
function unique(arr) {  
  /* your code */  
  let res = new Set(arr);  
  return [...res];  
}
```

 1  0 Reply 

M

Michael Song  [→ Marcin](#)

5 years ago

I think that depends on the complexity of accessing a key in a object, i.e. the complexity of `obj[item]`. If it is $O(N)$, your solution would be of a complexity of $O(N^2)$. If it is $O(1)$, then your solution would be $O(N)$. That's good I think.

```
"use strict";
```

```
function unique(arr) {  
  arr.sort((a, b) => {  
    if (a > b) return 1;  
    if (a == b) return 0;  
    if (a < b) return -1;  
  });
```

```
  let index = 0;
```

```
  while (index + 1 < arr.length) {  
    if (arr[index] == arr[index + 1]) {  
      arr.splice(index + 1, 1);
```

[see more](#)

0



0

[Reply](#)**Bhavy Bhatia**  [→ Marcin](#)

5 years ago

Correct ! Key to any object can't be duplicate.



0



0

[Reply](#)

HD

Huge Dork

2 years ago

These tasks have real "how to draw an owl" energy



3



1

[Reply](#)

J

Janssen Kuhn 

5 years ago

IMO "create extendable calculator" involves a lot of stuff with objects and methods that you

