Tip 15: Create Maps Without Side Effects

In this tip, you'll learn how to avoid side effects by creating new maps from an array of pairs.

We'll cover the following

Copying & mutation in maps

Up to this point, you've always worked on a single instance of a map. You've either added data or removed data directly from an instance of a Map object.

Working on the instance of a map can lead to a few problems. *How do you create copies of a map*? *How can you make changes without side effects*?

Fortunately, you can solve those problems by applying a few principles you've learned from arrays and objects.

To start, look at an example that combines the problems of copying and mutations: *applying a set of defaults to a map*.

Example: Applying defaults to a map

In your pet adoption code, you have filters that users have selected, but perhaps you want to add a set of default filters. Any additional filters will be overridden by the user, but any not explicitly set by the user will be the default.

```
const defaults = new Map()
    .set('color', 'brown')
    .set('breed', 'beagle')
    .set('state', 'kansas');

console.log(defaults)

const filters = new Map()
    .set('color', 'black');
    .set('color', 'black');

console.log(filters)
```

Now you’re in a bind. How can you make a new collection of filters, including the defaults and the user-applied filters?

`has` method

If you didn’t care about side effects (and I really hope you do by now), you might be tempted to check to see if the map has a key using the `has()` method. If no key exists, set the key value. If the key already exists, you can ignore it.

```
const defaults = new Map()
    .set('color', 'brown')
    .set('breed', 'beagle')
    .set('state', 'kansas');

const filters = new Map()
    .set('color', 'black');

function applyDefaults(map, defaults) {
    for (const [key, value] of defaults) {
        if (!map.has(key)) {
            map.set(key, value);
        }
    }
}

applyDefaults(filters,defaults);
console.log(filters);
```

If your goal is solely to combine defaults and user data, you’ve succeeded. But by now, your skepticism about mutations should get to you. Consider how you want to use the `filters` object. You use it to filter data, but you also use it to alert the user to the filters they’ve applied (as you did by creating a string in the previous tip).

Now that you’ve mutated the object, it will appear to the user that they applied a bunch of defaults they never selected. Notice that the defaults include a state. You want your users to see only the animals in their state, but you don’t want them to change the state directly. You’d rather they visit the pet adoption page for that state.

Copying & avoiding mutation

The simplest way around this problem is to create a *copy of the map*. As you may recall, you can create a new map by passing in an array of pairs. And you can create a list of pairs with the spread operator.

With that in mind, try to update the code to create a copy before it's mutated.

```
const defaults = new Map()
    .set('color', 'brown')
    .set('breed', 'beagle')
    .set('state', 'kansas');

const filters = new Map()
    .set('color', 'black');

function applyDefaults(map, defaults) {
    const copy = new Map([...map]);
    for (const [key, value] of defaults) {
        if (!copy.has(key)) {
            copy.set(key, value);
        }
    }
    return copy;
}

console.log(applyDefaults(filters,defaults));
```

If you got something like this, great work! You got the copy of the filters, and you applied the defaults to that (note that it's okay to mutate something that's scoped to the function), and then you returned the new map. Now you can be sure that your current `filters` map is safe from side effects while your new map contains all the defaults and all the applied information.

Yet it gets even better. You're still manually checking a bunch of keys for existence. Fortunately, that's not even necessary. Maps, like objects, *can only have a key once*. So if you tried to create a map with a new key, it will use whatever value for that key is declared last. It's as if you were updating the value instead of setting it.

```
const filters = new Map()
    .set('color', 'black')
    .set('color', 'brown');

console.log(filters.get('color'));
```

With this knowledge, you can combine what you know about the object spread operator to create a combination of two maps in one line.

```
let filters = new Map()
    .set('color', 'black');

let filters2 = new Map()
    .set('color', 'brown');

let update = new Map([...filters, ...filters2]);
console.log(update.get('color'));
```

Now when you update the function again, you realize you don't even need the function at all. Combining and creating maps becomes a one liner.

```
const defaults = new Map()
    .set('color', 'brown')
    .set('breed', 'beagle')
    .set('state', 'kansas');

const filters = new Map()
    .set('color', 'black');

function applyDefaults(map, defaults) {
    return new Map([...defaults, ...map]);
}

console.log(applyDefaults(filters,defaults));
```

Maps really do combine some of the best ideas from many other data structures. This should give you some ideas for how you can start using them in your code.

In the next tip, you'll learn about another new collection, Set, which does one thing very well: creating a list of unique items.