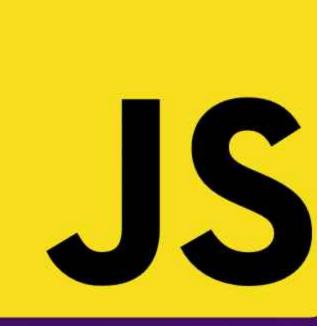Ultimate Courses™

JavaScript Array Methods

by Todd Motto

Google Developer Expert

JS

Master the common Array methods:
Reduce, Find, Filter, Every, Some, ForEach and Map

Google Developer Expert

Follow @toddmotto

Hi there! I'm Todd Motto[1] - author of this eBook and founder of Ultimate Courses[2].

This eBook is my minimal guide for you to learn some of JavaScript's most popular array methods:

- ForEach
- Map
- Filter
- Reduce
- Some
- Every
- Find

You're about to explore the syntax, function signatures, use cases, how to remember, my tips and tricks - sprinkled amongst examples that will teach you the fundamentals you need to master these array methods and remember them forever.

Whether you're a beginner or senior JavaScript developer, I'm sure you'll find lots of useful content inside.

Please tweet me and let me know[3] you enjoy the eBook!

P.S. I would highly encourage you to check out my expert JavaScript courses[4] I'm building over on Ultimate Courses - they'll teach you so much more!

Learn JavaScript at https://ultimatecourses.com

---

[1] https://twitter.com/toddmotto
[2] https://ultimatecourses.com
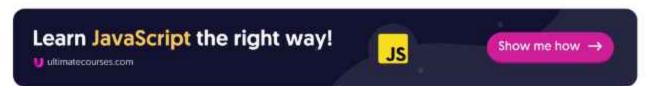[3] https://twitter.com/toddmotto
[4] https://ultimatecourses.com/courses/javascript

Contents

Array ForEach

What is Array ForEach?

Array ForEach is a method that exists on the `Array.prototype` that was introduced in ECMAScript 5 (ES5) and is supported in all modern browsers.

Array ForEach is the entry-level loop tool that will iterate over your array and pass you each value (and its index). You could render the data to the DOM for example. ForEach, unlike other array methods, does not return any value.

Think of Array ForEach as: "I want to access my array values one-by-one so I can do something with them"

Here's the syntax for Array ForEach:

```
1   array.forEach((value, index, array) => {...}, thisArg);
```

ForEach is also considered a "modern for...in loop" replacement, despite behaving differently and having less flexibility.

ForEach doesn't allow us to `break` the loop or even loop backwards, which may be a common scenario you'll run into - so it's certainly not a flat-out replacement. However, it's a pretty standard approach nowadays.

Array ForEach syntax deconstructed:

- ForEach's first argument is a callback function that exposes these parameters:
    - `value` (the current element)
    - `index` (the element's index - fairly commonly used)
    - `array` (the array we are looping - rarely used)

- Inside the body of the function we would access `value` and do something with us, sum values, inject template into the DOM, anything you like
- Every's second <u>argument</u> `thisArg` allows the this context[5] to be changed

See the ECMAScript Array ForEach specification[6]!

In its simplest form, here is how ForEach behaves:

```
1  ['a', 'b', 'c', 'd'].forEach(function(item, index) {
2    console.log(item, index);
3  });
```

We would then see something like this being printed out in the `console` (note the index number too):

```
1  a    0
2  b    1
3  c    2
4  d    3
```

Accessing each array element, and its index, couldn't be easier!

Once your array has been 'looped', ForEach will exit and your JavaScript program will continue.

Using Array ForEach

Here's our data structure that we'll be using Array ForEach with (we'll be using this same array throughout this series):

```
1  const items = [
2    { id: '🍔', name: 'Super Burger', price: 399 },
3    { id: '🍟', name: 'Jumbo Fries', price: 199 },
4    { id: '🥤', name: 'Big Slurp', price: 299 }
5  ];
```

Let's move to a more realistic scenario - we want to display the details of each array object to the user, the `id`, `name` and `price`!

We can construct a new `<li>` template for each array element, and inject it into the DOM:

---

[5]https://ultimatecourses.com/blog/understanding-the-this-keyword-in-javascript
[6]https://tc39.es/ecma262/#sec-array.prototype.foreach

```
1    const app = document.querySelector('#app');
2
3    items.forEach(item => {
4      app.innerHTML += `
5        <li>
6          ${item.id} ${item.name} - ${(item.price / 100).toFixed(2)}
7        </li>
8      `;
9    });
```

Nicely done. Rendering the data to the DOM is a great use case for ForEach, notice how we also do not return anything as well (which we will have to with most other array methods in this series).

Give the live Array ForEach demo[7] a try.

Bonus: ForEach-ing without ForEach

Let's check out a for...in loop example that mimics the behaviour of Array ForEach:

```
1    const app = document.querySelector('#app');
2
3    for (let i = 0 ; i < items.length; i++) {
4      const item = items[i];
5      app.innerHTML += `
6        <li>
7          ${item.id} ${item.name} - ${(item.price / 100).toFixed(2)}
8        </li>`;
9    }
```

Both achieve the identical output, it's merely how we compose the code.

Whilst technically the ForEach method on JavaScript arrays is more "functional" style, it does not return anything - therefore cannot be chained as part of a series of operators that transform data, for example .map().filter() which we'll learn about shortly.

ForEach is a nice and clean API and should be thought about as a one-way street. Here, I'm simply using it to render, there is no further action taking place and I get no return value from the method call.

We can also loop backwards and have the flexibility to break with the for...in loop.

Using ForEach or the newer for...of is preferred for simple looping.

---

[7]https://stackblitz.com/edit/js-1xn5av

Summary

You've now learned how to use Array ForEach to loop over your array elements and render some data.

ForEach is a great place to get started with JavaScript loops. Moving from a traditional `for...in` loop, the ForEach method can be introduced to bring a more functional approach and style to your programming.

> If you're enjoying this series, please keep following! To learn more JavaScript practices at this level and comprehensively, I've put together some online videos that explore the entire JavaScript language and the DOM[*] - I'd encourage you to check them out if you are serious about JavaScript.

Further tips and tricks:

- ForEach is great for simple looping as it doesn't return any values
- You cannot `break` or `continue` items inside a ForEach
- You cannot ForEach in reverse, use `for...in` or `for...of`
- You can access the array you're looping in the third argument of the callback
- You can change the `this` context via a second argument to `.forEach(callback, thisArg)` so that any references to `this` inside your callback point to your object
- You can use arrow functions with ForEach but remember that `this` will be incorrect if you also supply a `thisArg` due to arrow functions not having a `this` context
- Using ForEach will skip empty array slots
- You shouldn't need to in this day and age of evergreen browsers, but use a polyfill for older browsers if necessary

---

[*] https://ultimatecourses.com/courses/javascript

Array Map

Learn JavaScript at https://ultimatecourses.com

What is Array Map?

Array Map is a method that exists on the `Array.prototype` that was introduced in ECMAScript 5 (ES5) and is supported in all modern browsers.

Array Map allows us to loop our array, access each value and return a new value for each iteration - which in turn creates a new array.

Think of Array Map as: "I want a new array containing new copies, or changes, of each array element"

You could, for example, use Map to return a specific property from an object, which would result in an array of just those properties in the order your looped them.

Here's the syntax for Array Map:

```
1   const returnValue = array.map((value, index, array) => {...}, thisArg);
```

Our `returnValue` will contain our new array of, potentially new, return values.

Array Map syntax deconstructed:

- Map's first argument is a callback function that exposes these parameters:
  - `value` (the current element)
  - `index` (the element's index - sometimes used with Map)
  - `array` (the array we are looping - rarely used)
  - Inside the body of the function we need to `return` a value, this could be your array element, a modified version of it, or a completely new calculated value, this will then tell Map what to return after completing the loop
- Map's second argument `thisArg` allows the `this` context[9] to be changed

See the ECMAScript Array Map specification[10]!

[9] https://ultimatecourses.com/blog/understanding-the-this-keyword-in-javascript
[10] https://tc39.es/ecma262/#sec-array.prototype.map

In its simplest form, here is how Map behaves:

```
1  const mapped = [1, 2, 3, 4, 5].map((x) => x * 2);
2  // [2, 4, 6, 8, 10]
3  console.log(mapped);
```

I'm using x to identify whatever the value is and simply multiplying it by 2, giving us a new array of exactly each number doubled from the previous array. The original array would remain untouched and still accessible.

It's common to deal with all kinds of data with Map, as Arrays allow any value type, from primitive values through to Objects - giving us great programming flexibility.

So that's the basics of Map, let's take a look at a more real-world scenario where we've tasked with mapping an Array of Objects.

Using Array Map

Here's our data structure that we'll be using Array Map with:

```
1  const items = [
2    { id: '🍔', name: 'Super Burger', price: 399 },
3    { id: '🍟', name: 'Jumbo Fries', price: 199 },
4    { id: '🥤', name: 'Big Slurp', price: 299 }
5  ];
```

Let's assume we've just applied a coupon which applies HALF OFF our `Jumbo Fries`. We'd need to loop through our data and update that specific object.

Here's how we could solve that via Map by conditionally returning a new representation of the `item` object, with an updated `price`, otherwise we just return the `item`:

```
1  const halfOffFries = items.map(item => {
2    if (item.id === '🍟') {
3      return {
4        ...item,
5        price: item.price / 2
6      };
7    }
8    return item;
9  });
10
11 // log the return value
12 console.log(halfOffFries);
```

Using Array Map is an immutable pattern as it creates a new array from an existing array. We are also using the `...spread` operator to return a new object instead of mutating the existing one. Both operations do not mutate existing data structures and are considered immutable ways of achieving state change.

This would then give us some half price fries (which can only be good news):

```
1  [
2    { id: '🍔', name: 'Super Burger', price: 399 },
3    { id: '🍟', name: 'Jumbo Fries', price: 99.5 },
4    { id: '🥤', name: 'Big Slurp', price: 299 }
5  ]
```

Interestingly, our original `items` array remains unmodified, and we have a new collection to deal with now in our `halfOffFries` variable. This practice is called an immutable operation as we don't mutate the initial array.

Give the live Array Map demo[11] a try.

Bonus: Map-ing without Map

Let's check out a for...in loop example that mimics the behaviour of Array Map:

```
1   const halfOffFries = [];
2
3   for (let i = 0 ; i < items.length; i++) {
4     const item = items[i];
5     if (item.id === '🍟') {
6       halfOffFries.push({
7         ...item,
8         price: item.price / 2
9       });
10    } else {
11      halfOffFries.push(item);
12    }
13  }
```

First we declare halfOffFries as an empty array. Inside the loop we use pretty much the same logic, but instead of a return statement we use the Array.prototype.push method which adds each item to the new halfOffFries array.

Once the loop as finished, you're free to work with your new halfOffFries array.

This also demonstrates us the power and flexibility of using Map and other array prototype methods. The code is far smaller, promotes better practices, is easier to read and far more contained.

Summary

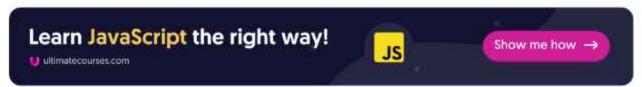You've now learned how to use Array Map to map your array to a new set of values.

Map is the next best place to begin after getting started with array ForEach. Moving from a traditional for...in loop, the Map method can be introduced to bring a more functional approach and style to your programming.

> If you are serious about your JavaScript skills, your next step is to take a look at my JavaScript courses[12], they will teach you the full language, the DOM, the advanced stuff and much more!

Further tips and tricks:

- Use Map to create a new collection with changed values of your initial collection
- Don't forget to return or your values will be undefined
- Map will shallow copy your object references into the new array
- Don't encourage bad habits by using .map() over .forEach() just because it can have the same effect and is 'shorter' - use the right tool for the right job or you will confuse people!

[12]https://ultimatecourses.com/courses/javascript

- You can access the array you're looping in the third argument of the callback
- You can change the `this` context via a second argument to `.map(callback, thisArg)` so that any references to `this` inside your callback point to your object
- You can use arrow functions with Map but remember that `this` will be incorrect if you also supply a `thisArg` due to arrow functions not having a `this` context
- Like ForEach and friends, you cannot Map in reverse or break a Map, use `for...in` or `for...of`
- Using Map will skip empty array slots
- You shouldn't need to in this day and age of evergreen browsers, but use a polyfill for older browsers if necessary

Array Filter

Learn JavaScript at https://ultimatecourses.com

What is Array Filter?

Array Filter is a method that exists on the `Array.prototype` that was introduced in ECMAScript 5 (ES5) and is supported in all modern browsers.

Array Filter allows us to conditionally return certain elements from our array, into a new array. It's commonly used to remove items from an array by excluding them from the result.

Think of Array Filter as: "I want a new array containing just the items I need"

With each iteration of a Filter loop, you will need to return an expression that Filter evaluates either `true` or `false`.

The key to understanding Filter is to realise your callback is returning an expression that will evaluate to `true` or `false`.

Array elements that evaluate to `true` are stored in a new array - and items that evaluate to `false` are excluded. Once Filter has completed, you can access the new array of your values.

Here's the syntax for Array Filter:

```
1   const returnValue = array.filter((value, index, array) => {...}, thisArg);
```

Our `returnValue` will contain our new array of filtered return values.

Array Filter syntax deconstructed:

- Filter's first argument is a callback function that exposes these parameters:
    - `value` (the current element)
    - `index` (the element's index - sometimes used with Filter)

- array (the array we are looping - rarely used)
- Inside the body of the function we need to `return` an expression which will evaluate to a Boolean (`true` or `false`)
- Filter's second <u>argument</u> `thisArg` allows the this context[13] to be changed

See the ECMAScript Array Filter specification[14]!

In its simplest form, here is how Filter behaves:

```
1  const array = [true, true, false];
2
3  // [true, true]
4  console.log(array.filter(Boolean));
```

Array Filter expects us to return an <u>expression</u> that will evaluate `true` or `false`, we can go straight to the finish line and make it easier by supplying literally `true` and `false` array values.

I'm using JavaScript's `Boolean` object to coerce the array element to a Boolean. As our array already contains Boolean values, any `false` values are omitted.

Notice how Filter is also returning <u>multiple</u> values, compared to its closely related sibling method Array Find.

Using Array Filter

Here's our data structure that we'll be using Array Filter with:

```
1  const items = [
2    { id: '🍔', name: 'Super Burger', price: 399 },
3    { id: '🍟', name: 'Jumbo Fries', price: 199 },
4    { id: '🥤', name: 'Big Slurp', price: 299 }
5  ];
```

Here let's assume we want to create a new array of more expensive items in this array, we can consider "expensive" to be 199 and above.

Let's return an expression now that compares each item's `price` property with greater than > 199, which aims to filter out values that aren't considered expensive:

---

[13]https://ultimatecourses.com/blog/understanding-the-this-keyword-in-javascript
[14]https://tc39.es/ecma262/#sec-array.prototype.filter

```
1  const expensiveItems = items
2    .filter(item => item.price > 199);
```

This would then give us the two items that are considered "expensive" based on our 199 threshold value:

```
1  [
2    { id: '🍔', name: 'Super Burger', price: 399 },
3    { id: '🥤', name: 'Big Slurp', price: 299 }
4  ]
```

Interestingly, our original items array remains unmodified, and we have a new collection to deal with now in our expensiveItems variable. This practice is called an immutable operation as we don't mutate the initial array.

Give the live Array Filter demo[15] a try.

Bonus: Filter-ing without Filter

Let's check out a for...in loop example that mimics the behaviour of Array Filter:

```
1  const expensiveItems = [];
2
3  for (let i = 0 ; i < items.length; i++) {
4    const item = items[i];
5    if (item.price > 199) {
6      expensiveItems.push(item);
7    }
8  }
```

First we declare expensiveItems as an empty array. Inside the loop we use pretty much the same logic, but instead of a return statement we use the Array.prototype.push method which adds each item to the new expensiveItems array.

Once the loop as finished, you're free to work with your new filtered array.

Summary

You've now learned how to use Array Filter to filter your array to a specific set of values.

You can think of Filter of like a functional "if statement". If my array element meets my criteria - give it to us... Else, we don't want it.

---

[15]https://stackblitz.com/edit/js-hd1ht6

If you are serious about your JavaScript skills, your next step is to take a look at my JavaScript courses[16], they will teach you the full language, the DOM, the advanced stuff and much more!

Further tips and tricks:

- Filter can be used to return specific values from a source array
- Don't forget to `return` or your values will be `undefined` which coerces to `false` (so an undetected bug could be introduced!)
- The easiest way to get all truthy values in the array is by using `.filter(Boolean)`
- Don't forget to `return` or your values will be `undefined`
- Filter will shallow copy your object references into the new array
- Filter is also similar to Find, but for multiple items!
- You can access the array you're looping in the third argument of the callback
- You can change the `this` context via a second argument to `.filter(callback, thisArg)` so that any references to `this` inside your callback point to your object
- You can use arrow functions with Filter but remember that `this` will be incorrect if you also supply a `thisArg` due to arrow functions not having a `this` context
- Using Filter will skip empty array slots
- You shouldn't need to in this day and age of evergreen browsers, but use a polyfill for older browsers if necessary

---

[16]https://ultimatecourses.com/courses/javascript

Array Reduce

What is Array Reduce?

Array Reduce is a method that exists on the `Array.prototype` that was introduced in ECMAScript 5 (ES5) and is supported in all modern browsers.

Array Reduce is the most misunderstood array method and a headache for many developers - if only they'd read this article! Pay close attention to the little details and you'll succeed with Reduce. The concept of Reduce itself isn't complex, but the method is just a little different to others that we're used to.

Think of Array Reduce as: "I want to reduce my array to just a single value"

Array Reduce is commonly used for performing things such as math expressions and equations, for instance calculating the total of a numbers array.

> You'll have likely heard the term "Reducer" before when dealing with things such as Redux[17] or NGRX[18], a Reducer also is a "Reduce", returning a single value inside a `switch` statement.

Here's the syntax for Array Reduce:

```
1  const reducedValue = array.reduce((prev, next, index, array) => {...}, initialValue);
```

Our `reducedValue` will contain our reduced value, this is typically a number - however you can compose objects and arrays inside your Reduce, that's more of an advanced use case though.

Array Reduce syntax deconstructed:

---

[17] https://redux.js.org
[18] https://ultimatecourses.com/learn/ngrx-store-effects

- Reduce's first argument is a callback function that exposes these parameters:
  - prev (sometimes called acc for "accumulator" as the value is dynamic, it accumulates the callback's value and is returned on the next iteration as prev)
  - next (the current element, sometimes called value)
  - index (the element's index - not commonly used with Find)
  - array (the array we are looping - rarely used)
  - Inside the body of the function we need to return an expression which is then passed to the next iteration as prev - Reduce essentially remembers the value from each iteration and keeps providing you it until your array completes
- Reduce's second argument is different to other Array method counterparts we've explored so far, instead of thisArg it's initialValue - allowing us to specifiy an optional initial value for the loop to begin (which gets passed in as prev on the first iteration of the loop, with next being the first array value)
- IMPORTANT: If no initialValue is set, Reduce will use your first array value as the prev on the first iteration - technically starting the loop on the second array element

See the ECMAScript Array Reduce specification[19]!

In its simplest form, here is how Reduce behaves:

```
1  const reduced = [1, 2, 3, 4, 5].reduce((prev, next) => prev + next, 0);
2  // 15
3  console.log(reduced);
```

Simple enough, right? Even if we don't "get it" right away, we can add up 1, 2, 3, 4, 5 to reach a comfortable 15 - so we're half way there to understanding Reduce.

When Reduce begins the initialValue (here it's 0) becomes the first prev value and next is our first array value of 1. If there was no initial value then prev would be 1 (first array value) and next would be 2 (second array value).

These small differences in how Reduce behaves with and without an initialValue likely also contributes to reasons to not fully understanding Reduce.

Developers, like I did, struggled at first with this whole prev and next thing.

So let's use a really simple demonstration to make sure we get it:

```
1  const reduced = [1, 2, 3, 4, 5].reduce((prev, next) => {
2    console.log(prev, next);
3    return prev + next;
4  }, 0);
```

Outputs in the `console` (each iteration):

```
1  0   1   // 0  = initial value,              1 = first array item
2  1   2   // 1  = previous result (0 + 1)     2 = second array item
3  3   3   // 3  = previous result (1 + 2)     3 = third array item
4  6   4   // 6  = previous result (3 + 3)     4 = fourth array item
5  10  5   // 10 = previous result (6 + 4)     5 = fifth array item
```

This now explains why we get 15 as a return value (10 + 5) - there is no next value in our array so Reduce will exit and return that final value to our `reduced` variable.

We've only considered numbers at this point, however as Reduce returns any value type, it has very flexible use cases! You could return flattened Arrays, concatenated Strings, new or merged Objects - or whatever else you can come up with!

That's the basics of Reduce, let's take a look at a more real-world scenario where we've tasked with calculating the sum from Objects - there are multiple approaches I'm also going to show you.

Using Array Reduce

Here's our data structure that we'll be using Array Find with:

```
1  const items = [
2    { id: '🍔', name: 'Super Burger', price: 399 },
3    { id: '🍟', name: 'Jumbo Fries', price: 199 },
4    { id: '🥤', name: 'Big Slurp', price: 299 }
5  ];
```

Let's calculate the total price of all `price` properties.

Reducing an Array of Numbers

A basic Reduce will sum an array of numbers, so let's add an Array Map before to return us just each `price` property - giving us an array of numbers to then Reduce:

```
1  const reduced = items
2    .map(item => item.price)
3    .reduce((prev, next) => prev + next);
4
5  // Total: 8.97
6  console.log(found);
```

This is a perfectly decent example, and completes the mission we set out to achieve to calculate the total price. However, Reduce offers us a really nice way to work with Objects - which involves the use of the initialValue property...

Reducing an Array of Objects

By supplying an initialValue the prev value becomes 0 to begin the Reduce. This works nicely when adding + to our next.price:

```
1  const reduced = items
2    .reduce((prev, next) => prev + next.price, 0);
3
4  // Total: 8.97
5  console.log(reduced);
```

> If we didn't supply 0 we would log Total: NaN because we'd be attempting to add an Object to a number property!

It also saves us another Array Map, keep the code a little cleaner and more efficient.

Give the live Array Reduce demo[20] a try.

Bonus: Reduce-ing without Reduce

Let's check out a for...in loop example that mimics the behaviour of Array Reduce:

```
1  let reduced;
2  let prev = 0; // "initialValue"
3
4  for (let i = 0; i < items.length; i++) {
5    const next = items[i];
6    prev = prev + next.price;
7  }
8
9  reduced = prev;
```

First we declare `reduced` and `prev` to assign an `initialValue` of 0 just like Reduce. From there, we'll loop our `items` and create a `next` variable. We then re-assign `prev` each time and add + our `next.price` to it.

Once the loop as finished, I've assigned `prev` to `reduced` to act like Reduce's final return value.

Summary

You've now learned how to use Array Reduce to reduce your array to just a single value.

Array Reduce is a nice and compact way that we can declaratively reduce an array and return any kind of value we'd like. More advanced use cases include composing Objects and Arrays inside your Reduce, but we'll save them for another time.

> If you are serious about your JavaScript skills, your next step is to take a look at my JavaScript courses[21], they will teach you the full language, the DOM, the advanced stuff and much more!

Further tips and tricks:

- Remember to specify an `initialValue` when dealing with an Array of Objects
- Reducing Arrays of Numbers is nice and clean, try not to overcomplicate your Reduce functions - this is a common pattern I've seen
- Don't forget to `return` inside your callback, or your values will be `undefined` and evaluate to `false` - avoid undetected bugs!
- You can access the array you're looping in the third argument of the callback
- You can change the `this` context via a second argument to `.reduce(callback, thisArg)` so that any references to `this` inside your callback point to your object
- You can use arrow functions with Every but remember that `this` will be incorrect if you also supply a `thisArg` due to arrow functions not having a `this` context
- Using Find will skip empty array slots as if it were a falsy value
- You shouldn't need to in this day and age of evergreen browsers, but use a polyfill for older browsers if necessary

---

[21]https://ultimatecourses.com/courses/javascript

Array Some

What is Array Some?

Array Some is a method that exists on the `Array.prototype` that was introduced in ECMAScript 5 (ES5) and is supported in all modern browsers.

Array Some tells you whether any element in your array passes your test. If one element passes then Array Some returns `true`. Some will return `false` if no elements pass the test.

As soon as Some finds a `true` result, it will short-circuit the loop and continue no more - giving us a performance boost.

Think of Array Some as: "I want to check if any value(s) in my array meets my condition - a yes/no answer"

Here's the syntax for Array Some:

```
1  const returnValue = array.some((value, index, array) => {...}, thisArg);
```

Our `returnValue` will contain a Boolean value `true` or `false`.

As Some returns a Boolean, its result is commonly used in conditional statements.

Array Some syntax deconstructed:

- Some's first argument is a callback function that exposes these parameters:
  - `value` (the current element)
  - `index` (the element's index - sometimes used with Filter)
  - `array` (the array we are looping - rarely used)
  - Inside the body of the function we need to `return` an expression which will evaluate to a Boolean (`true` or `false`)

- Some's second argument `thisArg` allows the this context[22] to be changed

  See the ECMAScript Array Some specification[23]!

In its simplest form, here is how Some behaves:

```
1  const greaterThanOne = [1, 2, 3].some(x => x > 1);
2  // true
3  console.log(greaterThanOne);
```

As our array contains values greater than > 1, our expression evaluates to `true`, and Some returns `true`.

Similarly, with an expression that checks if our values are greater than > 3, Some will return `false`:

```
1  const greaterThanThree = [1, 2, 3].some(x => x > 3);
2  // false
3  console.log(greaterThanThree);
```

As a performance benefit, Some will short-circuit and stop the loop on a `true` test result, otherwise it will continuously loop if results are `false` until the loop exits.

Let's check some examples out.

Using Array Some

Here's our data structure that we'll be using Array Some with (take note of the additional `promo` property):

```
1  const items = [
2    { id: '🍔', name: 'Super Burger', price: 399, promo: false },
3    { id: '🍟', name: 'Jumbo Fries', price: 199, promo: false },
4    { id: '🥤', name: 'Big Slurp', price: 299, promo: true}
5  ];
```

Let's use Some to check if any items are in the promo - we should expect to see our Big Slurp '🥤' trigger Some to return `true`:

```
1  const isInPromo = items
2    .some(item => item.promo);
3
4  // true
5  console.log(isInPromo);
```

Using a ternary statement to calculate our `total` - if an item is in the promo we set the price to a flat `600` cents. Otherwise we'll use Array Reduce to sum the `price` properties:

```
1  const total = isInPromo ? 600 : items.reduce((prev, next) => prev + next.price, 0);
```

Our example here is simple, but real enough. You can see how we've used the `isInPromo` resulting variable as part of a conditional statement - where it's most commonly used!

Notice how simple Some is, we're returning `item.promo` (either `true` or `false`) to get a final `true` result as one item matched our conditional test.

> You can return any type of expression inside Some's callback function (such as using comparison logic `item.price > 99`).

Give the live Array Some demo[24] a try, you can toggle `promo: true` to `promo: false` and see the result change.

Bonus: Some-ing without Some

Let's check out a `for...in` loop example that mimics the behaviour of Array Some:

```
1  let isInPromo = false;
2
3  for (let i = 0; i < items.length; i++) {
4    const item = items[i];
5    if (item.promo) {
6      isInPromo = true;
7      break;
8    }
9  }
```

First we set `isInPromo` to `false` and it's our job to detect when to set it to `true`. We'll loop the items and if one passes, we set `isInPromo` to `true`. This is the same behaviour as Some, and if no items match then `isInPromo` will remain `false`.

---

[24]https://stackblitz.com/edit/js-vwyaqj

Summary

You've now learned how to use Array Some to run a test on your array elements. If at least one element in your array, when returned as part of an expression, evaluates to `true` then Some will exit the loop and return `true`.

If no array elements pass the test Some will return `false`.

No array items are returned back to you, Some is exclusively for returning a Boolean result. If you do want items back, Array Map and Array Filter are better methods to use.

> If you are serious about your JavaScript skills, your next step is to take a look at my JavaScript courses[25], they will teach you the full language, the DOM, the advanced stuff and much more!

Further tips and tricks:

- Don't forget to `return` inside your callback, or your values will be `undefined` and evaluate to `false` - avoid undetected bugs!
- You can access the array you're looping in the third argument of the callback
- You can change the `this` context via a second argument to `.some(callback, thisArg)` so that any references to `this` inside your callback point to your object
- You can use arrow functions with Some but remember that `this` will be incorrect if you also supply a `thisArg` due to arrow functions not having a `this` context
- Using Some will skip empty array slots as if it were a falsy value
- You shouldn't need to in this day and age of evergreen browsers, but use a polyfill for older browsers if necessary

---

[25]https://ultimatecourses.com/courses/javascript

Array Every

What is Array Every?

Array Every is a method that exists on the `Array.prototype` that was introduced in ECMAScript 5 (ES5) and is supported in all modern browsers.

Array Every tells you whether <u>every</u> element in your array passes your test. If every element passes, Every returns `true`. If just one element in the array fails, Every will return `false`.

As soon as Every finds a `false` result, it will short-circuit the loop and continue no more - giving us a performance boost.

Think of Array Every as: "I want to check if every value in my array meets my condition - a yes/no answer"

Here's the <u>syntax</u> for Array Every:

```
1  const returnValue = array.every((value, index, array) => {...}, thisArg);
```

Our `returnValue` will contain a Boolean value `true` or `false`.

As Every returns a Boolean, its result is commonly used in conditional statements.

<u>Array Every syntax deconstructed:</u>

- Every's first <u>argument</u> is a callback function that exposes these parameters:
    - `value` (the current element)
    - `index` (the element's index - not commonly used)
    - `array` (the array we are looping - rarely used)
    - Inside the body of the function we need to `return` an expression which will evaluate to a Boolean (`true` or `false`), this will then tell Every what to return after completing the loop

- Every's second argument thisArg allows the this context[26] to be changed

  See the ECMAScript Array Every specification[27]!

In its simplest form, here is how Every behaves:

```
1  const isEveryValueTrue = [true, true, true].every(Boolean);
2  // true
3  console.log(isEveryValueTrue);
```

As our array contains true values, our expression evaluates to true and Every returns true.

> Using JavaScript's Boolean object we coerce each result to a Boolean, essentially running an "all-true" check on the array. Boolean is actually a function and typically called as Boolean([1, 2, 3]) to coerce any JavaScript value to a true or false value.

Similarly, by including a false value, Every will return false:

```
1  const isEveryValueTrue = [false, true, true].every(Boolean);
2  // false
3  console.log(isEveryValueTrue);
```

As our array contains a false value, it will trigger Every to return false. As a performance benefit, Every will short-circuit and stop the loop on a false test result.

Let's check some examples out.

Using Array Every

Here's our data structure that we'll be using Array Every with (take note of the additional stock property):

```
1  const items = [
2    { id: '🍔', name: 'Super Burger', price: 399, stock: true },
3    { id: '🍟', name: 'Jumbo Fries', price: 199, stock: true },
4    { id: '🥤', name: 'Big Slurp', price: 299, stock: false }
5  ];
```

Our Jumbo Fries '🍟' are out of stock based on the `stock: false` property.

If an item is out of stock, then we'll show a message in the `console`:

```
1  const isInStock = items.every((item) => item.stock);
2
3  // true
4  console.log(isInStock);
5
6  if (!isInStock) {
7    console.log(`Sorry, ${items.find(item => !item.stock).name} is out of Stock.`);
8  }
```

Our example here is simple, but real enough. You can see how we've used the `isInStock` resulting variable as part of a conditional statement - where it's most commonly used!

Notice how simple Every is, we're returning `item.stock` (either `true` or `false`) to get a final `false` result.

> You can return any type of expression inside Every's callback function (such as using comparison logic `item.price > 99`).

Give the live Array Every demo[28] a try, you can toggle `stock: true` to `stock: false` and see the result change.

Bonus: Every-ing without Every

Let's check out a `for...in` loop example that mimics the behaviour of Array Every:

---

[28]https://stackblitz.com/edit/js-uirufh

```
1   let isInStock = true;
2
3   for (let i = 0; i < items.length; i++) {
4     const item = items[i];
5     if (!item.stock) {
6       isInStock = false;
7       break;
8     }
9   }
```

First we set isInStock to true, and we need to disprove otherwise. We'll loop the items and if one fails, we set isInStock to false. This is the same behaviour as Every, even though we're inverting the conditional statement.

Summary

You've now learned how to use Array Every to run a test on your array elements. If at least one element in your array, when returned as part of an expression, evaluates to false then Every will exit the loop and return false.

If all array elements pass the test Every will return true.

No array items are returned back to you, Every is exclusively for returning a Boolean result. If you do want items back, Array Map and Array Filter are better methods to use.

> If you are serious about your JavaScript skills, your next step is to take a look at my JavaScript courses[29], they will teach you the full language, the DOM, the advanced stuff and much more!

Further tips and tricks:

- Don't forget to return inside your callback, or your values will be undefined and evaluate to false - avoid undetected bugs!
- You can access the array you're looping in the third argument of the callback
- You can change the this context via a second argument to .every(callback, thisArg) so that any references to this inside your callback point to your object
- You can use arrow functions with Every but remember that this will be incorrect if you also supply a thisArg due to arrow functions not having a this context
- Using Every will skip empty array slots as if it were a falsy value
- You shouldn't need to in this day and age of evergreen browsers, but use a polyfill for older browsers if necessary

---

[29]https://ultimatecourses.com/courses/javascript

Array Find

What is Array Find?

Array Find is a method that exists on the `Array.prototype` that was more recently introduced in ECMAScript 2015 (ES6) and is supported in all modern browsers.

Array Find searches your array and returns you the first matching element, or `undefined`. Find's return value is dynamic and could be of any JavaScript type that exists in your array, a string, number, object etc.

Think of Array Find as: "I want to find a particular element in my array"

In ways, Array Find is similar to Array Filter, however returns just the first result, whereas Filter would return you as many results that satisfy the test!

Here's the syntax for Array Find:

```
1   const foo = array.find((value, index, array) => {...}, thisArg);
```

The value `const foo` will contain any element from your array, and therefore it could be of any value type.

Array Find syntax deconstructed:

- Find's first argument is a callback function that exposes these parameters:
    - `value` (the current element)
    - `index` (the element's index - not commonly used with Find)
    - `array` (the array we are looping - rarely used)
    - Inside the body of the function we need to `return` an expression which will evaluate to a Boolean (`true` or `false`), this will then tell Every what to return after completing the loop
- Find's second argument `thisArg` allows the this context[30] to be changed

See the ECMAScript Array Find specification[31]!

[30]https://ultimatecourses.com/blog/understanding-the-this-keyword-in-javascript
[31]https://tc39.es/ecma262/#sec-array.prototype.find

In its simplest form, here is how Find behaves:

```
1  const found = ['a', 'b', 'c', 'd'].find(x => x === 'a');
2  // 'a'
3  console.log(found);
```

Find will return us a "shallow copy" of 'a' - which is always the case with any value in our array. We are always passed a copy rather than a direct reference - which helps us mitigate potential bugs.

It will also return undefined if, for example with the value 'e', the result does not exist in the aray:

```
1  const notFound = ['a', 'b', 'c', 'd'].find(x => x === 'e');
2  // undefined
3  console.log(notFound);
```

As Find returns any value type, it has very flexible use cases! You could return Booleans, Strings, Objects, Arrays to any degree - but a common use case could be finding an object inside an array by supplying an ID to lookup the object with. We could, for example, then use the return value to perhaps update that particular element or send it to a server.

As soon as Find successfully 'finds' a first element match, it will return it to you - so keep this in mind when dealing with duplicate array items, as you will only get one result back from Find. Find will also loop in ascending order, so there should be no surprises.

Using Array Find

Here's our data structure that we'll be using Array Find with:

```
1  const items = [
2    { id: '🍔', name: 'Super Burger', price: 399 },
3    { id: '🍟', name: 'Jumbo Fries', price: 199 },
4    { id: '🥤', name: 'Big Slurp', price: 299 }
5  ];
```

We could find any item we like, via any of the properties available.

Let's use Find to find an item based on it's id property:

```
1  const found = items.find((item) => item.id === '🍔');
2
3  // { id: '🍔', name: 'Jumbo Fries', price: 199 }
4  console.log(found);
```

As found could also contain undefined, it's best practice to safety check the variable in some way:

```
1  const found = items.find((item) => item.id === '🍔');
2
3  if (found) {
4    // Jumbo Fries 1.99
5    console.log(`${found.name} ${(found.price / 100).toFixed(2)}`);
6  }
```

Nicely done. If the item is available, let's do something with the data!

Notice how simple Find is, we're simply returning item.id === '🍔' and we immediately get it back once it matches.

Give the live Array Find demo[32] a try.

Bonus: Find-ing without Find

Let's check out a for...in loop example that mimics the behaviour of Array Find:

```
1  // `undefined` by default
2  let found;
3
4  for (let i = 0; i < items.length; i++) {
5    const item = items[i];
6    if (item.id === '🍔') {
7      found = item;
8      break;
9    }
10 }
```

First we declare let found and do not assign a value. Why? Because by default, it's undefined - you can explicitly assign it if you like, though.

Inside the loop, we then find the item and assign it to the found variable, and break the loop - giving us a nice imperative "find" solution.

Summary

You've now learned how to use Array Find to grab any particular element you want in your array, in any way you want to find it.

Array Find is a nice and compact way that we can declaratively search through an array and get a copy of the first matched element.

Remember as well, Find is similar to Filter! Filter just gives you all results if they match, rather than the first result only.

> If you are serious about your JavaScript skills, your next step is to take a look at my JavaScript courses[33], they will teach you the full language, the DOM, the advanced stuff and much more!

Further tips and tricks:

- Don't forget to `return` inside your callback, or your values will be `undefined` and evaluate to `false` - avoid undetected bugs!
- You can access the array you're looping in the third argument of the callback
- You can change the `this` context via a second argument to `.find(callback, thisArg)` so that any references to `this` inside your callback point to your object
- You can use arrow functions with Find but remember that `this` will be incorrect if you also supply a `thisArg` due to arrow functions not having a `this` context
- Using Find will skip empty array slots as if it were a falsy value
- You shouldn't need to in this day and age of evergreen browsers, but use a polyfill for older browsers if necessary

---

[33]https://ultimatecourses.com/courses/javascript