Tip 29: Access Object Properties with Destructuring

In this tip, you'll learn how to pull information out of objects and arrays quickly with destructuring.

We'll cover the following

Managing excessive parameters

In the previous tip, you learned how to create default parameters, which are a great addition to the language, but they still have one big problem: *Parameters always have to be given in order*. If you wanted to specify the *third* parameter but you didn't care about the second, you'd still be forced to enter a value. *Default parameters aren't helpful if you want to skip a parameter*.

What about situations where you need a large number of arguments for a function? What about situations where you know that the needs of functions are likely to change? In JavaScript, most developers add extra arguments to an object and pass the object as the last parameter to a function.

Example

For example, what if you wanted to display a number of photos and needed to translate the values into an HTML string? Specifically, you want to include the *image, title, photographer, and location* in that order in your string, but you also want any additional information. Some photographs include *equipment, image type, lenses information*, and any other customizations. You don't know what it all will be, but you still want to display it.

There is a lot of information associated with a photograph. Passing that information as individual parameters would be excessive—*you could end up with about ten parameters*. Besides, the information is already structured. *What's the point in changing it?* Here's an example of some information about a photograph.

```
const landscape = {
    title: 'Landscape',
    photographer: 'Nathan',
    equipment: 'Canon',
    format: 'digital',
    src: '/landscape-nm.jpg',
    location: [32.7122222, -103.1405556],
};
```

Pulling information from objects

In this case, it makes sense to pass the whole photo object directly into a function. Of course, once you have it in the function, what do you do with it?

You can either pull the information directly from the object when needed using **dot** syntax—*photo.title*—or you can *assign* the information to *variables* and then use the variables later in the code.

Getting the values you know ahead of time is easy. The real trick is getting the **excess information**—*information that you don't know about ahead of time*. The only way to get it is to remove the *key-value* pairs you're using elsewhere and then keep whatever is leftover.

Fortunately, you're smart enough to know that you should copy the object before mutating it (good work). And after you copy it, you can delete the keys you don't need one at a time. The end result is a lot of object assignments for a very small action. Nearly two-thirds of the function is pulling information from an object.

```
const anonymous = {
  title: 'Kids',
  equipment: 'Nikon',
  src: '/garden.jpg',
  location: [38.9675338, -95.2614205],
};

const landscape = {
```

```
const landscape = {
  title: 'Landscape',
  photographer: 'Nathan',
  equipment: 'Canon',
  format: 'digital',
  src: '/landscape-nm.jpg',
  location: [32.7122222, -103.1405556],
};


function displayPhoto(photo) {
  const title = photo.title;
  const photographer = photo.photographer || 'Anonymous';
  const location = photo.location;
  const url = photo.src;
  const copy = { ...photo };
  delete copy.title;
  delete copy.photographer;
  delete copy.location;
  delete copy.src;
  const additional = Object.keys(copy).map(key => `${key}: ${copy[key]}`);
  return (`
<img alt="Photo of ${title} by ${photographer}" src="${url}" />
<div>${title}</div>
<div>${photographer}</div>
<div>Latitude: ${location[0]} </div>
<div>Longitude: ${location[1]} </div>
<div>${additional.join(' <br/> ')}</div>
`);
}

console.log(displayPhoto(landscape));
console.log(displayPhoto(anonymous));
```

Remember back in Tip 10, Use Objects for Static Key-Value Lookups, where you learned that objects are great for passing around static information? You're about to learn why.

Destructuring

In JavaScript, you can assign variables directly from an object using a process called **destructuring assignment**.

It works like this: *Destructuring allows you to create a variable with the same name as an object's key assigned with the value from the object*.

Example

As usual, it's always easier to see. In this case, you have an object with a key of `photographer` and you're going to create a variable named `photographer` from that object.

```
const landscape = {
    photographer: 'Nathan',
};
const { photographer } = landscape;
console.log(photographer);
```

Notice a few things.

- First, you still have to *declare* a variable type. As usual, you should prefer `const`.
- Second, the *assignment* variable *must* match the *key* in the object.
- Finally, it's set against the object. You are merely assigning a variable. The curly braces merely signal the value that variable should use is inside an object.

Destructuring: Setting a default value

That's the bare bones—*set a variable using the key*. Of course, nothing is ever that simple. What happens when a key doesn't exist? Well, in that case, the value is merely *undefined*, but you can also set a default value while destructuring.

```
const landscape = {
};
const { photographer = 'Anonymous', title } = landscape;
console.log(photographer);
console.log(title);
```

Destructuring: Using rest operator

At this point, you've caught up to regular parameters. You can set a variable from a key. You can set default values. But what do you do if you don't know the key name? How do you get the leftover information? Remember that you want any additional information from a photograph and you have no clue what that will be.

Good news: *Your favorite three dots are back*. You can collect any additional values into a new variable using *three dots* (`...`) followed by the *variable name*. When into a new variable using *three* dots (`...`) followed by the *variable name*. When

you use the three-dot syntax to collect information, it's no longer called the *spread* operator. It's called the **rest** operator, and you'll see more of it in upcoming tips.

You can name the variable anything you want. It doesn't need to match a key (in fact, it shouldn't match a key). And the value of the variable will be an object of the remaining key-value pairs.

```
const landscape = {
    photographer: 'Nathan',
    equipment: 'Canon',
    format: 'digital',
};

const {
    photographer,
    ...additional
} = landscape;
console.log(additional);
```

`photographer` is pulled out from the object, and the remaining fields go into a new object. You essentially *copied* the photograph object and deleted the `photographer` key.

Notice how the variable assignments are on different lines: photographer is on one line and ...additional is on the next. It's simply a style preference to keep things more readable. You can keep both assignments on the same line as you do above.

Destructuring: Assign a key to a different variable name

Now you can pull information from an object, assign default parameters, and collect additional key-values. As if that weren't enough to celebrate, you can also assign a key to a different variable name. This is useful in situations where the key name is taken by a previously defined variable or you just don't like the key name and you want something more expressive.

In the original code, you assign the information from `photo.src` to the variable name `url`. To accomplish that with destructuring, you simply put the *key name first* with a *colon* followed by the *value you want to assign it to*.

```
const landscape = {
    src: '/landscape-nm.jpg',
};
const { src: url } = landscape;
console.log(url);
console.log(src); //ReferenceError: src is not defined
```

You still must use the *key name* to signal which value you want to use, but you are not bound to that key name.

Destructuring with arrays

Finally, you can also use destructuring assignment with *arrays*, with one big exception: *Because there are no keys in arrays, you can use any variable name you want, but you must assign the information in order*. If you want to assign the *third* item to a variable, you must first assign the *previous two* values to a variable. Otherwise, it's simple. Destructuring is a great way to work with array pairs in a situation where the order denotes some information. For example, if you had an array of `latitude` and `longitude`, you'd always know the *first* value corresponds to `latitude` and the *second* to `longitude`.

```
const landscape = {
    location: [32.7122222, -103.1405556],
};
const { location } = landscape;
const [latitude, longitude] = location
console.log(latitude);
console.log(longitude);
```

Of course, in the preceding situation, you pulled out `location` *first* from an object and then `latitude` and `longitude` from the array. There's no need to make it a two-step process. You can combine the assignments during destructuring.

```
const landscape = {
    location: [32.7122222, -103.1405556],
};
const { location: [latitude, longitude] } = landscape;
console.log(latitude);
console.log(longitude);
```

All right, that was a lot to think about. But it really can clean things up fast. Remember the original function? Here it is with destructuring:

```
const anonymous = {
  title: 'Kids',
  equipment: 'Nikon',
  src: '/garden.jpg',
  location: [38.9675338, -95.2614205],
};

const landscape = {
  title: 'Landscape',
  photographer: 'Nathan',
  equipment: 'Canon',
  format: 'digital',
  src: '/landscape-nm.jpg',
  location: [32.7122222, -103.1405556],
};

function displayPhoto(photo) {
    const {
        title,
        photographer = 'Anonymous',
        location: [latitude, longitude],
        src: url,
        ...other
    } = photo;
    const additional = Object.keys(other).map(key => `${key}: ${other[key]}`);
    return (`
<img alt="Photo of ${title} by ${photographer}" src="${url}" />
<div>${title}</div>
<div>${photographer}</div>
<div>Latitude: ${latitude} </div>
<div>Longitude: ${longitude} </div>
<div>${additional.join(' <br/> ')}</div>
`);
}

console.log(displayPhoto(landscape));
console.log(displayPhoto(anonymous));
```

Looks good, doesn't it? But you're probably wondering what this is doing in a chapter about cleaning up parameters.

Destructuring & functions

The best part about destructuring is that you can move it right into the parameters of a function. The information will be assigned just as it was in the body of the function, but there's no need to declare the variable type. If you're curious, it'll be assigned with `let` so it's possible to reassign the variable.

In other words, you can clean up the original code even more:

```
const anonymous = {
  title: 'Kids',
  equipment: 'Nikon',
  src: '/garden.jpg',
  location: [38.9675338, -95.2614205],
};

const landscape = {
  title: 'Landscape',
  photographer: 'Nathan',
  equipment: 'Canon',
  format: 'digital',
  src: '/landscape-nm.jpg',
  location: [32.7122222, -103.1405556],
};

function displayPhoto({
  title,
  photographer = 'Anonymous',
  location: [latitude, longitude],
  src: url,
  ...other
}) {
  const additional = Object.keys(other).map(key => `${key}: ${other[key]}`);
  return (`
<img alt="Photo of ${title} by ${photographer}" src="${url}" />
<div>${title}</div>
<div>${photographer}</div>
<div>Latitude: ${latitude} </div>
<div>Longitude: ${longitude} </div>
<div>${additional.join(' <br/> ')}</div>
`);
}

console.log(displayPhoto(landscape));
console.log(displayPhoto(anonymous));
```

Notice that you still need the curly braces, but otherwise, everything is the same. Now when you call the function, you can just pass the object and everything will be assigned to the proper parameters: `displayPhoto(landscape)`.

Not only did you save yourself all the assignment problems, but by passing an object as a parameter, you don't have to worry about the order of the keyvalues.

And if you wanted to pull out another key-value, it's just a matter of adding the new variable to the destructuring. Say you wanted to assign equipment explicitly.

All you need to do is add in the new variable name in the list of variables and

you'll be good to go. There's no need to worry about other times when the function is called. If equipment isn't part of another object, it will merely be undefined.

That was probably a whirlwind, but it should give you a taste for how easily you can pull information from objects. The only downside is this only works on objects used as key-value pairs or object instances of a class.

Destructuring won't work on `Map`, which is fine because this is primarily useful when you're sending information between functions, meaning you shouldn't be looping or reassigning values. In other words, the data is static, so an object is a great choice.

As if that wasn't overwhelming enough, you've only seen half of it.

Q

Which of the following is the correct way of destructuring `name` in the object give below?

```
let person = {
  name: {
    first: 'Robin',
    last: 'Doe'
  },
  age: 20,
  country: 'USA'
}
```

Retake Quiz

In the next tip, you’re going to go in the other direction and put information back into an object.