Tip 25: Combine Methods with Chaining

In this tip, you'll learn to perform multiple array methods with chaining.

We'll cover the following

Chaining

Chaining is an old concept in programming. You can find it in many object-oriented languages. Like a lot of programming concepts, it actually sounds more complicated than it is in practice.

> Here's a quick definition: **Chaining** is immediately calling a method on a *returned* object (which in some cases is the original object) *without* reassigning the value first.

Okay, now forget that definition. For our purposes, chaining means that you can call *several* array methods in a row (*as long as you get an array back*). It's a convenient way to perform several actions in a very clear manner.

Example

Think back to the last example: sending notifications to club members. The example was simplified (*as examples always are*). An actual array of club members would have a lot more data. It would have *member status, email addresses, mailing addresses, position, and so on*.

addresses, position, and so on.

To keep things simple, let’s add just two fields: `active` and `email`.

```
const sailors = [
    {
        name: 'yi hong',
        active: true,
        email: 'yh@yhproductions.io',
    },
    {
        name: 'alex',
        active: true,
        email: '',
    },
    {
        name: 'nathan',
        active: false,
        email: '',
    },
];
```

There’s not much more information, but you have enough that you can be more sophisticated about whom you email.

Step 1: `filter()`

First, you can *filter* out all the inactive members—they won’t want an invitation.

```
const sailors = [
    {
        name: 'yi hong',
        active: true,
        email: 'yh@yhproductions.io',
    },
    {
        name: 'alex',
        active: true,
        email: '',
    },
    {
        name: 'nathan',
        active: false,
        email: '',
    },
];

const active = sailors.filter(sailor => sailor.active);
console.log(active);
```

Step 2: `map()`

Next, you can normalize the email addresses. If members have an email set, use that. Otherwise, use their default club email address.

```
const sailors = [
    {
        name: 'yi hong',
        active: true,
        email: 'yh@yhproductions.io',
    },
    {
        name: 'alex',
        active: true,
        email: '',
    },
    {
        name: 'nathan',
        active: false,
        email: '',
    },
];

const active = sailors.filter(sailor => sailor.active);
const emails = active.map(member => member.email || `${member.name}@wiscsail.io`);
console.log(emails);
```

Step 3: `forEach()`

Finally, after the inactive members are removed and the email addresses are normalized, you can call `sendInvitation()` with the correct member information.

```
const sailors = [
    {
        name: 'yi hong',
        active: true,
        email: 'yh@yhproductions.io',
    },
    {
        name: 'alex',
        active: true,
        email: '',
    },
    {
        name: 'nathan',
        active: false,
        email: '',
    },
];

const active = sailors.filter(sailor => sailor.active);
const emails = active.map(member => member.email || `${member.name}@wiscsail.io`);
emails.forEach(sailor => sendEmail(sailor));
```

Notice that you assigned the result to a variable each time. With chaining, that's *not* necessary. Instead, you can *remove* the intermediate step of assigning to a variable by calling a method directly on the result.

Because `filter()` always returns an array (*even if it's an empty array*), you know that you can call any other array method on it. Similarly, because `map()` always returns an array, you can call another array method on it. Crucially, though, the final method—`forEach()`—doesn't return an array, so you can't call another method. In fact, it returns nothing, so you can't even assign the output of the whole group of actions to a variable.

Final step: Removing intermediate steps

Removing the intermediate steps, you get an identical set of actions without any variable declarations.

```
const sailors = [
    {
        name: 'yi hong',
        active: true,
        email: 'yh@yhproductions.io',
    },
    {
        name: 'alex',
        active: true,
        email: '',
    },
    {
        name: 'nathan',
        active: false,
        email: '',
    },
];

const active = sailors.filter(sailor => sailor.active);
sailors
.filter(sailor => sailor.active)
.map(sailor => sailor.email || `${sailor.name}@wiscsail.io`)
.forEach(sailor => sendEmail(sailor));
```

Now you're sending an email to the preferred email address of only active members. The best part is that because each array method does one very specific thing, you can understand the code at a glance.

The only downside to chaining array methods is that each time you call a new method, you're iterating over the whole returned array. Instead of three iterations —*one for each member*—if you performed all actions with a `for` loop, you're performing *seven* iterations (*three on the original array plus two more when mapping plus two more when calling `forEach()`*). Don't pay too much attention to this. It's not terribly important unless you're working with large data sets. Sometimes the minor performance increase is worth extra readability. Sometimes it's not. It's just something to keep in mind.

Tricks to chaining methods

There are a few tricks to chaining methods: First, notice how there are *no semicolons* until the final statement. The whole action is like a sentence. It's not over until you hit the period, even when it spans multiple lines.

This is one reason that many style guides still prefer semicolons even though they aren't strictly necessary in JavaScript. If you mess up and include a semicolon earlier, you'll get a `SyntaxError` so it's unlikely you'll get too far with that mistake.

More important, order does matter. You couldn't, for example, flip the `filter()` and the `map()` methods because the `map()` method would remove the property the `filter()` method would need to check. With this example at least, that would be very bad. You wouldn't get an *error* because `sailor.active` would return `undefined` for everything. The resulting array would be empty, which isn't an *error*. In other words, syntactically, everything makes sense even if you provide an empty array to `forEach()`.

This is why it always pays to have a test. Check out the test suite for this course to see examples.

Chaining isn't limited to array methods, but because arrays have so many methods that return arrays, they're very convenient examples to explore. You'll see more chaining as you continue. It pops up again and again. You may remember seeing using it with the Map object in Tip 13, Update Key-Value Data Clearly with Maps, when you chained multiple `set()` methods. And you'll see it again when you work with promises in Tip 43, Retrieve Data Asynchronously with Promises. It's a simple but important concept that's worth reviewing several times.

What will be the output of the following code?

```
const students = [
    {
        name: 'mark',
        marks: 40
    },
    {
        name: 'bill',
        marks: 95
    },
    {
        name: 'steve',
        marks: 60
    },
];


students.filter(student => student.marks > 50)
.map(student => student.name.toUpperCase())
.forEach(student => console.log(`${student} passed`));
```

Retake Quiz

In the next tip, you'll go back and look at one more array method, `reduce()`. It's the most flexible and interesting, but it's also the most unpredictable.