Tip 7: Mold Arrays with the Spread Operator

In this tip, you'll learn how to simplify many array actions with the spread operator.

We'll cover the following

Spread operator

As you've seen, arrays provide an incredible amount of flexibility for working with data. But the number of methods that an array contains can be confusing, and it could lead you to some problems with mutations and side effects. Fortunately, the spread operator gives you a way to create and manipulate arrays quickly with minimal code.

The **spread operator**, symbolized with three dots (`...`), may be the most widely used new feature in JavaScript. You're likely to find it in nearly every file containing ES6+ syntax.

That said, it's hard to take the spread operator seriously. I certainly didn't. What it does is so mundane: *It converts an array to a list of items*. Turns out, that tiny action has many benefits that we'll explore in the next few tips.

The benefits don't end with just arrays. You'll see the spread operator over and over. It pops up in the `Map` collection, as you'll see in Tip 14, Iterate Over Key-Value Data with Map and the Spread Operator. You'll use a variation called the *rest* operator in functions, as you'll see in Tip 31, Pass a Variable Number of Arguments with the Rest Operator. And you can use the spread operator on any data structure

or class property using *generators*, as you'll see in Tip 41, Create Iterable Properties with Generators.

Example: Using the spread operator on arrays

I hope that I've sparked your interest. To start, try using the spread operator on a simple array.

You begin with an array of items.

```
const cart = ['Naming and Necessity', 'Alice in Wonderland'];
```

You then use the spread operator (`...`) to turn that into a list—*a series of items that you can use in parameters or to build an array*:

```
const cart = ['Naming and Necessity', 'Alice in Wonderland'];
...cart
```

When you try the code above you get an error. If you try this out in a REPL or a browser console, you'll get an *error*. The syntax is correct, but you can't use the spread operator on its own. You can't, for example, assign the output to a variable. You have to spread the information into something.

```
const cart = ['Naming and Necessity', 'Alice in Wonderland'];
const copyCart = [...cart];
console.log(copyCart);
```

Now, before you think "big deal" and skip to the next tip, I want you to know I understand. I didn't appreciate the spread operator until I started seeing it pop up everywhere. And I didn't love it until I started using it. But now it's my favorite ES6 feature by far.

Example: Removing an item from an array

To see how powerful the spread operator can be, start with a simple task: *removing* *an item from an array.*

Using a loop

Here’s an approach using only a loop:

```
function removeItem(items, removable) {
    const updated = [];
    for (let i = 0; i < items.length; i++) {
        if (items[i] !== removable) {
            updated.push(items[i]);
        }
    }
    return updated;
}

const items = ['apple', 'banana', 'orange'];
console.log(removeItem(items,'banana'));
```

This isn’t bad code. But there’s certainly a lot of it. It’s a good rule to keep things as simple as you can. The more clutter and loops that exist, the harder it will be to read and understand the code.

Using `splice`

In trying to simplify, you may stumble on an array method called `splice()`. It removes an item from an array, and that’s exactly what you want! If you refactor the preceding function, it does become more simple.

```
function removeItem(items, removable) {
    const index = items.indexOf(removable);
    items.splice(index, 1);
    return items;
}

const items = ['apple', 'banana', 'orange'];
console.log(removeItem(items,'banana'));
```

The problem with `splice()` is that it mutates the original array. Take a look at the following example and see if you can spot the problem:

```
function removeItem(items, removable) {
    const index = items.indexOf(removable);
    items.splice(index, 1);
    return items;
}


const books = ['practical vim', 'moby dick', 'the dark tower'];
const recent = removeItem(books, 'moby dick');
console.log("Recent books: " + recent);
const novels = removeItem(books, 'practical vim');
console.log("Novels: " + novels);
```

What do you think the novels array will contain?

The only book it will contain is `'the dark tower'`. When you called `removeItem()` the first time, you passed it books and got back the array without `'moby dick'`. But it also changed the `books` array. When you passed it to the next function, it was only *two items* long.

This is why mutations can be so hazardous, particularly if you're using them in a function. You may not expect the information passed to be fundamentally different. Notice in this case that you're even assigning books with `const`. You may assume this won't be mutated, but that isn't always the case.

`splice` may seem like a good alternative to a for loop, but mutations can create so much confusion that you're better off avoiding them whenever possible.

Using `slice`

Finally, there's one last option. Arrays also have a method called `slice()`, which returns a part of an array without changing the original array. When you're using `slice`, you pass a *startpoint* and *endpoint*, and you get everything in between. Alternatively, you can pass just a startpoint and get everything from that point until the end of the array. Then you can use `concat()` to put the pieces of the array back together.

```
function removeItem(items, removable) {
    const index = items.indexOf(removable);
    return items.slice(0, index).concat(items.slice(index + 1));
}


const books = ['practical vim', 'moby dick', 'the dark tower'];
const recent = removeItem(books, 'moby dick');
console.log("Recent books: " + recent);
const novels = removeItem(books, 'practical vim');
```

```
console.log("Novels: " + novels);
```

This code is pretty great. You get the new array back without changing the original array, and you avoid a lot of code. However, it isn't clear what's being returned. Another developer would need to know that `concat()` joins two arrays into a single flat array. There's no visual clue for what you're doing. This is where the spread operator comes in.

Using the spread operator

Combined with a `slice`, the *spread operator* turns both sub-arrays into a list that's placed back into square brackets. It actually looks like an array. And more importantly, it gives you a smaller array without affecting the larger array.

```
function removeItem(items, removable) {
    const index = items.indexOf(removable);
    return [...items.slice(0, index), ...items.slice(index + 1)];
}


const books = ['practical vim', 'moby dick', 'the dark tower'];
const recent = removeItem(books, 'moby dick');
console.log("Recent books: " + recent);
const novels = removeItem(books, 'practical vim');
console.log("Novels: " + novels);
```

Notice a few things about this code.

- There are no mutations.
- It's easy to read.
- It's simple.
- It's reusable.
- It's predictable.

In short, it has all of your favorite attributes.

You can actually further improve this code. As you'll see in Tip 23, Pull Out Subsets of Data with filter() and find(), you can pass a function that removes a specific item

in an array. There are many ways to perform the same action. Go for the one that best communicates your intentions.

This is just the beginning. The spread operator lets you quickly pull out the items of an array with very few characters. And you'll always put them back into a structure that you can quickly and easily recognize.

If you look back through the four examples, you see that they all work. But the spread is the most readable and the easiest to predict.

Spread operator & functions

The other popular way to use the spread operator is to create a list of arguments for a function. Create a small function to format an array of information.

```
const book = ['Reasons and Persons', 'Derek Parfit', 19.99];

function formatBook(title, author, price) {
    return `${title} by ${author} ${price}`;
}
```

How can you put the information into the function? Try it out. You probably came up with something like this:

```
const book = ['Reasons and Persons', 'Derek Parfit', 19.99];

function formatBook(title, author, price) {
    return `${title} by ${author} ${price}`;
}

console.log(formatBook(book[0], book[1], book[2]));
```

But there's an even simpler version that you won't have to change if the amount of data on the book changes. For example, say you add a publication year.

If you came up with something like this, great work. Parameters are lists of arguments, so the spread operator allows you to convert an array to a list of parameters quickly and easily.

```
const book = ['Reasons and Persons', 'Derek Parfit', 19.99];

function formatBook(title, author, price) {
    return `${title} by ${author} ${price}`;
}

console.log(formatBook(...book));
```

Here's the interesting thing: This isn't the only way you can quickly extract information from an array in parameters. You could also pull it out directly using **array destructuring**. You'll explore destructuring in greater detail in Tip 29, Access Object Properties with Destructuring.

And that's not all! The spread operator really starts to shine in parameters once you begin to use a variable number of arguments. If you want a quick look, jump ahead to Tip 31, Pass a Variable Number of Arguments with the Rest Operator. As you can see, the spread operator is incredibly useful and there's plenty more to explore.

Q

What is the output of the following code?

```
function func1(arr1, arr2, n) {
  let arr3 = [...arr2];
  arr3.splice(n,0, ...arr1);
  return arr3;
}

console.log(func1([1, 2, 3], [4, 5, 6], 1));
```

Retake Quiz

Now that you’ve seen how it works, it’s time to look at how you can rewrite common array actions using the spread operator to avoid confusing mutations and side effects.