Tip 9: Avoid Sort Confusion with the Spread Operator

In this tip, you'll learn how to use the spread operator to sort an array multiple times while getting the same result.

We'll cover the following

Sorting tabular data

You've seen by now that you can replace many mutating functions with the spread operator. What should you do when there's a function that you can't easily replace? The answer is fairly simple: *Use the spread operator to create a copy of the original array, and then mutate that one*.

Don't let the simplicity of the answer fool you. Mutation bugs can sneak up when you least expect them.

This comes up in applications that have *tabular sorting data*. If you haven't written an application that displays tabular data, wait around—I guarantee you'll do it. And the minute you create that table of tabular data, the next request you'll hear from your account or project manager is to make the table sortable.

Example: Mutation when sorting a table

Skip the UI components and look purely at the data and functions. You need to make an application that takes an *array* of staff members and sorts them either by *name or years of service*.

Start with an array of employees.

```
const staff = [
    {
        name: 'Joe',
        years: 10,
    },
    {
        name: 'Theo',
        years: 5,
    },
    {
        name: 'Dyan',
        years: 10,
    },
];
```

Next, add a couple of custom *sorting functions* to sort by either *name* or *age*. If you don't understand the sort functions, don't worry. It's not necessary for this example. If interested, you can check out the sort documentation on the Mozilla Developer Network.

```
function sortByYears(a, b) {
    if (a.years === b.years) {
        return 0;
    }
    return a.years - b.years;
}

const sortByName = (a, b) => {
    if (a.name === b.name) {
        return 0;
    }
    return a.name > b.name ? 1 : -1;
};
```

At this point, you'd just call the sort function on the array whenever the user clicks a column heading. For example, if a user chooses to sort by years of service, the function will sort and update the array.

```
const staff = [
    {
        name: 'Joe',
        years: 10,
    },
    {
        name: 'Theo',
        years: 5,
    },
    {
        name: 'Dyan',
        years: 10,
    },
];

function sortByYears(a, b) {
    if (a.years === b.years) {
        return 0;
    }
    return a.years - b.years;
}

const sortByName = (a, b) => {

    if (a.name === b.name) {
        return 0;
    }
    return a.name > b.name ? 1 : -1;
};

//sorting by years
const values = [...staff];
console.log(values.sort(sortByYears));
```

Now this is where it gets tricky. When you sorted the array, you changed it. Even though the code looks likes it's finished executing, the change is still there.

Suppose the user then sorted by user name. Again, the array mutates.

```
const staff = [
    {
        name: 'Joe',
        years: 10,
    },
    {
        name: 'Theo',
        years: 5,
    },
    {
        name: 'Dyan',
        years: 10,
    },
];

function sortByYears(a, b) {
    if (a.years === b.years) {
        return 0;
    }
    return a.years - b.years;
}

const sortByName = (a, b) => {
    if (a.name === b.name) {
        return 0;
    }
    return a.name > b.name ? 1 : -1;
};

const values = [...staff];
//sorting by years followed by sorting by name
values.sort(sortByYears);
console.log(values.sort(sortByName));
```

Nothing spectacular, but look what happens if the user goes back and sorts by years of service again. Maybe they forgot a name. Maybe they needed some different information. Who knows?

What result would the user see? What result do you think the user should see? Turns out, sorting by years a second time yields completely different results.

```
const staff = [
    {
        name: 'Joe',
        years: 10,
    },
    {
        name: 'Theo',
        years: 5,
    },
    {
        name: 'Dyan',
        years: 10,
    },
];

function sortByYears(a, b) {
    if (a.years === b.years) {
        return 0;
    }
    return a.years - b.years;
}

const sortByName = (a, b) => {
    if (a.name === b.name) {
        return 0;
    }
    return a.name > b.name ? 1 : -1;
};

const values = [...staff];
//sorting by years
values.sort(sortByYears);
//next, sorting by name
values.sort(sortByName);
//lastly, sorting by years a second time
console.log(values.sort(sortByYears));
```

This is a simple example. Imagine a table of hundreds of employees with many of the employees sharing the same years of service. Every time a user clicks the sort button, they'd see a slightly different order.

At that point, your user has lost trust in the application. That's something you don't want. Mutations can have big impacts.

Mutating a copy

How do you stop mutations when the method you want to use has to mutate the data? The answer is simple: *Don't mutate the data*. Make a copy and then perform the mutation.

The only thing you need to change in your code is to *spread* the original array into a *new array* before sorting.

```
const staff = [
    {
        name: 'Joe',
        years: 10,
    },
    {
        name: 'Theo',
        years: 5,
    },
    {
        name: 'Dyan',
        years: 10,
    },
];

function sortByYears(a, b) {
    if (a.years === b.years) {
        return 0;
    }
    return a.years - b.years;
}

const sortByName = (a, b) => {
    if (a.name === b.name) {
        return 0;
    }
    return a.name > b.name ? 1 : -1;
};

const values = [...staff];
//sorting by years
values.sort(sortByYears);
//next, sorting by name
values.sort(sortByName);
//lastly, sorting by years a second time
console.log([...staff].sort(sortByYears));
```

Now your users can sort however much they want because we aren't changing the original array. The results will always be the same as the previous sort for that type.

The spread operator is great, not because it’s complex (you’ll see some fancier collections in just a moment), but because it’s so incredibly simple while still being incredibly powerful.

In the next chapter, you’ll start to branch out into other collections. You’ll learn when it is appropriate to use Map, Set, or standard objects.