Tip 16: Keep Unique Values with Set

In this tip, you'll learn how to quickly pull unique items from an array with Set.

We'll cover the following

Set

Set is a fairly simple collection that can do only one thing, but it does it very well. Set is like a *specialized array* that can contain only **one** instance of each *unique* item. You'll often want to collect values from a large array of objects, but you only need to know the unique values. There are other use cases as well, but collecting a list of distinct information from a group of objects is very, very common.

Example

In that spirit, return once again to our set of filters that you're building. To even know what a user can filter on, you need to gather all the possible values. Recall the array of `dogs` that you worked with earlier.

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

How would you get a list of all the `color` options? In this case, the answer is obvious, but what if the list grows into several hundred dogs? How can you be sure we get all the potential choices from golden retrievers to blue pit bulls to mottled border collies?

Accessing values in a collection

One simple way to get a collection of all the colors is to use the `map()` array method. You'll explore this more in Tip 22, Create Arrays of a Similar Size with map(), but for now, all you need to know is that it will return an array of only the colors.

```
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'
    }
]

function getColors(dogs) {
    return dogs.map(dog => dog.color);
}

console.log(getColors(dogs));
```

Getting unique values using `for` loop

Getting unique values using `for` loop

The problem is that this is only the first part. Now that you have all the colors, you need to reduce that to an array of *unique* values. You could pull those out in a number of different ways. There are `for` loops and `reduce()` functions. But for now, stick with a simple `for` loop.

```
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'
    }
]

function getColors(dogs) {
    return dogs.map(dog => dog.color);
}

function getUnique(attributes) {
    const unique = [];
    for (const attribute of attributes) {
        if (!unique.includes(attribute)) {
            unique.push(attribute);
        }
    }
    return unique;
}

const colors = getColors(dogs);
console.log(getUnique(colors));
```

Seems easy enough, but fortunately now you don’t even need to write that much code. You can use the `Set` object to handle the work of pulling out unique values.

Getting unique values using `Set`

A **set** is a common data type and you may be familiar with it from other languages.

A Set is a common data type and you may be familiar with it from other languages.

The interface is very simple and resembles `Map` in many ways. The main difference is that instead of taking an array of pairs, you can create a new instance of `Set` by passing a flat array as an argument.

If you pass your array of `colors` into a set, you're nearly there.

```
const colors = ['black', 'black', 'chocolate'];
const unique = new Set(colors);
console.log(unique);
```

You probably noticed that the value of the object is a `Set` containing only one instance of each color. And that may seem like a problem. You don't want a Set—you want an array of unique items.

Using the spread operator on `Set`

Well, by now you may have guessed the solution: *the spread operator*. You can use the spread operator on `Set` much like you did with `Map`. The only difference is that `Set` returns an *array*. Exactly what you want! Now you can refactor the `getUnique()` function to a one liner. Notice that you can even use the spread operator on instance creation—you don't even need to assign it to a variable.

```
const colors = ['black', 'black', 'chocolate', 'yellow'];

function getUnique(attributes) {
    return [...new Set(attributes)];
}

console.log(getUnique(colors));
```

Maybe this code still doesn't sit well with you. Good! That means your intuition is sharpening. If it seems like you're being inefficient, you're correct. You're first looping over the array of dogs to get an array of colors; then you're manipulating that array to get a list of unique values. Can't you do both at once? You sure can.

Using `Set` methods

Set, again, is similar to `Map` in that you have methods to add and check for values. For a set, you can add a value with `add()` and check a value with `has()`. You also have `delete()` and `clear()`, which work exactly as they do in Map.

This all means that you can add items to a set individually as you go through a loop instead of all at once by passing an array of values. A set can keep only one of each value. If you try to add a value that isn't yet in the set, it will be added. If you try to add a value that already exists, it will be ignored. Order is preserved, and the initial point a value is added will remain. If you try to add an item that's there already, it keeps the original position.

```
let names = new Set();
names.add('joe');
console.log(names);
names.add('bea');
console.log(names);
names.add('joe');
console.log(names);
```

You now have the tools to get the unique values in one pass through the array of dogs. There's no need to first get all colors and then get all the unique items. You can get them in one loop.

```
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'
  }
]

function getUniqueColors(dogs) {
  const unique = new Set();
  for (const dog of dogs) {
    unique.add(dog.color);
  }
  return [...unique];
}


console.log(getUniqueColors(dogs));
```

In this code, you used a simple `for` loop. But you can easily simplify this action to a one liner with a `reduce()` function. Reduce functions are awesome and you'll love them, but they're a little more complicated. You'll get a chance to explore them thoroughly in Tip 26, Transform Array Data with reduce(), but here's a sample of how you can get the unique values in one line.

```
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'
    }
]

function getUniqueColors(dogs) {
   return [...dogs.reduce((colors, { color }) => colors.add(color), new Set())];
}

console.log(getUniqueColors(dogs));
```

By now you're probably feeling excited about all the new ways you can experiment with collections in your code. There are a few more you haven't touched, such as `WeakMap` and `WeakSet`, and you should try them out. The best place for JavaScript documentation is always The Mozilla Developer Network.

But that's enough talk about collections. It's time to start building things. The next step is learning to use control structures to handle conditional actions.

1

What will be the output of the code below?

```
let arr = [5,5,5,7,6,2,3,8,8];
let uniqueElem = [...new Set(arr)];
console.log(uniqueElem);
```

2

What will be the output of the code below?

```
let ans = new Set([1, 1, 2, 2, 3, 5 ,8])
console.log([...ans].map(x => x * 2));
```

Retake Quiz

In the next chapter, you’ll look at how you can apply the same standards of simplicity and readability to conditional statements.