Tip 10: Use Objects for Static Key-Value Lookups

In this tip, you'll learn why objects are the best collection for simple key-value lookups.

We'll cover the following

When to use a key-value collection?

You probably noticed that I love arrays. But they are not appropriate in many situations. As you saw, it is always possible to store any type of information in arrays—they are really are that flexible—but it can make things more confusing than necessary. And you often end up obscuring information more than you communicate it.

What if you had some data and you wanted to conditionally apply some colors in the UI. You want the data to be **red** if data was below threshold, **green** if everything is within normal range, and **blue** if some data was above a threshold. As usual, some very smart designer with a lot of training picked the absolute perfect shades of these colors (personally, I can never tell the difference, but that's why I don't design).

You could put the *hex* values in an array, but that doesn't really communicate much.

```
const colors = ['#d10202', '#19d836', '#0e33d8'];
```

What the heck does `#d10202` even mean? It happens to be a shade of **red**, but there's no way to know that without actually knowing it ahead of time. The problem is that this data is related—it's all colors—but not interchangeable. Unlike

an array of users where all users are structurally similar and one can be substituted for another, the different colors will serve different purposes

(indicating value to users). When a developer wants the *hex* code for red, they don't care what other values are in the collection. They don't need to know that red is the first or third color. In this case, a **key-value** collection will be more appropriate. You really need to give future developers a better idea of what the information means.

In cases where arrays aren't appropriate and you want a *key-value* collection, most developers reach for **objects**. And objects are great, but as you will see in upcoming tips, there are now more options for key-value collections.

Objects as key-value collections

The **TC39** committee added more options for collections because *objects* are complex. They can be *key-value* collections, which is how you will use them in this chapter, or they can be closer to classes with *constructors, methods, and properties*. Most things in JavaScript, including other collection types, are objects at their core.

This chapter will leave aside some of the complexities of *object properties, prototypes*, and the keyword `this` and instead look at how objects are used as *key-value* collections. The keyword `this`, for example, is a huge topic that's well covered by Kyle Simpson in You Don't Know JS: this & Object Prototypes.

When to use objects?

Now that you are thinking about objects primarily as collections competing against other collection types, such as `Map`, the new challenge is knowing when to chose plain objects deliberately, as the best solution for the problem, and not as a default.

As a rule, objects are great when you want to share unchanging structured key-value data, but are not appropriate for dynamic information that is updated frequently or unknown until runtime, as you will see in later tips.

Use cases

If you wanted to share your collection of colors, objects are a great choice. The data doesn't change. You wouldn't dynamically change the hex value for red. In this case, you can change your array of colors to an object by adding keys and wrapping the whole thing in curly braces. When you create an object this way, with key-values in curly braces, you are using object literal syntax.

with key-values in curly braces, you are using object literal syntax.

```
const colors = {
    red: '#d10202',
    green: '#19d836',
    blue: '#0e33d8'
}
```

When a future developer wants to get the proper color red, they don't need to know a position; they just call it directly: `colors.red`. Alternatively, they can use array syntax `colors['red']`. It's simple. That's why objects are so valuable for retrieving static information.

The key here is *static information*. Objects are not good for information that's continually *updated, looped over, altered, or sorted*. In those cases, use `Map`. Objects are a path to find information when you know where it will be. Config files are often objects because they are set up before runtime and are simple key-value stores of static information.

```
export const config = {
    endpoint: 'http://pragprog.com',
    key: 'secretkey',
}
```

But static objects can also be defined programmatically. For example, you can build an object in a function and then pass it to another function. The information is collected, sent, and then unpacked in another function. In this way, it's static because it is not mutated and updated.

The trick is that the data is set and then retrieved the same way every time. You are not mutating an existing object; you are creating a new object in each function. And more importantly, you know the key names when you are writing the code. You are not setting the keys using variables. The next function knows in advance what it will be getting.

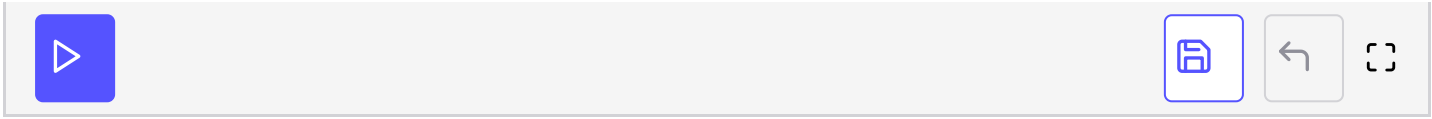
```
function getBill(item) {
    return {
        name: item.name,
        due: twoWeeksFromNow(),
        total: calculateTotal(item.price),
    };
}


const bill = getBill({ name: 'Room Cleaning', price: 30 });
```

```
function displayBill(bill) {
    return `Your total is ${bill.total} for ${bill.name} is due on ${bill.due}`;

}

console.log("Displaying the bill: " + displayBill(bill));
```



Advantages of objects

In the preceding example, an object is being used to add structure to information passed between objects. Instead of writing `displayBill()` as a function that takes each item as a parameter, you are passing the object, and the function is pulling out the values it needs.

This is where objects are far superior to other collections. Not only are they quick and clear, but with object destructuring, pulling data from objects is even quicker and cleaner than ever. Jump ahead to Tip 29, Access Object Properties with Destructuring, if you want to see it in action. Destructuring is part of the reason why nothing beats an object for a quick lookup.

But again, notice that the function is creating a new object. It's setting the information and then immediately retrieving it in a different function. It's not setting the information repeatedly. If you want to add lots of information to an object programmatically, other collections may be better suited for the task, such as the `Map` object, which we'll explore in Tip 13, Update Key-Value Data Clearly with Maps.

For now, you know that objects still play a huge role in JavaScript. You'll use them all the time when you're sharing information. In the next two tips, you'll look at a common use case: combining two similar objects together. And then you'll explore some other collections that you can use in place of objects.

Objects will come up more when you get to functions and classes, but for now, remember to keep their usage at a basic level and take a moment to consider other collections before creating an object.

Given a collection of movies and a list of their directors, would you store the movie-director pairs in an object or an array?

2

Given a class of students, you want to list the rank of each student from the lowest to the highest. Should you use an array or object to store the student ranks?

Retake Quiz

In the next tip, you'll dive into working with objects, beginning with making changes to objects without mutations.