# Destructuring for Easier Data Access

Object and array literals are two of the most frequently used notations in JavaScript, and thanks to the popular JSON data format, they've become a particularly important part of the language. It's quite common to define objects and arrays, and then systematically pull out relevant pieces of information from those structures. ECMAScript 6 simplifies this task by adding *destructuring*, which is the process of breaking a data structure down into smaller parts. This chapter shows you how to harness destructuring for both objects and arrays.

## Why is Destructuring Useful?

In ECMAScript 5 and earlier, the need to fetch information from objects and arrays could lead to a lot of code that looks the same, just to get certain data into local variables. For example:

```
let options = {
        repeat: true,
        save: false
    };

// extract data from the object
let repeat = options.repeat,
    save = options.save;
```

This code extracts the values of `repeat` and `save` from the `options` object and stores that data in local variables with the same names. While this code looks simple, imagine if you had a large number of variables to assign; you would have to assign them all one by one. And if there was a nested data structure to traverse to find the information instead, you might have to dig through the entire structure just to find one piece of data.

That's why ECMAScript 6 adds destructuring for both objects and arrays. When you break a data structure into smaller parts, getting the information you need out of it becomes much easier. Many languages implement destructuring with a minimal amount of syntax to make the process simpler to use. The ECMAScript 6 implementation actually makes use of syntax you're already familiar with: the syntax for object and array literals.

## Object Destructuring

Object destructuring syntax uses an object literal on the left side of an assignment operation. For example:

```
let node = {
        type: "Identifier",
        name: "foo"
    };

let { type, name } = node;

console.log(type);      // "Identifier"
console.log(name);      // "foo"
```

In this code, the value of `node.type` is stored in a variable called `type` and the value of `node.name` is stored in a variable called `name`. This syntax is the same as the object literal property initializer shorthand introduced in Chapter 4. The identifiers `type` and `name` are both declarations of local variables and the properties to read the value from on the `node` object.

A> #### Don't Forget the Initializer A> A>When using destructuring to declare variables using `var`, `let`, or `const`, you must supply an initializer (the value after the equals sign). The following lines of code will all throw syntax errors due to a missing initializer: A> A>`js A>// syntax error! A>var { type, name }; A> A>// syntax error! A>let { type, name }; A> A>// syntax error! A>const { type, name }; A>` A> A>While `const` always requires an initializer, even when using nondestructured variables, `var` and `let` only require initializers when using destructuring.

**Destructuring Assignment**

The object destructuring examples so far have used variable declarations. However, it's also possible to use destructuring in assignments. For instance, you may decide to change the values of variables after they are defined, as follows:

```
let node = {
        type: "Identifier",
        name: "foo"
    },
    type = "Literal",
    name = 5;

// assign different values using destructuring
({ type, name } = node);

console.log(type);      // "Identifier"
console.log(name);      // "foo"
```

In this example, `type` and `name` are initialized with values when declared, and then two variables with the same names are initialized with different values. The next line uses destructuring assignment to change those values by reading from the `node` object. Note that you must put parentheses around a destructuring assignment statement. That's because an opening curly brace is expected to a be a block statement, and a block statement cannot appear on the left side of an assignment. The parentheses signal that the next curly brace is not a block statement and should be interpreted as an expression, allowing the assignment to complete.

A destructuring assignment expression evaluates to the right side of the expression (after the `=`). That means you can use a destructuring assignment expression anywhere a value is expected. For instance, passing a value to a function:

```
let node = {
        type: "Identifier",
        name: "foo"
    },
    type = "Literal",
    name = 5;
```

```
function outputInfo(value) {
    console.log(value === node);
}

outputInfo({ type, name } = node);        // true

console.log(type);      // "Identifier"
console.log(name);      // "foo"
```

The `outputInfo()` function is called with a destructuring assignment expression. The expression evaluates to `node` because that is the value of the right side of the expression. The assignment to `type` and `name` both behave as normal and `node` is passed into `outputInfo()`.

W> An error is thrown when the right side of the destructuring assignment expression (the expression after `=`) evaluates to `null` or `undefined`. This happens because any attempt to read a property of `null` or `undefined` results in a runtime error.

**Default Values**

When you use a destructuring assignment statement, if you specify a local variable with a property name that doesn't exist on the object, then that local variable is assigned a value of `undefined`. For example:

```
let node = {
        type: "Identifier",
        name: "foo"
    };

let { type, name, value } = node;

console.log(type);      // "Identifier"
console.log(name);      // "foo"
console.log(value);     // undefined
```

This code defines an additional local variable called `value` and attempts to assign it a value. However, there is no corresponding `value` property on the `node` object, so the variable is assigned the value of `undefined` as expected.

You can optionally define a default value to use when a specified property doesn't exist. To do so, insert an equals sign (`=`) after the property name and specify the default value, like this:

```
let node = {
        type: "Identifier",
        name: "foo"
    };

let { type, name, value = true } = node;
```

```
console.log(type);      // "Identifier"
console.log(name);      // "foo"
console.log(value);     // true
```

In this example, the variable `value` is given `true` as a default value. The default value is only used if the property is missing on `node` or has a value of `undefined`. Since there is no `node.value` property, the variable `value` uses the default value. This works similarly to the default parameter values for functions, as discussed in Chapter 3.

#### Assigning to Different Local Variable Names

Up to this point, each example destructuring assignment has used the object property name as the local variable name; for example, the value of `node.type` was stored in a `type` variable. That works well when you want to use the same name, but what if you don't? ECMAScript 6 has an extended syntax that allows you to assign to a local variable with a different name, and that syntax looks like the object literal nonshorthand property initializer syntax. Here's an example:

```
let node = {
        type: "Identifier",
        name: "foo"
    };

let { type: localType, name: localName } = node;

console.log(localType);     // "Identifier"
console.log(localName);     // "foo"
```

This code uses destructuring assignment to declare the `localType` and `localName` variables, which contain the values from the `node.type` and `node.name` properties, respectively. The syntax `type: localType` says to read the property named `type` and store its value in the `localType` variable. This syntax is effectively the opposite of traditional object literal syntax, where the name is on the left of the colon and the value is on the right. In this case, the name is on the right of the colon and the location of the value to read is on the left.

You can add default values when using a different variable name, as well. The equals sign and default value are still placed after the local variable name. For example:

```
let node = {
        type: "Identifier"
    };

let { type: localType, name: localName = "bar" } = node;

console.log(localType);     // "Identifier"
console.log(localName);     // "bar"
```

Here, the `localName` variable has a default value of `"bar"`. The variable is assigned its default value because there's no `node.name` property.

So far, you've seen how to deal with destructuring of an object whose properties are primitive values. Object destructuring can also be used to retrieve values in nested object structures.

**Nested Object Destructuring**

By using a syntax similar to object literals, you can navigate into a nested object structure to retrieve just the information you want. Here's an example:

```
let node = {
        type: "Identifier",
        name: "foo",
        loc: {
            start: {
                line: 1,
                column: 1
            },
            end: {
                line: 1,
                column: 4
            }
        }
    };

let { loc: { start }} = node;

console.log(start.line);        // 1
console.log(start.column);      // 1
```

The destructuring pattern in this example uses curly braces to indicate that the pattern should descend into the property named `loc` on `node` and look for the `start` property. Remember from the last section that whenever there's a colon in a destructuring pattern, it means the identifier before the colon is giving a location to inspect, and the right side assigns a value. When there's a curly brace after the colon, that indicates that the destination is nested another level into the object.

You can go one step further and use a different name for the local variable as well:

```
let node = {
        type: "Identifier",
        name: "foo",
        loc: {
            start: {
                line: 1,
                column: 1
            },
            end: {
                line: 1,
```

```
                column: 4
            }
        }
    };

// extract node.loc.start
let { loc: { start: localStart }} = node;

console.log(localStart.line);   // 1
console.log(localStart.column); // 1
```

In this version of the code, `node.loc.start` is stored in a new local variable called `localStart`. Destructuring patterns can be nested to an arbitrary level of depth, with all capabilities available at each level.

Object destructuring is very powerful and has a lot of options, but array destructuring offers some unique capabilities that allow you to extract information from arrays.

A> #### Syntax Gotcha A> A>Be careful when using nested destructuring because you can inadvertently create a statement that has no effect. Empty curly braces are legal in object destructuring, however, they don't do anything. For example: A> A>`js A>// no variables declared! A>let { loc: {} } = node; A>` A> A>There are no bindings declared in this statement. Due to the curly braces on the right, `loc` is used as a location to inspect rather than a binding to create. In such a case, it's likely that the intent was to use `=` to define a default value rather than `:` to define a location. It's possible that this syntax will be made illegal in the future, but for now, this is a gotcha to look out for.

## Array Destructuring

Array destructuring syntax is very similar to object destructuring; it just uses array literal syntax instead of object literal syntax. The destructuring operates on positions within an array, rather than the named properties that are available in objects. For example:

```
let colors = [ "red", "green", "blue" ];

let [ firstColor, secondColor ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

Here, array destructuring pulls out the values `"red"` and `"green"` from the `colors` array and stores them in the `firstColor` and `secondColor` variables. Those values are chosen because of their position in the array; the actual variable names could be anything. Any items not explicitly mentioned in the destructuring pattern are ignored. Keep in mind that the array itself isn't changed in any way.

You can also omit items in the destructuring pattern and only provide variable names for the items you're interested in. If, for example, you just want the third value of an array, you don't need to supply variable names for the first and second items. Here's how that works:

```
let colors = [ "red", "green", "blue" ];

let [ , , thirdColor ] = colors;

console.log(thirdColor);        // "blue"
```

This code uses a destructuring assignment to retrieve the third item in `colors`. The commas preceding `thirdColor` in the pattern are placeholders for the array items that come before it. By using this approach, you can easily pick out values from any number of slots in the middle of an array without needing to provide variable names for them.

W> Similar to object destructuring, you must always provide an initializer when using array destructuring with `var`, `let`, or `const`.

**Destructuring Assignment**

You can use array destructuring in the context of an assignment, but unlike object destructuring, there is no need to wrap the expression in parentheses. For example:

```
let colors = [ "red", "green", "blue" ],
    firstColor = "black",
    secondColor = "purple";

[ firstColor, secondColor ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

The destructured assignment in this code works in a similar manner to the last array destructuring example. The only difference is that `firstColor` and `secondColor` have already been defined. Most of the time, that's probably all you'll need to know about array destructuring assignment, but there's a little bit more to it that you will probably find useful.

Array destructuring assignment has a very unique use case that makes it easier to swap the values of two variables. Value swapping is a common operation in sorting algorithms, and the ECMAScript 5 way of swapping variables involves a third, temporary variable, as in this example:

```
// Swapping variables in ECMAScript 5
let a = 1,
    b = 2,
    tmp;

tmp = a;
a = b;
b = tmp;
```

```
console.log(a);     // 2
console.log(b);     // 1
```

The intermediate variable `tmp` is necessary in order to swap the values of `a` and `b`. Using array destructuring assignment, however, there's no need for that extra variable. Here's how you can swap variables in ECMAScript 6:

```
// Swapping variables in ECMAScript 6
let a = 1,
    b = 2;

[ a, b ] = [ b, a ];

console.log(a);     // 2
console.log(b);     // 1
```

The array destructuring assignment in this example looks like a mirror image. The left side of the assignment (before the equals sign) is a destructuring pattern just like those in the other array destructuring examples. The right side is an array literal that is temporarily created for the swap. The destructuring happens on the temporary array, which has the values of `b` and `a` copied into its first and second positions. The effect is that the variables have swapped values.

W> Like object destructuring assignment, an error is thrown when the right side of an array destructured assignment expression evaluates to `null` or `undefined`.

**Default Values**

Array destructuring assignment allows you to specify a default value for any position in the array, too. The default value is used when the property at the given position either doesn't exist or has the value `undefined`. For example:

```
let colors = [ "red" ];

let [ firstColor, secondColor = "green" ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

In this code, the `colors` array has only one item, so there is nothing for `secondColor` to match. Since there is a default value, `secondColor` is set to `"green"` instead of `undefined`.

**Nested Destructuring**

You can destructure nested arrays in a manner similar to destructuring nested objects. By inserting another array pattern into the overall pattern, the destructuring will descend into a nested array, like this:

```
let colors = [ "red", [ "green", "lightgreen" ], "blue" ];

// later

let [ firstColor, [ secondColor ] ] = colors;

console.log(firstColor);        // "red"
console.log(secondColor);       // "green"
```

Here, the `secondColor` variable refers to the `"green"` value inside the `colors` array. That item is contained within a second array, so the extra square brackets around `secondColor` in the destructuring pattern are necessary. As with objects, you can nest arrays arbitrarily deep.

**Rest Items**

Chapter 3 introduced rest parameters for functions, and array destructuring has a similar concept called *rest items*. Rest items use the `...` syntax to assign the remaining items in an array to a particular variable. Here's an example:

```
let colors = [ "red", "green", "blue" ];

let [ firstColor, ...restColors ] = colors;

console.log(firstColor);        // "red"
console.log(restColors.length); // 2
console.log(restColors[0]);     // "green"
console.log(restColors[1]);     // "blue"
```

The first item in `colors` is assigned to `firstColor`, and the rest are assigned into a new `restColors` array. The `restColors` array, therefore, has two items: `"green"` and `"blue"`. Rest items are useful for extracting certain items from an array and keeping the rest available, but there's another helpful use.

A glaring omission from JavaScript arrays is the ability to easily create a clone. In ECMAScript 5, developers frequently used the `concat()` method as an easy way to clone an array. For example:

```
// cloning an array in ECMAScript 5
var colors = [ "red", "green", "blue" ];
var clonedColors = colors.concat();

console.log(clonedColors);      //"[red,green,blue]"
```

While the `concat()` method is intended to concatenate two arrays together, calling it without an argument returns a clone of the array. In ECMAScript 6, you can use rest items to achieve the same thing through syntax intended to function that way. It works like this:

```
// cloning an array in ECMAScript 6
let colors = [ "red", "green", "blue" ];
let [ ...clonedColors ] = colors;

console.log(clonedColors);      //"[red,green,blue]"
```

In this example, rest items are used to copy values from the `colors` array into the `clonedColors` array. While it's a matter of perception as to whether this technique makes the developer's intent clearer than using the `concat()` method, this is a useful ability to be aware of.

W> Rest items must be the last entry in the destructured array and cannot be followed by a comma. Including a comma after rest items is a syntax error.

## Mixed Destructuring

Object and array destructuring can be used together to create more complex expressions. In doing so, you are able to extract just the pieces of information you want from any mixture of objects and arrays. For example:

```
let node = {
        type: "Identifier",
        name: "foo",
        loc: {
            start: {
                line: 1,
                column: 1
            },
            end: {
                line: 1,
                column: 4
            }
        },
        range: [0, 3]
    };

let {
    loc: { start },
    range: [ startIndex ]
} = node;

console.log(start.line);        // 1
console.log(start.column);      // 1
console.log(startIndex);        // 0
```

This code extracts `node.loc.start` and `node.range[0]` into `start` and `startIndex`, respectively. Keep in mind that `loc:` and `range:` in the destructured pattern are just locations that correspond to properties in the `node` object. There is no part of `node` that cannot be extracted using destructuring when you use a mix of object and array destructuring. This approach is particularly useful for pulling values out of JSON configuration structures without navigating the entire structure.

## Destructured Parameters

Destructuring has one more particularly helpful use case, and that is when passing function arguments. When a JavaScript function takes a large number of optional parameters, one common pattern is to create an `options` object whose properties specify the additional parameters, like this:

```
// properties on options represent additional parameters
function setCookie(name, value, options) {

    options = options || {};

    let secure = options.secure,
        path = options.path,
        domain = options.domain,
        expires = options.expires;

    // code to set the cookie
}

// third argument maps to options
setCookie("type", "js", {
    secure: true,
    expires: 60000
});
```

Many JavaScript libraries contain `setCookie()` functions that look similar to this one. In this function, the `name` and `value` arguments are required, but `secure`, `path`, `domain`, and `expires` are not. And since there is no priority order for the other data, it's efficient to just have an `options` object with named properties, rather than list extra named parameters. This approach works, but now you can't tell what input the function expects just by looking at the function definition; you need to read the function body.

Destructured parameters offer an alternative that makes it clearer what arguments a function expects. A destructured parameter uses an object or array destructuring pattern in place of a named parameter. To see this in action, look at this rewritten version of the `setCookie()` function from the last example:

```
function setCookie(name, value, { secure, path, domain, expires }) {

    // code to set the cookie
}

setCookie("type", "js", {
    secure: true,
    expires: 60000
});
```

This function behaves similarly to the previous example, but now, the third argument uses destructuring to pull out the necessary data. The parameters outside the destructured parameter are clearly expected, and at the same time, it's clear to someone using `setCookie()` what options are available in terms of extra arguments.

And of course, if the third argument is required, the values it should contain are crystal clear. The destructured parameters also act like regular parameters in that they are set to `undefined` if they are not passed.

A>Destructured parameters have all of the capabilities of destructuring that you've learned so far in this chapter. You can use default values, mix object and array patterns, and use variable names that differ from the properties you're reading from.

### Destructured Parameters are Required

One quirk of using destructured parameters is that, by default, an error is thrown when they are not provided in a function call. For instance, this call to the `setCookie()` function in the last example throws an error:

```
// Error!
setCookie("type", "js");
```

The third argument is missing, and so it evaluates to `undefined` as expected. This causes an error because destructured parameters are really just a shorthand for destructured declaration. When the `setCookie()` function is called, the JavaScript engine actually does this:

```
function setCookie(name, value, options) {

    let { secure, path, domain, expires } = options;

    // code to set the cookie
}
```

Since destructuring throws an error when the right side expression evaluates to `null` or `undefined`, the same is true when the third argument isn't passed to the `setCookie()` function.

If you want the destructured parameter to be required, then this behavior isn't all that troubling. But if you want the destructured parameter to be optional, you can work around this behavior by providing a default value for the destructured parameter, like this:

```
function setCookie(name, value, { secure, path, domain, expires } = {}) {

    // ...
}
```

This example provides a new object as the default value for the third parameter. Providing a default value for the destructured parameter means that the `secure`, `path`, `domain`, and `expires` will all be `undefined` if the third argument to `setCookie()` isn't provided, and no error will be thrown.

### Default Values for Destructured Parameters

You can specify destructured default values for destructured parameters just as you would in destructured assignment. Just add the equals sign after the parameter and specify the default value. For example:

```
function setCookie(name, value,
    {
        secure = false,
        path = "/",
        domain = "example.com",
        expires = new Date(Date.now() + 360000000)
    } = {}
) {

    // ...
}
```

Each property in the destructured parameter has a default value in this code, so you can avoid checking to see if a given property has been included in order to use the correct value. Also, the entire destructured parameter has a default value of an empty object, making the parameter optional. This does make the function declaration look a bit more complicated than usual, but that's a small price to pay for ensuring each argument has a usable value.

## Summary

Destructuring makes working with objects and arrays in JavaScript easier. Using the familiar object literal and array literal syntax, you can pick data structures apart to get at just the information you're interested in. Object patterns allow you to extract data from objects while array patterns let you extract data from arrays.

Both object and array destructuring can specify default values for any property or item that is `undefined` and both throw errors when the right side of an assignment evaluates to `null` or `undefined`. You can also navigate deeply nested data structures with object and array destructuring, descending to any arbitrary depth.

Destructuring declarations use `var`, `let`, or `const` to create variables and must always have an initializer. Destructuring assignments are used in place of other assignments and allow you to destructure into object properties and already-existing variables.

Destructured parameters use the destructuring syntax to make "options" objects more transparent when used as function parameters. The actual data you're interested in can be listed out along with other named parameters. Destructured parameters can be array patterns, object patterns, or a mixture, and you can use all of the features of destructuring.