Tip 26: Transform Array Data with reduce()

In this tip, you'll learn how use reduce() to generate a new array with a different size and shape.

We'll cover the following

You're probably tired of hearing me say that good code is predictable. But it's true. Array methods are wonderful because you have an idea of the result at a glance without even understanding the callback function. Not only that, but array methods are easier to test and, as you'll see in Tip 32, Write Functions for Testability, it's much easier to write testable code than it is to add tests to complex code.

`reduce()`

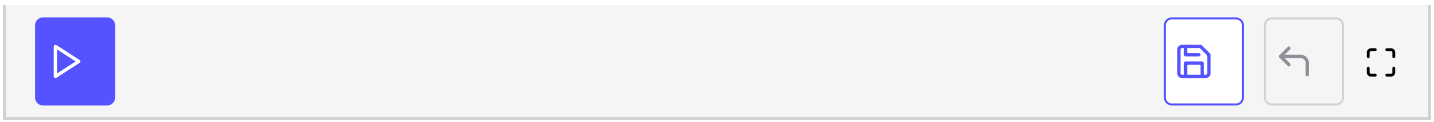
Still, there are times when you need to create a new, radically different piece of data from an array. Maybe you need to get a count of certain items. Maybe you want to transform the array to a different structure, such as an object. That's where `reduce()` comes in. The `reduce()` method is different from other array methods in several ways, but the most important is that it can *change* both the *size* and the *shape* of data (*or just one or just the other*). And it doesn't necessarily return an array.

Building a `reduce()` function

As usual, it's much easier to see than to explain. Here's a reduce function that returns the exact same array. It's useless, but it lets you see how a `reduce()` function is built.

```
const callback = function (collectedValues, item) {
    return [...collectedValues, item];
};

const saying = ['veni', 'vedi', 'veci'];
const initialValue = [];
const copy = saying.reduce(callback, initialValue);
console.log(copy);
```



What's going on here? To start, notice that you pass two arguments into the `reduce()` *callback* function on **line 1**: the *return* item (called `collectedValues`) and the individual item. The *return value*, sometimes called the **carry**, is what makes `reduce()` unique. It can range from an *integer* to a *collection* such as `Set`.

The `reduce()` method itself on **line** 7 takes two values: the *callback function* and the *initial value*. Although the initial value is *optional*, it's usually included because you need something to hold the return values and, as a bonus, it gives other developers a clue about what they'll get back. The trickiest part of a reduce method is that the callback function *must always* return the *carry* item.

It's really worth reading the documentation to see more examples, but many of those examples are abstract ideas using numbers. Consider a situation that's much more common: *getting the unique values from an array*.

You probably remember that this is a problem you already solved with `Set` in Tip 16, Keep Unique Values with Set. You're absolutely correct, but you're going to expand on the solution to get several sets of unique values.

Example 1

As in the previous example, you're going to get a list of unique values from a collection of `dogs` for an adoption website.

```
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',
    },
];
```

If you want to see a few approaches using a `for` loop or `Set` directly, flip back to that earlier tip. For now, you'll jump right into getting the values with the `reduce()` method.

If all you wanted was the set of unique colors, you'd write a `reduce` method that loops through the objects checking the colors and saving the unique values.

```
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 colors = dogs.reduce((colors, dog) => {
    if (colors.includes(dog.color)) {
        return colors;
    }
    return [...colors, dog.color];
}, []);

console.log(colors);
```

Explanation

When you see a `reduce` method, the best place to start is at the end so you can see what kind of item you'll end up with. Remember that it can be anything—*a string, a Boolean, an object*. Make no assumptions.

If you look at **line 6**, you can see that you're initializing the function with an *empty* array.

The next trick to grasping a `reduce` function is understanding what the name of that initial value is after it enters the function. Generically, it's often called a **"carry"**, but you can name it whatever you want because it's just a parameter. In this function on **line 1**, you name it something a little more revealing: `colors`.

Without going any further into the body of the function, you already know that you'll be getting back another array. That's valuable information, and it's the reason you should always start with an explicit carry value. You want the next developer to have as many clues as possible.

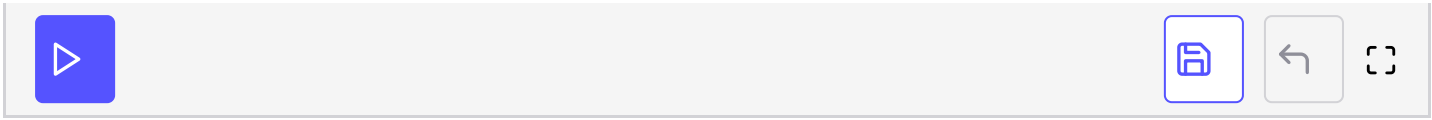
You have to be careful, though, because if you forget to return the carry value, it will effectively disappear. If you were to run the following function, you'd get a `TypeError: Cannot read property 'includes' of undefined`. When you forget to *return* the carry on **line 5**, the function will return `undefined`. This means the parameter `colors` is now *undefined* and doesn't have an `includes()` method.

```
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 colors = dogs.reduce((colors, dog) => {
    if (colors.includes(dog.color)) {
        return colors;
    }
    [...colors, dog.color];
```

```
}, []);

console.log(colors);
```



Moving in to the body of the initial unique colors function, you start to see the value of `reduce()` over other methods. On **line 2**, you check to see if the value is already in the array. If it is, no need to add it. Return the collection so far. If it's a new value, then you add it to the other `colors` on **line 5** and return the updated array.

Let that sink in a moment. You're doing two things:

- You're returning a subset of data (changing the size)
- You're returning modified data (changing the shape).

More importantly, you're changing the size based on information contained inside the array itself. That's not something you can do with `filter()` or `find()`.

Now here's the interesting part. Because you can change both the size and the shape of the data, you can recreate any other array method with `reduce()`.

As a quick example, if you wanted to just get the colors of the dogs, you could use the `map()` method like this:

```
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 colors = dogs.map(dog => dog.color);
console.log(colors);
```

You could get the same value with a reduce function that takes an empty array to start and returns the array on every iteration.

```
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 colors = dogs.reduce((colors, dog) => {
    return [...colors, dog.color];
}, []);

console.log(colors);
```

Exercise

Why use a reducer?

You may be wondering why you should bother with a `reducer` at all when you just pass the results of the `map()` method in to `Set` and get the same result?

That's easy. Reducers give you the flexibility to handle more values with ease. And if you were getting the values for one set of properties, `map()` would make more sense. Remember that flexibility is good, but you should use it only when you've exhausted simpler options. When you need it, though, it's good to have.

For example, what if you wanted to get the unique values for all the keys in the dog object? You could run multiple `map` functions and pass those to `Set`. Or, you can use a `reduce` function that starts with empty sets and fills the objects in as you go.

There are many ways to do this, but the easiest would be to start with an object that contains empty sets. In the body of the `reduce` function, add each item to the set (remember that it will keep only the unique items). When you're finished, you have a collection of unique properties.

```
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 = dogs.reduce((filters, item) => {
        filters.breed.add(item.breed);
        filters.size.add(item.size);
        filters.color.add(item.color);
        return filters;
}, {
        breed: new Set(),
        size: new Set(),
        color: new Set(),
});

console.log(filters);
```

Now you have the benefit of keeping iterations low while also signaling the shape of the transformed data to other developers.

And it's precisely because you can change the size and shape of data that the possibilities are nearly endless.

Example 2

Look at another example. In this case, you have a list of `developers`, and along with `language` specialty, you want a count by speciality.

```
const developers = [
    {
        name: 'Jeff',
        language: 'php',
    },
    {
        name: 'Ashley',
        language: 'python',
    },
    {
        name: 'Sara',
        language: 'python',
    },
    {
        name: 'Joe',
        language: 'javascript',
    },
];
```

You could easily get a count by incrementing the language specialty on each iteration.

```
const developers = [
    {
        name: 'Jeff',
        language: 'php',
    },
    {
        name: 'Ashley',
        language: 'python',
    },
    {
        name: 'Sara',
        language: 'python',
    },
    {
        name: 'Joe',
        language: 'javascript',
    },
];

const aggregated = developers.reduce((specialities, developer) => {
    const count = specialities[developer.language] || 0;
    return {
        ...specialities,
        [developer.language]: count + 1,
    };
}, {});

console.log(aggregated);
```

Notice that the initial value is just an empty object. In this case, you don't know what languages are going to be used so you'll need to add them dynamically. In case you're wondering: Yes, you can build this reduce function with `Map` instead of an object. *Try it out and see what you come up with*.

That's all for array methods. They provide a lot of value, and when you get comfortable with them, you'll appreciate how quickly you can reduce the lines of code while being even more transparent about the information you're returning. Don't be surprised that you turn to them more and more.

Still, there are times when normal `for` loops are the way to go.

What will be the output of the following code?

```
const func = (string) => string.split('').reduce((rev, char) => c
har + rev, '')
console.log(func("hello"));
```

Retake Quiz

In the next tip, you'll look at a slight variation to the for loop called a `for...in` loop that lets you ignore all the annoying declarations of iterators and length by taking each item directly from the iterable.