Tip 22: Create Arrays of a Similar Size with map()

In this tip, you'll learn how to pull out a subset of information from an array using map().

We'll cover the following

In the previous tip, you saw how you could rewrite a simple `for` loop with an array method. Now you're going to start exploring how to use specific array methods.

Using `map()`

You'll begin with `map()` (not to be confused with the `Map` object). It's fairly common, and your new array receives the information you return in a brand new array. In other words, the return value is transparent, which isn't the case with other array methods. Here's another good reason to start with `map()`: the name "map" isn't very expressive. *What does that even mean?* When I first learned it, I needed a fair amount of experience before I could see and understand a map function at a glance.

Your goal is to get an idea of how most array methods work. And your secondary goal is to gain enough experience with `map()` that you'll start to see why it's one of the most popular methods.

Example

Start with a simple `map` function. A `map` function takes a piece of information from an input array and returns something new. Sometimes it returns part of the

information. Other times, it transforms the information and returns the new value. That means it can take a single property from an array, or it can take a value in an array and return an altered version. For example, it can return an array with all the values capitalized or converted from integers to currency.

The easiest example is pulling specific information from an object. Let's start with a collection of musicians.

```
const band = [
    {
        name: 'corbett',
        instrument: 'guitar',
    },
    {
        name: 'evan',
        instrument: 'guitar',
    },
    {
        name: 'sean',
        instrument: 'bass',
    },
    {
        name: 'brett',
        instrument: 'drums',
    },
];
```

You have the `band`, but what you really want is just a list of instruments the band members play.

Every array method takes a callback function that you'll apply to each member of the array. These functions are very simple by design. They can only take one argument: *the individual member of an array* (the `reduce()` method is an exception that we'll discuss later).

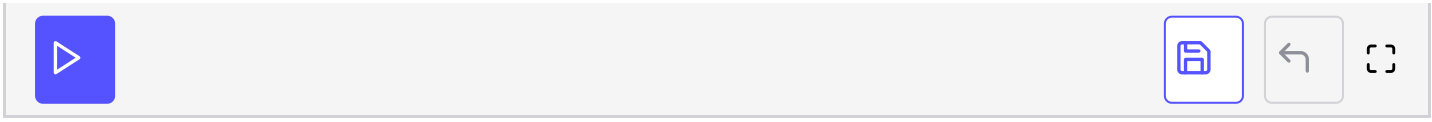
A simple `for` loop

Before you dive into building a map function, create a basic `for` loop to use as a comparison. Once you have that loop, you'll start slowly refactoring it until you get to a working map function. This will help you gain an understanding for how a `map()` function is just a simplified loop.

Okay, here's a simple for loop to get the band instruments:

```
const band = [
    {
        name: 'corbett',
        instrument: 'guitar',
```

```
    },
    {
        name: 'evan',

        instrument: 'guitar',
    },
    {
        name: 'sean',
        instrument: 'bass',
    },
    {
        name: 'brett',
        instrument: 'drums',
    },
];

const instruments = [];

for (let i = 0; i < band.length; i++) {
    const instrument = band[i].instrument;
    instruments.push(instrument);
}
console.log(instruments);
```



Refactoring a `for` loop to a `map()` method

Now it's time to start refactoring. The first thing to do is to combine **line 3** and **line 4**. Instead of getting the `instrument` and passing it to the `push` method, you'll get the `instrument` as part of the argument for `push()`. To keep things readable, put the logic to retrieve the instrument into a separate function.

You'll get a function that looks like this:

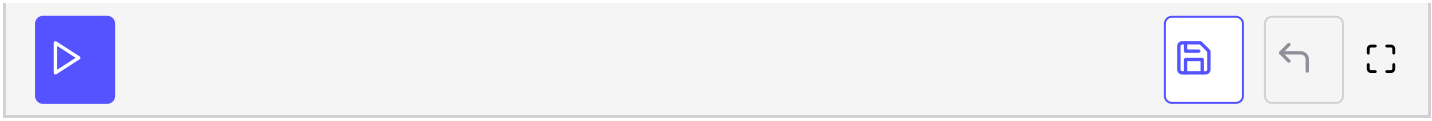
```
function getInstrument(member) {
    return member.instrument;
}
```

Sure it doesn't shorten things up much, but it helps. And more important, you've made a huge step by separating the iterator `band[i]` and the information you want from the individual member: `member.instrument`. Remember that with `map()` methods, you want to think about the individual pieces, not the whole array.

Here's how your new method fits into the current for loop:

```
const band = [
    {
        name: 'corbett',
        instrument: 'guitar',
    },
```

```
    {
        name: 'evan',
        instrument: 'guitar',

    },
    {
        name: 'sean',
        instrument: 'bass',
    },
    {
        name: 'brett',
        instrument: 'drums',
    },
];

const instruments = [];

function getInstrument(member) {
    return member.instrument;
}

for (let i = 0; i < band.length; i++) {
    instruments.push(getInstrument(band[i]));
}

console.log(instruments);
```



At this point, you've pretty much written your map function.

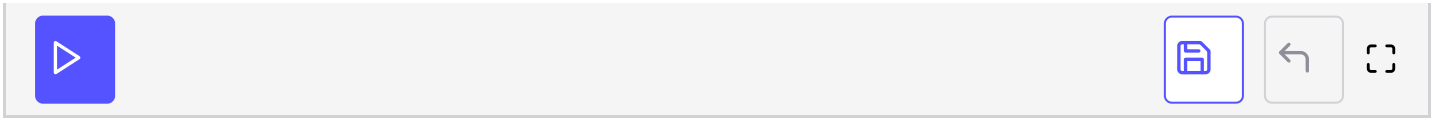
With `map()`, there's no need to set up a return array—that's included as part of the array method. There's also no need to push information. `map()` pushes the result of the function into its own return array.

The only thing you need for `map()` is a function that takes each item as an argument and returns something to put in the return array. What do you know—you already have that written out!

Most of the time, you'll just write an anonymous function for an array method, but that's not a requirement. You can name a function if you want (and sometimes that's a smart move for testing purposes). That means you can reuse the `getInstrument()` function you already have and pass it directly to `map()`. At this point, you can abandon your `for` loop.

```
const band = [
    {
        name: 'corbett',
        instrument: 'guitar',
    },
    {
        name: 'evan',
```

```
        instrument: 'guitar',
    },
    {
        name: 'sean',
        instrument: 'bass',
    },
    {
        name: 'brett',
        instrument: 'drums',
    },
];


function getInstrument(member) {
    return member.instrument;
}

const instruments = band.map(getInstrument);
console.log(instruments);
```



Advantages of the `map()` method

Look at what you've accomplished. You removed excess code while keeping things more transparent:

- You know you're going to get an array. You don't need to define one ahead of time.
- You know it will be the same size as the original array.
- You know it will contain the instruments and nothing else.

Predictable and simple.

If you understand this, congratulations—you understand most array methods. All array methods are just methods that take a callback that act on each member of an array. The type of array method determines what happens with the return value of that function. But writing the function itself is very similar for each array method.

Converting `map()` to an anonymous function

Now that you've refactored your `for` loop to a `map` method, you can take the next step and convert the named function to an *anonymous* function. Remember those arrow functions you just learned? Now is a perfect time to use them.

You're taking a single argument, so you don't need parentheses. And the body of

the function is only one line long, so you don't need curly braces or a return

statement. Go ahead and try writing it out. These functions become much easier with practice.

```
const band = [
    {
        name: 'corbett',
        instrument: 'guitar',
    },
    {
        name: 'evan',
        instrument: 'guitar',
    },
    {
        name: 'sean',
        instrument: 'bass',
    },
    {
        name: 'brett',
        instrument: 'drums',
    },
];


const instruments = band.map(member => member.instrument);
console.log(instruments);
```

`map()` is fairly simple, but it's flexible. You can use it for anything—yes, anything—when the goal is to have an array of the same size. Up to now, you've only been elevating data from an array of objects. But you can also transform information, as you saw when you converted strings to values with `parseInt()` in the previous tip.

What does the following program do?

```
const arr = [{name: 'Bryan', age: '21'}, {name: 'Allen', age: '2
2'}]
const ans = arr.map(obj => {
      obj.name = obj.name + 'a'
      obj.hobby = 'sports'
      return obj;
})

console.log(ans)
```

Retake Quiz

In the next tip, you'll see an array method that does something a little different. You'll maintain the shape of the array items, but you'll return only a subset by performing a true or false check on each item.