JavaScript Array Methods Tutorial – The Most Useful Methods Explained with Examples

freecodecamp.org/news/complete-introduction-to-the-most-useful-javascript-array-methods

February 17, 2021

If you're a JavaScript developer and want to improve your coding, then you should be familiar with the most commonly used ES5 and ES6+ array methods.

These methods make coding a lot easier and also make your code look clean and easy to understand.

So in this article, we will explore some of the most popular and widely used array methods. So let's get started.

The Array.forEach Method

The `Array.forEach` method has the following syntax:

```
Array.forEach(callback(currentValue [, index [, array]])[, thisArg]);
```

The `forEach` method executes a provided function once for every element in the array.

Take a look at the below code:

```
const months = ['January', 'February', 'March', 'April'];

months.forEach(function(month) {
  console.log(month);
});

/* output

January
February
March
April

*/
```

Here's a Code Pen Demo.

Here, inside the `forEach` loop callback function, each element of the array is automatically passed as the first parameter of the function.

The equivalent for loop code for the above example looks like this:

```
const months = ['January', 'February', 'March', 'April'];

for(let i = 0; i < months.length; i++) {
  console.log(months[i]);
}

/* output

January
February
March
April

*/
```

Here's a Code Pen Demo.

The thing you need to keep in mind is that the `forEach` method does not return any value.

Take a look at the below code:

```
const months = ['January', 'February', 'March', 'April'];
const returnedValue = months.forEach(function (month) {
  return month;
});

console.log('returnedValue: ', returnedValue); // undefined
```

Here's a Code Pen Demo.

> *Note that* `forEach` *is only used to loop through the array and perform some processing or logging. It does not return any value, even if you explicitly return a value from the callback function (this means that the returned value comes as* `undefined` *in the above example).*

In all the above examples, we have used only the first parameter of the callback function. But the callback function also receives two additional parameters, which are:

- index - the index of the element which is currently being iterated
- array - original array which we're looping over

```
const months = ['January', 'February', 'March', 'April'];

months.forEach(function(month, index, array) {
  console.log(month, index, array);
});

/* output

January 0 ["January", "February", "March", "April"]
February 1 ["January", "February", "March", "April"]
March 2 ["January", "February", "March", "April"]
April 3 ["January", "February", "March", "April"]

*/
```

Here's a Code Pen Demo.

Depending on the requirement, you may find it useful to use the `index` and `array` parameters.

Advantages of using forEach instead of a for loop

- Using a `forEach` loop makes your code shorter and easier to understand
- When using a `forEach` loop, we don't need to keep track of how many elements are available in the array. So it avoids the creation of an extra counter variable.
- Using a `forEach` loop makes code easy to debug because there are no extra variables for looping through the array
- The `forEach` loop automatically stops when all the elements of the array are finished iterating.

Browser Support

- All modern browsers and Internet Explorer (IE) version 9 and above
- Microsoft Edge version 12 and above

The Array.map Method

The Array map method is the most useful and widely used array method among all other methods.

The `Array.map` method has the following syntax:

```
Array.map(function callback(currentValue[, index[, array]]) {
    // Return element for new_array
}[, thisArg])
```

The `map` method executes a provided function once for every element in the array and it **returns a new transformed array.**

Take a look at the below code:

```
const months = ['January', 'February', 'March', 'April'];
const transformedArray = months.map(function (month) {
  return month.toUpperCase();
});

console.log(transformedArray); // ["JANUARY", "FEBRUARY", "MARCH", "APRIL"]
```

Here's a Code Pen Demo.

In the above code, inside the callback function, we're converting each element to uppercase and returning it.

The equivalent for loop code for the above example looks like this:

```
const months = ['January', 'February', 'March', 'April'];
const converted = [];

for(let i = 0; i < months.length; i++) {
 converted.push(months[i].toUpperCase());
};

console.log(converted); // ["JANUARY", "FEBRUARY", "MARCH", "APRIL"]
```

Here's a Code Pen Demo.

Using `map` helps to avoid creating a separate `converted` array beforehand for storing the converted elements. So it saves memory space and also the code looks much cleaner using array `map`, like this:

```
const months = ['January', 'February', 'March', 'April'];

console.log(months.map(function (month) {
  return month.toUpperCase();
})); // ["JANUARY", "FEBRUARY", "MARCH", "APRIL"]
```

Here's a Code Pen Demo.

Note that the `map` method returns a new array that is of the exact same length as the original array.

The difference between the `forEach` and `map` methods is that `forEach` is only used for looping and does not return anything back. On the other hand, the `map` method returns a new array that is of the exact same length as the original array.

Also, note that `map` does not change the original array but returns a new array.

Take a look at the below code:

```
const users = [
  {
    first_name: 'Mike',
    last_name: 'Sheridan'
  },
  {
    first_name: 'Tim',
    last_name: 'Lee'
  },
  {
    first_name: 'John',
    last_name: 'Carte'
  }
];

const usersList = users.map(function (user) {
  return user.first_name + ' ' + user.last_name;
});

console.log(usersList); // ["Mike Sheridan", "Tim Lee", "John Carte"]
```

Here's a Code Pen Demo.

Here, by using the array of objects and `map` methods, we're easily generating a single array with first and last name concatenated.

In the above code, we're using the `+` operator to concatenate two values. But it's much more common to use ES6 template literal syntax as shown below:

```
const users = [
  {
    first_name: 'Mike',
    last_name: 'Sheridan'
  },
  {
    first_name: 'Tim',
    last_name: 'Lee'
  },
  {
    first_name: 'John',
    last_name: 'Carte'
  }
];

const usersList = users.map(function (user) {
  return `${user.first_name} ${user.last_name}`;
});

console.log(usersList); // ["Mike Sheridan", "Tim Lee", "John Carte"]
```

Here's a Code Pen Demo.

The array `map` method is also useful, if you want to extract only specific data from the array like this:

```
const users = [
  {
    first_name: 'Mike',
    last_name: 'Sheridan',
    age: 30
  },
  {
    first_name: 'Tim',
    last_name: 'Lee',
    age: 45
  },
  {
    first_name: 'John',
    last_name: 'Carte',
    age: 25
  }
];

const surnames = users.map(function (user) {
  return user.last_name;
});

console.log(surnames); // ["Sheridan", "Lee", "Carte"]
```

Here's a Code Pen Demo.

In the above code, we're extracting only the last names of each user and storing them in an array.

We can even use `map` to generate an array with dynamic content as shown below:

```
const users = [
  {
    first_name: 'Mike',
    location: 'London'
  },
  {
    first_name: 'Tim',
    location: 'US'
  },
  {
    first_name: 'John',
    location: 'Australia'
  }
];

const usersList = users.map(function (user) {
  return `${user.first_name} lives in ${user.location}`;
});

console.log(usersList); // ["Mike lives in London", "Tim lives in US", "John lives in
Australia"]
```

Here's a Code Pen Demo.

Note that in the above code, we're not changing the original `users` array. We're creating a new array with dynamic content because `map` always returns a new array.

Advantages of using the map method

- It helps quickly generate a new array without changing the original array
- It helps generate an array with dynamic content based on each element
- It allows us to quickly extract any element of the array
- It generates an array with the exact same length as the original array

Browser Support:

- All modern browsers and Internet Explorer (IE) version 9 and above
- Microsoft Edge version 12 and above

The Array.find Method

The `Array.find` method has the following syntax:

```
Array.find(callback(element[, index[, array]])[, thisArg])
```

> *The `find` method returns the `value` of the `first element` in the array that satisfies the provided test condition.*

The `find` method takes a callback function as the first argument and executes the callback function for every element of the array. Each array element value is passed as the first parameter to the callback function.

Suppose, we have a list of employees like this:

```
const employees = [
 { name: "David Carlson", age: 30 },
 { name: "John Cena", age: 34 },
 { name: "Mike Sheridan", age: 25 },
 { name: "John Carte", age: 50 }
];
```

and we want to get the record for the employee whose name is `John`. In this case, we can use the `find` method as shown below:

```
const employee = employees.find(function (employee) {
  return employee.name.indexOf('John') > -1;
});

console.log(employee); // { name: "John Cena", age: 34 }
```

Here's a Code Pen Demo.

Even though there is `"John Carte"` in the list, the `find` method will stop when it finds the first match. So it will not return the object with the name `"John Carte".`

The equivalent for loop code for the above example looks like this:

```
const employees = [
 { name: "David Carlson", age: 30 },
 { name: "John Cena", age: 34 },
 { name: "Mike Sheridan", age: 25 },
 { name: "John Carte", age: 50 }
];

let user;

for(let i = 0; i < employees.length; i++) {
  if(employees[i].name.indexOf('John') > -1) {
    user = employees[i];
    break;
  }
}

console.log(user); // { name: "John Cena", age: 34 }
```

Here's a Code Pen Demo.

As you can see, using normal for loop makes the code much larger and harder to understand. But using the `find` method, we can write the same code in an easy to understand way.

Advantages of using the find method

- It allows us to quickly find any element without writing a lot of code
- It stops looping as soon as it finds a match so there is no need for an extra break statement

Browser Support:

- All modern browsers except Internet Explorer (IE)
- Microsoft Edge version 12 and above

The Array.findIndex Method

The `Array.findIndex` method has the following syntax:

```
Array.findIndex(callback(element[, index[, array]])[, thisArg])
```

The `findIndex` method returns the **index** of the first element in the array **that satisfies the provided test condition**. Otherwise, it returns `-1`, indicating that no element passed the test.

```
const employees = [
  { name: 'David Carlson', age: 30 },
  { name: 'John Cena', age: 34 },
  { name: 'Mike Sheridan', age: 25 },
  { name: 'John Carte', age: 50 }
];

const index = employees.findIndex(function (employee) {
  return employee.name.indexOf('John') > -1;
});

console.log(index); // 1
```

Here's a Code Pen Demo.

Here we get the output as **1** which is the index of the first object with the name `John`. Note that the index starts with zero.

The equivalent for loop code for the above example looks like this:

```
const employees = [
  { name: 'David Carlson', age: 30 },
  { name: 'John Cena', age: 34 },
  { name: 'Mike Sheridan', age: 25 },
  { name: 'John Carte', age: 50 }
];

let index = -1;

for(let i = 0; i < employees.length; i++) {
  if(employees[i].name.indexOf('John') > -1) {
    index = i;
    break;
  }
}

console.log(index); // 1
```

Here's a Code Pen Demo.

Advantages of using the findIndex method

- It allows us to quickly find the index of an element without writing a lot of code
- It stops looping as soon as it finds a match so there is no need for an extra break statement
- We can find the index using the array `find` method also, but using `findIndex` makes it easy and avoids creating extra variables to store the index

Browser Support:

- All modern browsers except Internet Explorer (IE)
- Microsoft Edge version 12 and above

The Array.filter Method

The `Array.filter` method has the following syntax:

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

The `filter` method returns `a new array` with all the elements that satisfy the provided test condition.

The `filter` method takes a callback function as the first argument and executes the callback function for every element of the array. Each array element value is passed as the first parameter to the callback function.

```
const employees = [
  { name: 'David Carlson', age: 30 },
  { name: 'John Cena', age: 34 },
  { name: 'Mike Sheridan', age: 25 },
  { name: 'John Carte', age: 50 }
];

const employee = employees.filter(function (employee) {
  return employee.name.indexOf('John') > -1;
});

console.log(employee); // [ { name: "John Cena", age: 34 }, { name: "John Carte",
age: 50 }]
```

Here's a Code Pen Demo.

As can be seen in the above code, using `filter` helps to find all the elements from the array that match the specified test condition.

So using `filter` does not stop when it finds a particular match but keeps checking for other elements in the array that match the condition. Then it returns all the matching elements from the array.

> The main difference between `find` and `filter` is that `find` only returns the first matching element of the array, but using `filter` returns all the matching elements from the array.

Note that the `filter` method always returns an array. If no element passes the test condition, an empty array will be returned.

The equivalent for loop code for the above example looks like this:

```
const employees = [
  { name: 'David Carlson', age: 30 },
  { name: 'John Cena', age: 34 },
  { name: 'Mike Sheridan', age: 25 },
  { name: 'John Carte', age: 50 }
];

let filtered = [];

for(let i = 0; i < employees.length; i++) {
 if(employees[i].name.indexOf('John') > -1) {
   filtered.push(employees[i]);
 }
}

console.log(filtered); // [ { name: "John Cena", age: 34 }, { name: "John Carte",
age: 50 }]
```

Here's a Code Pen Demo.

Advantages of using the filter method

- It allows us to quickly find all the matching elements from the array
- It always returns an array even if there is no match, so it avoids writing extra `if` conditions
- It avoids the need of creating an extra variable to store the filtered elements

Browser Support:

- All modern browsers and Internet Explorer (IE) version 9 and above
- Microsoft Edge version 12 and above

The Array.every Method

The `Array.every` method has the following syntax:

```
Array.every(callback(element[, index[, array]])[, thisArg])
```

The `every` method tests whether all elements in the array pass the provided test conditions and returns a boolean `true` or `false` value.

Suppose we have an array of numbers and we want to check if every element of the array is a positive number. We can use the `every` method to achieve it.

```
let numbers = [10, -30, 20, 50];

let allPositive = numbers.every(function (number) {
  return number > 0;
});
console.log(allPositive); // false

numbers = [10, 30, 20, 50];

allPositive = numbers.every(function (number) {
  return number > 0;
});
console.log(allPositive); // true
```

Imagine you have a registration form, and you want to check if all of the required fields are entered or not before submitting the form. You can use the `every` method to check for each field value easily.

```
window.onload = function () {
  const form = document.getElementById('registration_form');
  form.addEventListener('submit', function (event) {
    event.preventDefault();
    const fields = ['first_name', 'last_name', 'email', 'city'];
    const allFieldsEntered = fields.every(function (fieldId) {
      return document.getElementById(fieldId).value.trim() !== '';
    });

    if (allFieldsEntered) {
      console.log('All the fields are entered');
      // All the field values are entered, submit the form
    } else {
      alert('Please, fill out all the field values.');
    }
  });
};
```

Here's a Code Pen Demo.

Here, inside the callback function of the `every` method, we're checking if each field value is not empty and returning a boolean value.

In the above code, the `every` method returns `true` if, for all the elements in the `fields` array, the callback function returns a `true` value.

If the callback function returns a `false` value for any of the elements in the `fields` array, then the `every` method will return `false` as the result.

Advantage of using the every method

> It allows us to quickly check if all the elements match certain criteria without writing a lot of code

Browser Support:

- All modern browsers and Internet Explorer (IE) version 9 and above
- Microsoft Edge version 12 and above

The Array.some Method

The `Array.some` method has the following syntax:

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

The `some` method tests whether at least one element in the array passes the test condition given by the provided function and returns a boolean `true` or `false` value.

It returns `true` once it finds the first match and returns `false` if there is no match.

Suppose we have an array of numbers and we want to check if the array contains at least one positive element. We can use the `some` method to achieve it.

```
let numbers = [-30, 40, 20, 50];

let containsPositive = numbers.some(function (number) {
  return number > 0;
});
console.log(containsPositive); // true

numbers = [-10, -30, -20, -50];

containsPositive = numbers.every(function (number) {
  return number > 0;
});
console.log(containsPositive); // false
```

There are some useful scenarios for using the `some` method.

`Some` method example 1:

Let's say we have a list of employees and we want to check if a particular employee is present in that array or not. We also want to get the index position of that employee if the employee is found.

So instead of using the `find` and `findIndex` methods separately, we can use the `some` method to do both of these.

```
const employees = [
  { name: 'David Carlson', age: 30 },
  { name: 'John Cena', age: 34 },
  { name: 'Mike Sheridon', age: 25 },
  { name: 'John Carte', age: 50 }
];

let indexValue = -1;
const employee = employees.some(function (employee, index) {
  const isFound = employee.name.indexOf('John') > -1;
  if (isFound) {
    indexValue = index;
  }
  return isFound;
});

console.log(employee, indexValue); // true 1
```

Here's a Code Pen Demo.

`Some` method example 2:

The array `forEach` , `map` , and `filter` methods run from start to finish until all of the elements of the array are processed. There is no way of stopping of breaking out of the loop, once a particular element is found.

In such cases, we can use the array `some` method. The `map` , `forEach` and `some` method takes the same parameters in the callback function:

- The first parameter is the actual value
- The second parameter is the index
- The third parameter is the original array

The `some` method stops looping through the array once it finds a particular match as can be seen in the above example 1.

Advantages of using the some method

- It allows us to quickly check if some of the elements match certain criteria without writing a lot of code
- It allows us to quickly break out of the loop, which was not possible with other looping methods seen above

Browser Support:

- All modern browsers and Internet Explorer (IE) version 9 and above
- Microsoft Edge version 12 and above

The Array.reduce Method

The `Array.reduce` method has the following syntax:

```
Array.reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])
```

The `reduce` method executes a **reducer** function (that you provide) on each element of the array, resulting in a single output value.

> Note that the output of the `reduce` method is always a single value. It can be an object, a number, a string, an array, and so on. It depends on what you want the output of `reduce` method to generate but it's always a single value.

Suppose that you want to find the sum of all the numbers in the array. You can use the `reduce` method for that.

```
const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduce(function(accumulator, number) {
  return accumulator + number;
}, 0);

console.log(sum); // 15
```

Here's a Code Pen Demo.

The `reduce` method accepts a callback function that receives `accumulator` , `number` , `index` and `array` as the values. In the above code, we're using only `accumulator` and `number` .

The `accumulator` will contain the `initialValue` to be used for the array. The `initialValue` decides the return type of the data returned by the `reduce` method.

The `number` is the second parameter to the callback function that will contain the array element during each iteration of the loop.

In the above code, we have provided `0` as the `initialValue` for the `accumulator` . So the first time the callback function executes, the `accumulator + number` will be `0 + 1 = 1` and we're returning back the value `1` .

The next time the callback function runs, `accumulator + number` will be `1 + 2 = 3` (`1` here is the previous value returned in the last iteration and `2` is the next element from the array).

Then, the next time the callback function runs, `accumulator + number` will be `3 + 3 = 6` (the first `3` here is the previous value returned in the last iteration and the next `3` is the next element from the array) and it will continue this way until all the elements in the `numbers` array are not iterated.

So the `accumulator` will retain the value of the last operation just like a static variable.

In the above code, `initialValue` of `0` is not required because all the elements of the array are integers.

So the below code will also work:

```
const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduce(function (accumulator, number) {
  return accumulator + number;
});

console.log(sum); // 15
```

Here's a Code Pen Demo.

Here, the `accumulator` will contain the first element of the array and `number` will contain the next element of the array (`1 + 2 = 3` during the first iteration and then `3 + 3 = 6` during the next iteration, and so on).

But it’s always good to specify the `initialValue` of `accumulator` as it makes it easy to understand the return type of the `reduce` method and get the correct type of data back.

Take a look at the below code:

```
const numbers = [1, 2, 3, 4, 5];

const doublesSum = numbers.reduce(function (accumulator, number) {
  return accumulator + number * 2;
}, 10);

console.log(doublesSum); // 40
```

Here's a Code Pen Demo.

Here, we’re multiplying each element of the array by 2. We have provided an `initialValue` of 10 to the `accumulator` so 10 will be added to the final result of the sum like this:

```
[1 * 2, 2 * 2, 3 * 2, 4 * 2, 5 * 2] = [2, 4, 6, 8, 10] = 30 + 10 = 40
```

Suppose, you have an array of objects with x and y coordinates and you want to get the sum of x coordinates. You can use the `reduce` method for that.

```
const coordinates = [
  { x: 1, y: 2 },
  { x: 2, y: 3 },
  { x: 3, y: 4 }
];

const sum = coordinates.reduce(function (accumulator, currentValue) {
    return accumulator + currentValue.x;
}, 0);

console.log(sum); // 6
```

Here's a Code Pen Demo.

Advantages of using the reduce method

- Using `reduce` allows us to generate any type of simple or complex data based on the array
- It remembers the previously returns data from the loop so helps us avoid creating a global variable to store the previous value

Browser Support:

- All modern browsers and Internet Explorer (IE) version 9 and above
- Microsoft Edge version 12 and above

Thanks for reading!

Want to learn all ES6+ features in detail including `let` and `const`, promises, various promise methods, array and object destructuring, arrow functions, async/await, import and export and a whole lot more?

Check out my Mastering Modern JavaScript book. This book covers all the pre-requisites for learning React and helps you to become better at JavaScript and React.

Also, check out my free Introduction to React Router course to learn React Router from scratch.

Want to stay up to date with regular content regarding JavaScript, React, Node.js? Follow me on LinkedIn.

Yogesh Chavan

Technical Writer | Freelancer and Full Stack Developer | Javascript | React | Nodejs.
https://dev.to/myogeshchavan97