Tip 27: Reduce Loop Clutter with for...in and for...each

In this tip, you'll learn how to maintain clarity with loops over iterables using for...in and over objects using for...of.

We'll cover the following

Using array methods

Hopefully by now you're convinced that array methods can handle most of your iterations in clear and predictable ways. Sometimes, however, an array method may be either inefficient or cumbersome.

There may be times you want to exit out of a loop when a result doesn't match what you need. In those cases, it makes no sense to keep iterating over information.

Alternatively, an array method may be overly complex when you're working with a collection that isn't an array. Remember that just because a structure isn't an array doesn't mean you can't use array methods. If you're working with an object, you can use `Object.keys()` to create an array of keys and then perform whatever method you want. Or you can use the spread operator to convert a `Map` to an array of pairs. If you need a refresher, head back to Tip 14, Iterate Over Key-Value Data with Map and the Spread Operator.

In fact, those are great approaches. The popular Airbnb style guide, for example, insists that you always use array methods and restricts the use of the `for...of` and `for...in` loops.

That opinion isn't shared by all. Sometimes it's not worth the hassle to convert structures to arrays and it's worth knowing other options.

Example

Consider an application where you can select and compare multiple sets of information. Perhaps you're building an application that has a list of consulting firms. A user can select multiple firms and compare and contrast services.

Knowing what you know now, you'd probably use a `Map` to hold the various firms as users click on options. After all, you're constantly adding and deleting information, which is an action a `Map` can handle easily.

As the user clicks on firms they're interested in, you could add the firms to a simple map that uses the ID of the firm as a key and the name of the firm as the value.

```
const firms = new Map()
    .set(10, 'Ivie Group')
    .set(23, 'Soundscaping Source')
    .set(31, 'Big 6');

console.log(firms);
```

You can do a lot with that small amount of information. You could select details from a database. You could check availability or create a comparison chart. In all cases, you'd need to act on the collection one piece at a time.

Looping using `for`

For this example, loop through the firms a user has selected to check and see if they're available. (*For the purposes of this example, you'll use a generic* `isAvailable()` *function that would be defined elsewhere.*) If one isn't available, return a message saying the firm is unavailable. Otherwise, return a message saying all are available.

If you try writing this out, you'll immediately notice a problem. You can't use a traditional `for` loop because the collection isn't an array. You can easily bypass that problem by converting the map to an array with the *spread operator* before looping.

```
const firms = new Map()
    .set(10, 'Ivie Group')
    .set(23, 'Soundscaping Source')
    .set(31, 'Big 6');

function checkConflicts(firms, isAvailable) {
    const entries = [...firms];
    for (let i = 0; i < entries.length; i++) {
        const [id, name] = entries[i];
        if (!isAvailable(id)) {
            return `${name} is not available`;
        }
    }
    return 'All firms are available';
}

const soundscapeUnavailable = id => id !== 23;
console.log(checkConflicts(firms,soundscapeUnavailable));
```

That loop is pretty straightforward. It gets the information you need in a fairly transparent way. By now, though, you know there are better ways to loop. And you probably noticed that because you have to convert to an array, you might as well use an array method.

Looping using array methods

But there's no good array method to perform the action. Sure, there are plenty of options that you could try. You might use `find()` to see if there's a firm that's unavailable.

```
const firms = new Map()
    .set(10, 'Ivie Group')
    .set(23, 'Soundscaping Source')
    .set(31, 'Big 6');


function findConflicts(firms, isAvailable) {
    const unavailable = [...firms].find(firm => {
        const [id] = firm;
        return !isAvailable(id);
    });

    if (unavailable) {

        return `${unavailable[1]} is not available`;
    }

    return 'All firms are available';
}

const soundscapeUnavailable = id => id !== 23;
console.log(findConflicts(firms,soundscapeUnavailable));
```

You might also write a `reduce()` method that returns a string with the success message as a default.

```
const firms = new Map()
  .set(10, 'Ivie Group')
  .set(23, 'Soundscaping Source')
  .set(31, 'Big 6');


function checkConflicts(firms, isAvailable) {
  const message = [...firms].reduce((availability, firm) => {
    const [id, name] = firm;
    if (!isAvailable(id)) {
      return `${name} is not available`;
    }
    return availability;
  }, 'All firms are available');
  return message;
}

const soundscapeUnavailable = id => id !== 23;
console.log(checkConflicts(firms,soundscapeUnavailable));
```

Issues with using array methods

There are many ways to solve the problem. Maybe those solutions are fine for you and your team. Still, they're a little clunky. You'd probably have to read them twice to understand what's happening.

The problem is the `find()` approach is a *two-step* process *(find if there are unavailable firms, and then build a message*), and the `reduce()` approach is a little difficult to understand.

There's also the problem that the `find()` function will give you only the first unavailable firm and the `reduce()` function will give you only the last.

To be fair, you won't solve that problem here. Try to find a solution both with array methods and with other loops.

Hide Hint

Chain `filter()` and `map()` to make an array of messages.

For now, however, ignore that optimization and focus instead on whether all are available or not.

You've seen three ways to solve the exact same problem with the same result. They all share a common feature: *They all require you to first convert the map to an array*. Turns out that's not even necessary. The property on the `Map` that lets you use the *spread operator*, the `MapIterator`, is the same property that will let you iterate over a map directly.

In the tip on using the spread operator with `Map`, you learned about the `MapIterator`. It's just a specific instance of the more generalized `Iterator`, which designates a specific type of object that can access pieces one at a time. You can find them on *maps, arrays, and sets*, and you can even make your own, as you'll see in Tip 41, Create Iterable Properties with Generators.

Using a `for...of` loop

Most important, you can use the iterator with a special loop called a `for...of` loop. This loop is very similar to the `for` loop except that you don't iterate over the *indexes* (that `let i = 0` part). Instead, you loop directly over the members of the collection.

In the loop parameters, you declare a name for the individual item and then use that in the body.

Instead of converting a specialized object to an array, you use the exact same idea of a `for` loop while removing reference to indexes. You effectively use the callback method from an array method.

Example

Here’s a translation of the functions you saw earlier.

```
const firms = new Map()
    .set(10, 'Ivie Group')
    .set(23, 'Soundscaping Source')
    .set(31, 'Big 6');

function checkConflicts(firms, isAvailable) {
    for (const firm of firms) {
        const [id, name] = firm;
        if (!isAvailable(id)) {
            return `${name} is not available`;
        }
    }
    return 'All firms are available';
}

const soundscapeUnavailable = id => id !== 23;
console.log(checkConflicts(firms,soundscapeUnavailable));
```

Advantages of using a `for...of`

Notice a few things: First, you declare the variable, firm with `const`. Because `const` is block scoped, this variable won’t exist outside the loop so you don’t have to worry about polluting the rest of the code. Next, using the same ideas from array methods, you act on the item directly. You don’t need to reference the full collection as you do with *entries[i]* in the `for` loop. It’s a combination of array *callback* methods and `for` loops.

As a bonus, you also gain a slight optimization by not converting an iterable to an array before then iterating over it again. You don’t have to avoid array methods to gain that micro-optimization, but it’s something to consider.

What are the trade-offs? The most obvious is that because the loop can do anything, you lose some predictability. Honestly, that’s about the only problem as long as you don’t mutate the collection as you loop through it (which you could easily do). But you can mutate collections with array methods, too. Avoiding side effects and mutations requires discipline more than syntax.

To loop or not?

With all those advantages you may wonder, should you always loop directly? In short, no. As a rule, use array methods when they're clear fits and you prefer them

as the default. When you're filtering data in a map, for example, you should use filters. When you're converting a map to an array of values, use the `map()` method. Otherwise, you'll be stuck creating a container array and mutating it on each loop. Use `for...of` when it makes the most sense.

There's another slight complication—or benefit, depending on how you think about it—to `for...of` loops. There's a similar but different loop that only works on *key-value* objects. It's called the `for...in` loop.

The `for...in` loop is very similar to the `for...of` loop. You don't need to convert an object's keys to an array with `Object.keys()` because you operate directly on the object itself. Specifically, you loop over the properties of an object.

Using a `for...in` loop

If you've worked with JavaScript objects in-depth, you'll likely know there are some complications with object properties because they can be inherited from other objects in a prototype chain. In addition, objects have non-enumerable properties that are also skipped during iteration.

In short, properties on objects can be complex. You can read more about it on the Mozilla Developer Network.

Most times, though, you're working with simple things, and that's what you'll focus on here. To start off, convert your map of firms to an object. It's almost identical, but because keys have to be strings, you'll need to convert them from numbers. In reality, you can use numbers as keys in object literal syntax and they'll be covertly converted to strings, but that's a problem with objects, not an advantage.

```
const firms = {
    '10': 'Ivie Group',
    '23': 'Soundscaping Source',
    '31': 'Big 6',
};
```

When using a `for...in` loop, you'll get each property one at a time. Unlike the `for...of` loop, you don't get the values, and you'll have to reference the full collection using the key on each iteration. Everything else should be familiar. You name the variable, preferably with `const`, and then you use that in the body, knowing it will change on each iteration.

> Try and convert the last `for...of` loop and see what you get.

You probably came up with something like this:

```
const firms = {
    '10': 'Ivie Group',
    '23': 'Soundscaping Source',
    '31': 'Big 6',
};

function checkConflicts(firms, isAvailable) {
    for (const id in firms) {
        if (!isAvailable(parseInt(id, 10))) {
            return `${firms[id]} is not available`;
        }
    }
    return 'All firms are available';
}

const alwaysTrue = id => id;
const soundscapeUnavailable = id => id !== 23;
console.log(checkConflicts(firms,alwaysTrue));
console.log(checkConflicts(firms,soundscapeUnavailable));
```

Because you're getting the property and not a pair, you don't need to extract the name and value separately. Any time you need the value, you can grab it using array notation on the individual item. If you need the key to be an integer, which you do in this case, you'll need to convert it using `parseInt()`. This is why the subtle conversions that happen with object keys can be so confusing.

As with the `for...of` loop, use the `for...in` loop when it makes sense, but try not to use it as the default. If you're only going to use the keys, it may make more sense to pull them out with `Object.keys()` before using an array method. The same is true if you just plan on using the values. You can use `Object.values()` to convert those to an array, though that's less common.

One other precaution: *Don't mutate the object as you loop over it. That can be very dangerous, and bugs can creep in quickly, especially if you add or modify properties other than the property currently being iterated.*

Now you have a whole new set of tools for iterating over collections. As you saw in this example, you can solve most problems with multiple methods, so it often

comes down to a matter of personal and team preference. Over time, you’ll find that you prefer some methods over the others, and that’s fine. There are fewer right and wrong answers in development than people think.

In the next chapter, you’ll be moving from the nuts and bolts of working with data into composing functions. You’ll start with simply exploring new ways of working with parameters. And yes, there are enough changes that we’ll need a whole chapter to explore them. JavaScript can do a lot of interesting things with functions, so it’s exciting that even simple parameters are now more flexible. This is where the fun really begins.