Tip 21: Write Shorter Loops with Array Methods

In this tip, you'll learn to reduce long loops to single lines with array methods.

We'll cover the following

Using loops

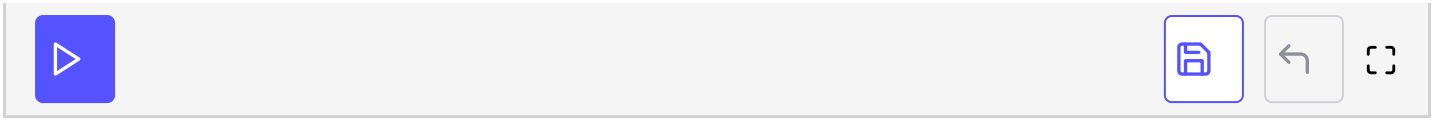
Before we begin, I want you to know that `for` loops and `for...of` loops are good. You'll want to use them, and you should use them. They're never going away.

But you should use them less. Why? Well, the reason is simple: *They're unnecessary clutter*. You're writing modern JavaScript, which means you're going for simplicity, readability, and predictability, and traditional loops don't meet those goals. Array methods do. Mastering array methods is the fastest way to improve your JavaScript code.

Example

Look at this basic loop that converts an array of `price` strings into floating-point values.

```
const prices = ['1.0', '2.15'];
const formattedPrices = [];
for (let i = 0; i < prices.length; i++) {
    formattedPrices.push(parseFloat(prices[i]));
}
console.log(formattedPrices);
```

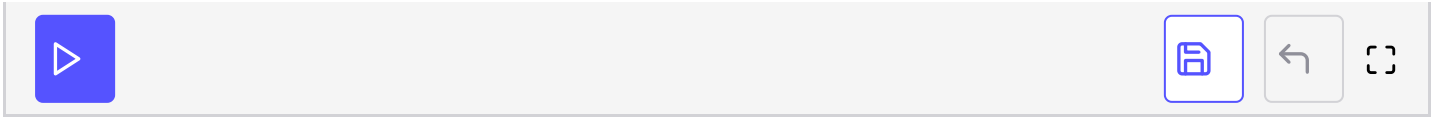


It's short, sure. The whole thing takes four lines. Not bad. But consider a file that has five functions. And each function has a loop. You just added 20 lines to the file.

And this function is pretty short to begin with. What if you had an array that contained some non-numerical strings and you wanted only parsable numbers? Your function starts to grow.

```
const prices = ['1.0', 'negotiable', '2.15'];

const formattedPrices = [];
for (let i = 0; i < prices.length; i++) {
    const price = parseFloat(prices[i]);
    if (price) {
        formattedPrices.push(price);
    }
}
console.log(formattedPrices);
```



The problem with using loops

Clutter. Clutter. Clutter. Just look at all the extra stuff you need just to get the float prices. On **line 3**, you declare a new collection before you even begin working with the data. And before you even enter the `for` loop, you're facing a paradox. Because `let` is block-scoped, you have to declare the collection outside of the loop. Now you have an extraneous array with no members sitting outside the loop.

Next, you have to follow the crazy three-part pattern to declare the iterator on **line 4**. You can often get around that with a `for...of` loop, but many times, you still need to declare the iterator.

And finally, you're mixing two different concerns: *transforming the value and filtering out bad values with your conditional* on **line 6**. This isn't a horrible problem, but it does hurt predictability. Is the code standardizing the values in the array or is it filtering out unwanted values? In this case, it does both.

Is it simple? No. It takes multiple lines and several variable declarations.

Is it readable? Sure. But as the number of lines grows, the file decreases in readability.

Is it predictable? No. It creates an empty array that may or may not be changed later because it's not a constant (and yes, you can push to a variable defined with `const`, but that's a no-no). Besides, we can't predict at a glance if `formattedPrices` will include everything. In this case, it won't. In the first case, it will. There's no clue inherent in the action.

Using array methods

Array methods are a great way to get clean predictable code with no extraneous data. Some find them intimidating, but with a little effort, you'll master them quickly and find your code better than ever.

The most popular array methods change either the *size array or the shape of the data* in the array. There's one big exception: the `reduce()` method. But you'll get to that soon enough.

What does size and shape even mean? Look at a simple array of members of a digital marketing team.

```
const team = [
    {
        name: 'melinda',
        position: 'ux designer'
    },
    {
        name: 'katie',
        position: 'strategist'
    },
    {
        name: 'madhavi',
        position: 'developer'
    },
    {
        name: 'justin',
        position: 'manager'
    },
    {
        name: 'chris',
        position: 'developer'
    }
]
```

At a glance, you can clearly see that this array has a size of five objects. You can also see that every item has a shape: a `name` and a `position`.

Nearly any array method you choose will alter either the size or the shape of the

return array. You simply need to decide if you want to change the size or the shape (or both).
Do you want to change the size by reducing the number of members in the array by removing them? Do you want to change the shape by only getting the names of the team members? Do you want to do both and get the names of just the developers?

Cheatsheet

You'll explore all these in upcoming tips, but here's a cheat sheet:

	Action	Example	Result
`map()`	Changes the shape, but not the size.	Get the `name` of everyone on the team.	*['melinda', 'katie', 'madhavi', 'justin', 'chris']*
`sort()`	Changes neither the size nor the shape, but changes the order.	Get the team members in alphabetical order.	*[{name: 'chris', position: 'developer'}, {name: 'justin' ...}]*
`filter()`	Changes the size, but not the shape.	Get the developers.	*[{name: 'madhavi', position: 'developer'}, {name: 'chris', position:'developer'}]*
`find()`	Changes the size to exactly one, but not the shape. Does not return an array.	Get the `manager`.	*{name: 'justin', position: 'manager'}*
`forEach()`	Uses the shape	Give all	*Melinda gets a*

`forEach()`	Uses the shape, but returns nothing.	Give all members a bonus!	*Melinda gets a bonus! Katie get a bonus!... (but no return value).*
`reduce()`	Changes the size and the shape to pretty much anything you want.	Get the total developers and non-developers.	{*developers: 2, non-developers: 3*}

Example

Now that you can see all the options ahead, I'm sure you're excited to get started. Well, you don't have to wait any longer! Here's how you can rewrite the `for` loop using an array method.

```
const prices = ['1.0', '2.15'];
const formattedPrices = prices.map(price => parseFloat(price));
console.log(formattedPrices)
```

Evaluating the code

Try evaluating the code using the usual criteria: *simple, readable, predictable*.

Is it simple? **Yes**. Everything fits on a single line.

Is it readable? **Yes**. You can see the action on one line. As the file grows, the number of lines won't grow any faster than the number of simple actions.

Is it predictable? **Yes**. The value is assigned with `const` so you know it won't change. And because you used a *map*, you know that you'll have an array of exactly the same size. At a glance, you can tell that the goal is to get the float values with `parseFloat()`. So you also know that the array will be the exact size as the original with only float values.

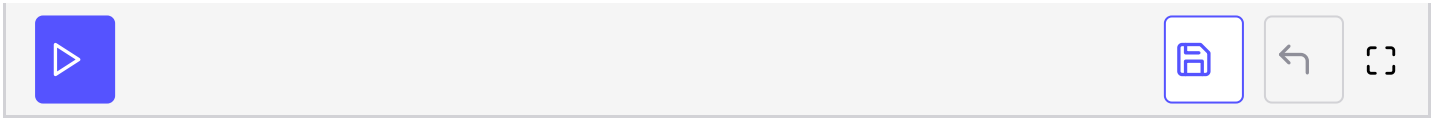
I know what you're going to say: You may have solved the simple loop, but what

about the more complicated loop that removes false values?

Chaining array methods

Array methods can be chained. That means you can call one right after the other and it will pass the result down to the next item.

```
const prices = ['1.0', 'negotiable', '2.15'];
const formattedPrices = prices.map(price => parseFloat(price)) // [1.0, NaN, 2.15]
.filter(price => price);
console.log(formattedPrices);
```



First you convert values to floating point while keeping the array the same size. Then you change the size, but not shape, by pulling out false values.

The process is much easier to follow, and because you can chain them together, you can still assign the resulting array with `const`.

Do I have your interest now? Awesome. Because it's time to dive into writing your own array methods. You'll find you'll use the same methods over and over. But remember—they each have their strengths. If you become frustrated trying to fit an action into a method, try using a different method or think about how you can break the action into pieces and chain it together.

Study the program below:

```
const double = (x) => {
  return x*x;
}
const values = [2, 4, 6]
for (let i = 0; i < values.length; i++) {
    double(values[i])
}
```

Which lines of the code below are equivalent to the for loop in the example above?

Retake Quiz

Now that you have the foundation, it's time to jump in. In the next tip, you'll start changing the shape of members of an array with `map()`.