

**Reed Barger**Posted on Jun 18, 2020 • Originally published at reedbarger.com on Jun 18, 2020

How to Transform JavaScript Objects - The Power of Object.Keys(), .Values(), .Entries()

#javascript #webdev #beginners #objects

How do we iterate over objects? How do we loop over all of the key-value pairs to get all of our data?

The way that we have been able to conventionally do so is through the for-in loop:

```
const obj = { a: 1, b: 2 };

for (const key in obj) {
  console.log("key", key);
  console.log("value", obj[key]);
}

// key a
// value 1
// key b
// value 2
```

This is one way to iterate over object data. But compared to arrays, we are limited in our ability to iterate over object data in different ways. The object doesn't have anywhere near as many methods that enable us to work with it in the way that we want.

In many cases, we may find it necessary to convert our object data into array data, particularly if we want to make use of all of the different array methods to transform our data in a more dynamic way.

With the help of some built-in Object methods we can convert objects into arrays in three different ways. These methods are `Object.keys`, `Object.values` and `Object.entries`.

Object.keys()

The first of these methods to arrive to the language was `Object.keys`. As its name indicates, it allows us to take the keys of an object and turns it into an array. Let's say we have some user data, expressed as an object:

```
const user = {  
  name: "John",  
  age: 29,  
};
```

When we use `Object.keys`, on our user object,

```
console.log(Object.keys(user)); // ["name", "age"]
```

We get an array, where `name` and `age` are elements. Note that since keys in objects are always strings, when they become array elements, they will be strings there as well.

Practical use - Object.keys()

But what's valuable about this?

One valuable thing is that we can actually our keys' names. Before we haven't had the ability to get access to the keys of objects themselves, just the properties. So this gives us a new ability to check whether a given property exists.

So now, using `Object.keys`, how would we check to see that our object includes a key with a certain name. In this case, maybe a key with the name `age`...

Well we know the array method to check and see if a given string exists as an element—the `includes` method. So we can use `Object.keys` and then chain on `includes`.

We can store the result of this operation in a variable, `ageExists` and log it:

```
const ageExists = Object.keys(user).includes("age");
console.log(ageExists);
```

And we get true. So that's a good way to see whether a given key exists on an object.

What if we want to get the values from the object as well? Well now that we can use any array method, we can map over key elements, and use property access with the original user object to get them.

So with map, every element can be called `prop` or `key`. And note that we have to use the square brackets syntax here instead of the dot property access. Why is that?

Because each key element has a different, dynamic value. Since dynamic values or variables only work with computed property names, we have to use square brackets:

```
Object.keys(user).map((key) => user[key]);
```

And then let's put the resulting array in a variable called `values` and see what we get:

```
const values = Object.keys(user).map((key) => user[key]);
console.log(values); // ["John", 29]
```

Object.values()

Now there is a much easier way to get the values of an Object. That's what we use `Object.values` for. We can replace all of the previous code we wrote with just `Object.values` and pass in the object whose property values we want to get:

```
// const values = Object.keys(user).map(key => user[key])
// console.log(values)

const values = Object.values(user);
console.log(values); // ["John", 29]
```

And we get the same result.

Practical use - Object.values()

What can we do with `object.values`? Let's say we had an object that contained a number of the user's monthly expenses:

```
const monthlyExpenses = {  
  food: 400,  
  
  rent: 1700,  
  insurance: 550,  
  internet: 49,  
  phone: 95,  
};
```

Try to imagine if this had even more properties than it does. How would we easily get a sum total of all of these combined expenses?

We could come up with some way of doing this using a for in loop or something, but it's not nearly as easy as throwing all of the values in a single array. And we know how to easily get the sum of an array of numbers using `reduce`.

See if you can do this on your own. How would you combine `Object.values` and `reduce` to get a monthly total of all John's expenses...

First we could create a variable, `monthlyTotal`, get the array of values, the cost of each expense. And then using the `reduce` method, we can sum everything in one line. First our return value will be as a number, so our initial value will be 0. Then we have our accumulator, and each element will be an expense. The shorthand to do this operation is to say `acc + expense`. And since we have an implicit return with our arrow function, the accumulator will always be returned:

```
const monthlyTotal = Object.values(monthlyExpenses).reduce(  
  (acc, expense) => acc + expense,  
  0  
);  
  
console.log(monthlyTotal); // 2794
```

As a result, our monthly total is 2794. This is probably as concise as such an operation can get. So `Object.values` is great for when you need a more flexible way to work with all the values of a given object.

Practical use - Object.entries()

And lastly, if we need to the entire object, both keys and values, mapped to a single array, we use `Object.entries`:

```
console.log(Object.entries(user)); // (2) [Array(2), Array(2)]
```

This gives us an interesting result—an array of arrays. So what use is having both keys and values?

Think about if we had a far more complex object, maybe a nested one with a bunch of user data, where each key is equal to the users' id. In fact, I can guarantee that in the future, when you work with fetching external data, you will get data returned to you that looks like this.

```
const users = {  
  "1": {  
    name: "John",  
    age: 29,  
  },  
  "2": {  
    name: "Jane",  
    age: 42,  
  },  
  "3": {  
    name: "Fred",  
    age: 17,  
  },  
};
```

And let's say that we need to get very specific data from this object, say we just wanted to get the user data, both their name, age, and id, but specifically for users over 20.

If we had only to rely on object's methods, we would have had no way of getting this data from a nested object.

But with `Object.entries`, since it converts our object into an array, we can solve this problem, no problem.

First let's pass `users` to `Object.entries` and `console.log` it to see what we get, since we're going to get twice as much data as either `Object.keys` or `Object.values`:

```
console.log(Object.entries(users));  
  
// 0: (2) ["1", {...}]  
// 1: (2) ["2", {...}]  
// 2: (2) ["3", {...}]
```

For each nested array, we have the key, or the id of the user as the first element and the value, or the user object as the second element.

Replacing map / filter with a single reduce

Now we can chain on any array method we need to get the job done. So let's think about what we want to do: we need to transform the data into an array of object, which we also want to filter based on a condition. So based of these two criteria, think for a second and take a guess about what array method we need to use...

You might be saying that we need to use two methods, `map` and `filter`, because we want to both transform and filter the array. So one approach would be to chain on `map` and then `filter`. However, let be give you a quick tip for doing a `map` and `filter` transformation. Based off of our deep dive into both methods, what do we know about both of them? That they can both be implemented with `reduce`. So since they are both reduction operations, we can replace both of them with a single `reduce`. In fact, most times you think you need to do array method chaining, you can replace them with `reduce`.

So using `reduce`, we'll create the basic structure. We'll create the callback function with the accumulator and then provide the initial value, based off of the final value we want to get, an array:

```
Object.entries(users).reduce((acc) => {}, []);
```

And then for the current element, we can again use array destructuring to get the first and second elements. The first, the key, will be called `id`, and then it's value, the user data, will be `user`.

```
Object.entries(users).reduce((acc, [id, user]) => {}, []);
```

And now to conditionally put the array element in our final array if it's age property is greater than 20, we can add an `if` statement. If `user.age > 20`, then push an object onto the accumulator. To make this object, we can spread in the object's properties and add on the `id` at the end. Since `id` will be used at the property and value, we'll use the object shorthand:

```
Object.entries(users).reduce((acc, [id, user]) => {  
  if (user.age > 20) {  
    acc.push({ ...user, id });  
  }  
}, []);
```

And finally, we just need to return the accumulator at the end. Let's just put the result

array in a new variable called `usersOver20` and log it.

```
const usersOver20 = Object.entries(users).reduce((acc, [id, user]) => {  
  if (user.age > 20) {  
    acc.push({ ...user, id });  
  }  
  return acc;  
}, []);  
console.log(usersOver20);
```

Now we have a cleaner data structure in this form, where all of a user's data is in one single object. Note that in most JavaScript frameworks, it's ideal to have sets of data such as these, consisting of arrays with objects as their elements, particularly to iterate over for displaying their content to our users in the browser.

Summary

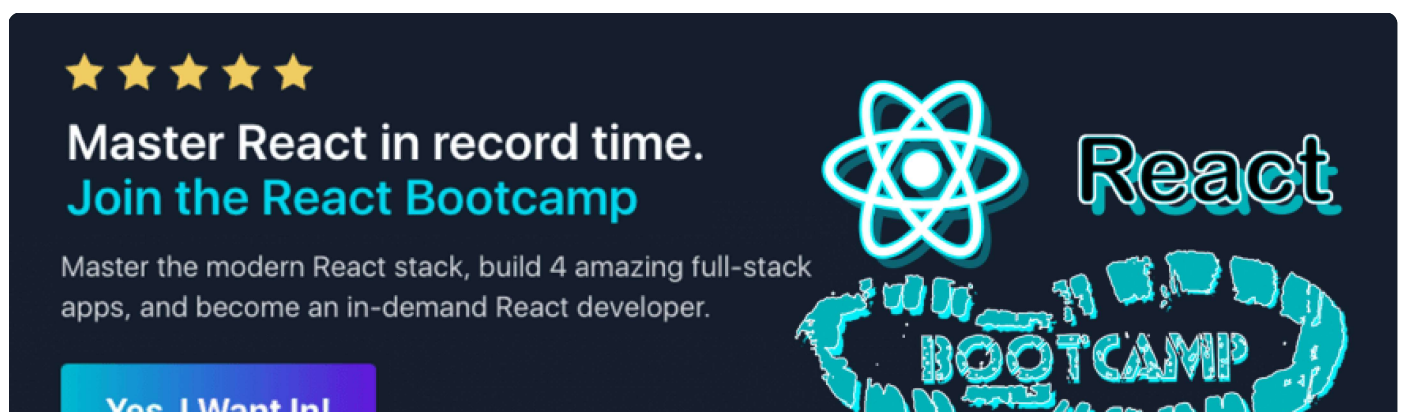
When you are in a situation where you have a more static data structure like an object, but you need to manipulate it's contents in a more dynamic way, look into to using one of the methods we've covered here, either `Object.keys`, `values` or `entries`.

You'll get access to everything on the object, both it's keys and values if you need, plus have the ability to chain on any additional array methods to get the job done (or replace them with a single `reduce`, as we saw in this example) and transform and filter the data as you need.

Enjoy this post? Join The React Bootcamp

[The React Bootcamp](#) takes everything you should know about learning React and bundles it into one comprehensive package, including videos, cheatsheets, plus special bonuses.

Gain the insider information hundreds of developers have already used to master React, find their dream jobs, and take control of their future:



The advertisement banner features a dark blue background. On the left, there are five yellow stars at the top, followed by the text 'Master React in record time. Join the React Bootcamp' in white and light blue. Below this, it says 'Master the modern React stack, build 4 amazing full-stack apps, and become an in-demand React developer.' At the bottom left is a blue button with the text 'Yes, I Want In!'. On the right side, there is the React logo (a glowing blue atom) and the word 'React' in a stylized font. Below the logo is a circular graphic with the word 'BOOTCAMP' in the center, surrounded by various icons representing different aspects of development.

Click here to be notified when it opens

Discussion (1)



Angelo&Angelica • Jun 18 '20



Bellini verament

[Code of Conduct](#) • [Report abuse](#)



Reed Barger

Full-stack React developer, sharing cheatsheets, articles, and premium courses to help you get ahead @ ReactBootcamp.com

JOINED

Aug 13, 2019

More from Reed Barger

React Context for Beginners – The Complete Guide (2021)

[#react](#) [#javascript](#) [#beginners](#) [#tutorial](#)

13 Amazing React Libraries You Should Try in 2021

[#react](#) [#javascript](#) [#tutorial](#) [#beginners](#)

Build an Amazing Portfolio Website with React

[#react](#) [#javascript](#) [#tutorial](#) [#beginners](#)