Tip 13: Update Key-Value Data Clearly with Maps

In this tip, you'll learn how to use the Map object for key-value collections of frequently updated data.

We'll cover the following

In Tip 10, Use Objects for Static Key-Value Lookups, you learned that you should only use objects deliberately and not as a default collection. Now you're going to get a chance to look at an alternative: `Map`.

Map

Map is a special kind of collection that can do certain things very easily. The Mozilla Developer Network has a nice list of circumstances where Map is a better option for a collection than a plain object. I encourage you to read the full list, but this tip examines two specific situations:

- Key-value pairs are frequently added or removed.
- A key isn't a string.

In the next tip, you'll see another big advantage: using Map for iterating over collections. For now, you just need to be familiar with adding or removing values to maps.

Adding & removing key-value pairs: Example

First, think about what it means that key-value pairs are frequently added and

removed. Consider a pet adoption website. The site has a list of all the adorable dogs that need homes. Because people have different expectations of their pets

(some want big dogs, some like certain breeds), you'll need to include a way to filter the animals.

You'll start off with a collection of animals:

```
const dogs = [
    {
        name: 'max',
        size: 'small',
        breed: 'boston terrier',
        color: 'black'
    },
    {
        name: 'don',
        size: 'large',
        breed: 'labrador',
        color: 'black'
    },
    {
        name: 'shadow',
        size: 'medium',
        breed: 'labrador',
        color: 'chocolate'
    }
]
```

The collection of all `dogs` is an array, which makes sense because the shape of each item in the collection is the same.

You'll need to create one more collection: *your list of applied filters*. The filters will be a collection containing a key (`color`) and a value (`black`). The user will need to be able to *add a filter, remove a filter, and clear all filter values*.

If you added the key "`color`" and the value "`black`" to the collection, then somewhere else in the code base, you'll filter the objects using that information and be left with an array of two dogs. Don't worry about the implementation details. But if you're curious, take a look at Tip 23, Pull Out Subsets of Data with `filter()` and `find()`, to see how to filter an array.

Solving the example using objects

To understand why Map was added to the spec, think of how you might solve the problem with standard objects.

First, you'd make an empty object that will hold the new information:

```
let filters = {};
```

Then you'd need three actions to update the information on the object: add filter, remove a filter, clear all filters.

```
const dogs = [
    {
        name: 'max',
        size: 'small',
        breed: 'boston terrier',
        color: 'black'
    },
    {
        name: 'don',
        size: 'large',
        breed: 'labrador',
        color: 'black'
    },
    {
        name: 'shadow',
        size: 'medium',
        breed: 'labrador',
        color: 'chocolate'
    }
]


const filters = {
      color: 'brown',
};

function addFilters(filters, key, value) {
    filters[key] = value;
}

function deleteFilters(filters, key) {
    delete filters[key];
}

function clearFilters(filters) {
    filters = {};
    return filters;
}

console.log("Addiing a filter:");
addFilters(filters,'size','large');
console.log(filters);
console.log("Deleting a filter:");
deleteFilters(filters,'color');
console.log(filters);
console.log("Clearing filters:");
console.log(clearFilters(filters));
```

The strange thing here is even though you're performing three basic actions on a

collection—setting a key-value, deleting a key-value, clearing all keyvalues—you're

using three different paradigms. The first, setting a key-value, uses a method on the object itself. The second, deleting a key-value pair, uses an operator defined by the language. The third, clearing all data, reassigns a variable. It's not even an action on the object. It's variable reassignment. When you "clear" an object, you're really just writing `filters = new Object();`.

Solving the example using map

By contrast, maps are designed specifically to update key-value pairs frequently. The interface is clear, methods have predictable names, and actions such as loops (as you'll see in the next tip) are built-in. This will make you a more productive developer. The more you can predict, the faster you can create.

To begin, you need to create a new instance of Map and add some data.

Unlike an object, which has a simple constructor shortcut using curly braces, you must always explicitly create a new instance of a `Map`.

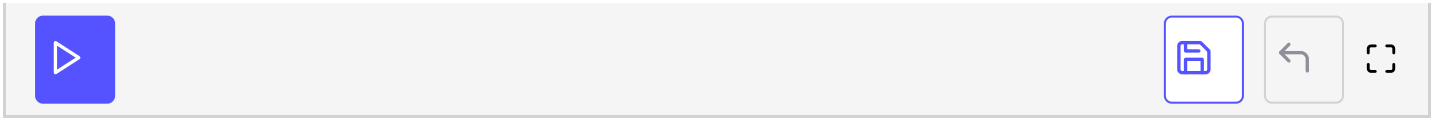
```
let filters = new Map();
```

Notice that you assigned the new map with `let`. `let` is a better choice because you'll be mutating the object by adding some data. You may be a little confused. After spending a lot of time learning how mutations are bad, here's an object that, by necessity, must be mutated whenever you add or remove data. For now, don't worry about the mutations. There is a way around mutating the object, and you'll see it in the next tip.

After creating an instance, you add data with the `set()` method. To add a breed of `'labrador'` to the filter list, you'd pass in the key name `'breed'` as the *first* argument and the value `'labrador'` as the *second* argument.

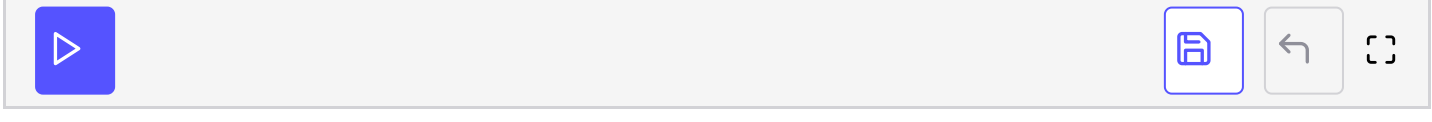
```
const dogs = [
    {
        name: 'max',
        size: 'small',
        breed: 'boston terrier',
```

```
        color: 'black'
    },
    {

        name: 'don',
        size: 'large',
        breed: 'labrador',
        color: 'black'
    },
    {
        name: 'shadow',
        size: 'medium',
        breed: 'labrador',
        color: 'chocolate'
    }
]

let filters = new Map();
filters.set('breed', 'labrador');
console.log(filters);
```



To retrieve date, use the `get()` method, passing in the key as the only argument.

```
const dogs = [
    {
        name: 'max',
        size: 'small',
        breed: 'boston terrier',
        color: 'black'
    },
    {
        name: 'don',
        size: 'large',
        breed: 'labrador',
        color: 'black'
    },
    {
        name: 'shadow',
        size: 'medium',
        breed: 'labrador',
        color: 'chocolate'
    }
]

let filters = new Map();
filters.set('breed', 'labrador');
console.log(filters.get('breed'));
```



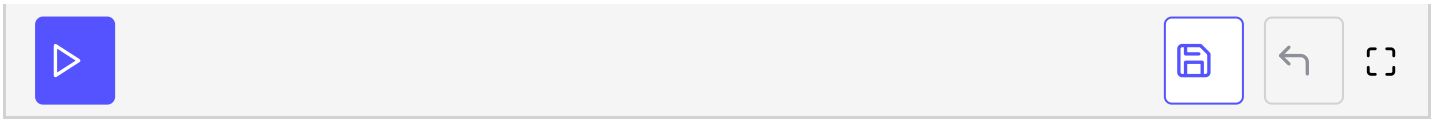
Shortcuts for setting data using map

Getting and setting data is simple, but it can be tedious for a large map.

Fortunately, the creators of the spec anticipated this and created a few shortcuts when setting data.

You can easily add several values with chaining—applying methods one after the other. You can even chain directly from the creation of the new instance. You'll see more about chaining methods in Tip 25, Combine Methods with Chaining.

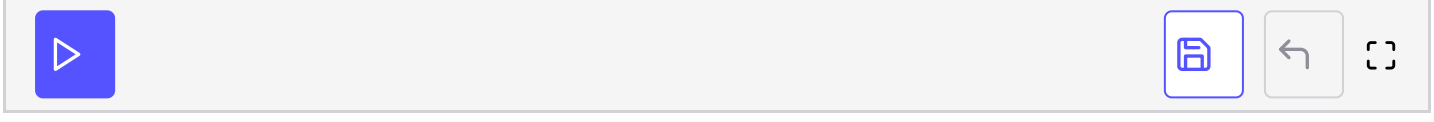
```
let filters = new Map()
    .set('breed', 'labrador')
    .set('size', 'large')
    .set('color', 'chocolate');
console.log(filters);
console.log("Getting the size:");
console.log(filters.get('size'));
```



That's not the only way you can add data. You can also add information using an array.

Remember in Tip 5, Create Flexible Collections with Arrays, you learned that key-value objects can be represented as an array of pairs. Here's a perfect use case. Instead of creating a new `Map` and then chaining setters, you can pass an array of pairs with the first element being a *key* and the second element being a *value*.

```
let filters = new Map(
    [
        ['breed', 'labrador'],
        ['size', 'large'],
        ['color', 'chocolate'],
    ]
)
console.log(filters.get('color'));
```



If you want to remove values, you just need to use the `delete()` method rather then the language operator.

```
let filters = new Map(
    [
        ['breed', 'labrador'],
        ['size', 'large'],
        ['color', 'chocolate'],
    ]
)
filters.delete('color');
console.log(filters.get('color'));
```

Similarly, you can delete all the key-value pairs with the `clear()` method.

```
let filters = new Map(
    [
        ['breed', 'labrador'],
        ['size', 'large'],
        ['color', 'chocolate'],
    ]
)
filters.clear()
console.log(filters.get('color'));
```

With these methods outlined, you have the foundation to change your functions to use a map instead of an object.

```
const petFilters = new Map();

function addFilters(filters, key, value) {
    filters.set(key, value);
}

function deleteFilters(filters, key) {
    filters.delete(key);
}

function clearFilters(filters) {
    filters.clear();
}

console.log("Adding filters:");
addFilters(petFilters,'color','brown');
addFilters(petFilters,'size','large');
console.log(petFilters);
console.log("Deleting filters:");
deleteFilters(petFilters,'color');
console.log(petFilters);
console.log("Clearing filters:");
clearFilters(petFilters);
```

Advantages of using map

The change is subtle but very important. First, the code is much cleaner. That's a big advantage in itself. But you'll see far bigger advantages when you compare these functions to the ones you created with an object. With these functions:

- You always use a method on a `Map` instance.
- You don't mix in language operators after you create the initial instance.
- You don't ever have to create a new instance to perform a simple action.

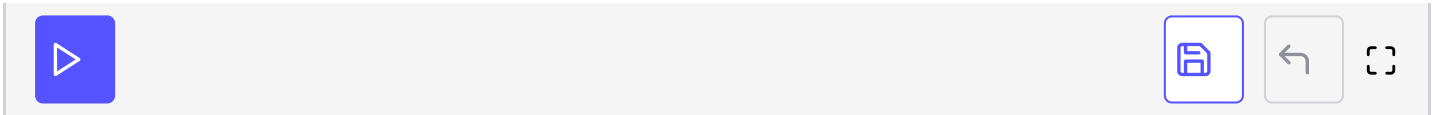
These are the reasons why maps are so much easier to work with than objects when you're frequently changing the information. Every action and intention is very clear.

In addition, with objects you're limited in the types of keys you can use. Objects can use only certain types of keys. Most significantly, you can't use integers as a string, which causes problems if you want to store information by a numerical ID. For example, if you have an object of error codes:

```
const errors = {
    100: 'Invalid name',
    110: 'Name should only contain letters',
    200: 'Invalid color'
};
```

You may innocently think you could retrieve error text by the numerical code.

```
const errors = {
    100: 'Invalid name',
    110: 'Name should only contain letters',
    200: 'Invalid color'
};

function isDataValid(data) {
    if (data.length < 10) {
        return errors.100
    }
    return true;
}

console.log(isDataValid("cat"));
```



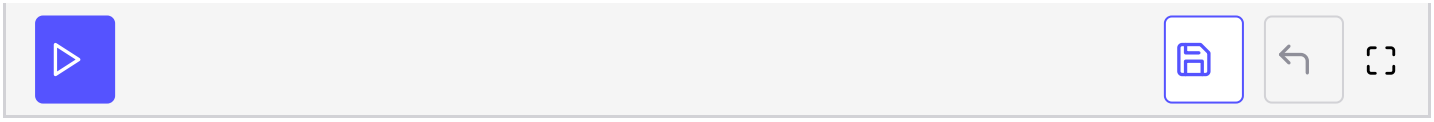
This code would throw an **Error**.

> Integers as keys can't be accessed with dot syntax.

You're still able to access the information using array notation `errors[100]`. But

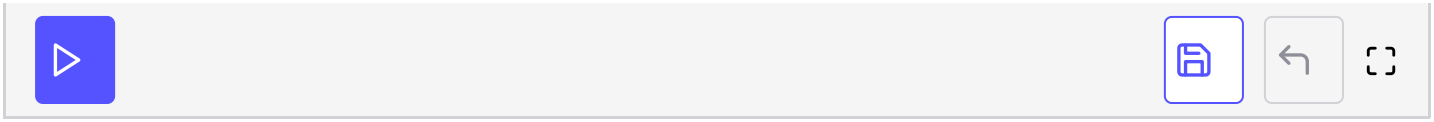
that's actually a bit of a trick. You get the right result because when you created the error array, it converted all the integers to strings. And when you use array syntax, it's also converting the integer to a string before lookup. If you tried to get the keys, it would return an array of strings:

```
const errors = {
    100: 'Invalid name',
    110: 'Name should only contain letters',
    200: 'Invalid color'
};

console.log(Object.keys(errors));
```



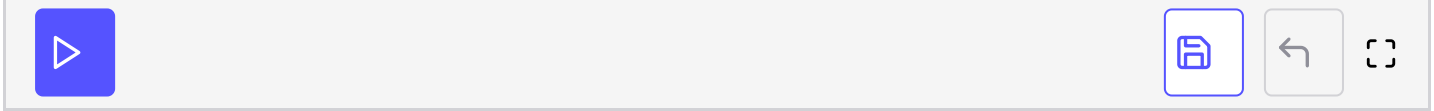
A `Map` wouldn't have that problem. It can take many different types as keys.

```
let errors = new Map([
    [100, 'Invalid name'],
    [110, 'Name should only contain letters'],
    [200, 'Invalid color']
]);
console.log(errors.get(100));
```



In case you're wondering, you can also get the keys from a `Map` as you could with an object.

```
let errors = new Map([
    [100, 'Invalid name'],
    [110, 'Name should only contain letters'],
    [200, 'Invalid color']
]);

console.log(errors.keys());
```



Notice something strange? When you asked for the keys, you didn't get an array, as you do with `Object.keys()`. You didn't get an object, or even another `Map`; the return value is something called *MapIterator*. Don't worry—it's actually a great thing to have. The **MapIterator** is what will allow us to loop through data.

What is the output of the code below?

```
const map = new Map();

const foods = { dinner: 'Curry', lunch: 'Sandwich', breakfast: 'E
ggs' };
const normalfoods = {};

map.set(normalfoods, foods);

for (const [key] of map) {
  console.log(map.get(key));
}
```

Retake Quiz

In the next tip, you'll see how the *MapIterator* is the killer feature that will make you return to `Map` over and over again.