Tip 11: Create Objects Without Mutations Using Object.assign()

In this tip, you'll learn how to update an object without mutations, using Object.assign().

We'll cover the following

- Mutation problem in objects
- Updating data without mutation
- Object.assign
    - Example
- Mutation problem with Object.assign
    - Solution
- Problem with copying nested objects
    - Deep copy
    - Solution

Mutation problem in objects

In the previous tip, you took a quick look at objects and got rules for when they offer distinct advantages over other collections. Still, you need to be careful when using them because they can leave you open to the same problems with mutations and side effects that you saw in arrays. Casually adding and setting fields on objects can create unseen problems.

Consider a very common problem. You have an object with a number of *key-values* pairs. The problem is that the object is incomplete. This happens often when you have legacy data and there are new fields, or you are getting data from an external API and you need it to match your data model. Either way, the issue is the same: *you want to fill in the remaining fields using a default object*.

Updating data without mutation

How can you create a new object that preserves the original data while adding in the defaults? And, of course, you don't want side effects or mutations.

Take a moment to write out the code. See what you come up with.

If you wrote the code out, it would probably look something like this:

```
const defaults = {
    author: '',
    title: '',
    year: 2017,
    rating: null,
};
const book = {
    author: 'Joe Morgan',
    title: 'Simplifying JavaScript',
};
function addBookDefaults(book, defaults) {
    const fields = Object.keys(defaults);
    const updated = {};
    for (let i = 0; i < fields.length; i++) {
        const field = fields[i];
        updated[field] = book[field] || defaults[field];
    }
    return updated;
}

console.log(addBookDefaults(book,defaults));
```

There's nothing wrong with this code, but it sure is wordy. Fortunately, this was a common enough issue that ES5 introduced `Object.assign()` to *create and update* fields on an object with keys and values from another object (or objects).

In other words, `Object.assign()` lets you update an object with properties from another object.

`Object.assign`

So how does `Object.assign()` work? It's fairly simple. The method takes a series of objects and updates the inner-most object with the keys and values from outer objects, then returns the updated first object. The outermost object has precedence over any inner objects.

Example

It's easier to see than explain, but when you see how simple it is, you'll love it. Here's how you can rewrite `addBookDefaults()` using `Object.assign()`:

```
const defaults = {
    author: '',
    title: '',
    year: 2017,
    rating: null,
};

const book = {
    author: 'Joe Morgan',
    title: 'Simplifying JavaScript',
};

function addBookDefaults(book, defaults) {
    return Object.assign(defaults, book);
}

console.log(addBookDefaults(book,defaults));
```

Mutation problem with `Object.assign`

Your nine-line function dropped to a single line. But by now, you may have guessed there’s a problem in this code. When it updates the initial object— the default object—it also mutates the original. If you run the code again with a different book object, you’ll get an unexpected result.

```
const defaults = {
    author: '',
    title: '',
    year: 2017,
    rating: null,
};

const book = {
    author: 'Joe Morgan',
    title: 'Simplifying JavaScript',
};

const anotherBook = {
    title: 'Another book',
    year: 2016,
};

function addBookDefaults(book, defaults) {
    return Object.assign(defaults, book);
}


console.log("Using book: ");
console.log(addBookDefaults(book,defaults));
console.log("\n");
console.log("Using another book: ");
console.log(addBookDefaults(anotherBook,defaults));
```

You accidentally changed the default object to make me, `‘Joe’`, the default author, so I’m going to start getting credit for a whole bunch of books I’ve never written.

Solution

Fortunately, the solution is simple. Just make the first object an *empty object*. After you do that, the returned object will be the updated empty object. The other objects will have no mutations.

```
const defaults = {
    author: '',
    title: '',
    year: 2017,
    rating: null,
};
const book = {
    author: 'Joe Morgan',
    title: 'Simplifying JavaScript',
};

const updated = Object.assign({}, defaults, book);
console.log(`Updated object:`);
console.log(updated);
console.log("\n");

console.log(`Original object:`);
console.log(defaults);
```

Problem with copying nested objects

Now, there’s one problem with copying objects using `Object.assign()`. When it copies over properties, it just copies the values. That may seem like it’s not a problem, but it is.

Up to this point, you’ve been working with flat objects. Every key had a simple value: *a string or an integer*. And when all you have is a series of strings or integers, it copies them just fine, as you saw earlier. But when the value is another object, you start to have problems.

```
const defaultEmployee = {
    name: {
        first: '',

        last: '',
    },
    years: 0,
};
const employee = Object.assign({}, defaultEmployee);
console.log(employee);
```

Deep copy

Copying objects that have nested objects is called **“deep copying”** (or **“deep merging”** or some variation). The property `years` will copy over just fine, but the property `name` isn’t copied. All that’s copied is a *reference* to the independent object that’s assigned to the key `name`. The nested object essentially exists independently of the object that holds it. All the containing object has is a *reference* to that object. When you copy the reference, you aren’t making a deep copy of the nested object. You’re merely copying the location of the reference.

So if you change a value of a nested object on either the original or the copy, it will change the value on both.

```
const defaultEmployee = {
    name: {
        first: '',
        last: '',
    },
    years: 0,
};

const employee = Object.assign({}, defaultEmployee);
employee.name.first = 'Joe';
console.log("Employee:");
console.log(employee);
console.log("Default employee:");
console.log(defaultEmployee);
```

Solution

There are two ways around this problem: The first and simplest is to keep your objects flat—don’t have nested objects if you can avoid it.

Unfortunately, that doesn’t work in a situation where you start off with a nested

Unfortunately, that doesn't work in a situation where you start off with a nested object. Maybe the software was designed with nested objects. Maybe you're getting

a result from an API that's nested. It doesn't matter. Nested objects are very common.

In that case, you can copy the nested objects with `Object.assign()`; you just need a little more code. Whenever there is a nested object, copy that with `Object.assign()` and everything will be updated.

```
const defaultEmployee = {
    name: {
        first: '',
        last: '',
    },
    years: 0,
};

const employee = Object.assign(
    {},
    defaultEmployee,
    {
        name: Object.assign({}, defaultEmployee.name),
    },
);

employee.name.first = 'Joe';
console.log("Employee:");
console.log(employee);
console.log("\n");
console.log("Default employee:");
console.log(defaultEmployee);
```

Of course, there are other options: A library like `Lodash` has a method called `cloneDeep()` that can do this for you. And by all means, take advantage of community libraries, but sometimes you may want to make a change without external code.

If you're thinking that code is getting ugly fast, you're not wrong. It feels like it could be simpler. Sure, you can abstract it out into a helper function, but fortunately, you may not even need to do that. There's experimental syntax that, though not adopted, is widely used throughout the JavaScript community and will likely be part of the official spec soon. The best part is, it looks exactly like something you've already seen. It's called the *Object Spread* operator, and it will give you the ability to make new objects with the now familiar spread operator.

Q !

What will be the output of the code given below?

```
var obj1 = { a: 5, b: 5, c: 5 };
var obj2 = { b: 20, c: 30 };
var obj3 = { a: 10 ,c: 40 };
var newobj = Object.assign({}, obj1, obj2, obj3);
console.log(newobj);
```

Retake Quiz

In the next tip, you’ll learn how use the new syntax to update object information quickly and clearly.