**JavaScript gets fancy – destructuring**

<https://dmitripavlutin.com/javascript-object-destructuring/> has written a beautiful description:

I advise you to read it online, but have reproduced it here in case the link gets broken.

Please read sections 1-3 and then summarize it for yourself in words.  
Then read sections 4 and 5, and again summarize it for yourself in words.  
Next, finish reading the article and summarizing.  
Finally, go to the end of this document and do the exercises there.



**How to Use Object Destructuring in JavaScript**

*Updated January 27, 2023*

[javascript](https://dmitripavlutin.com/tag/javascript/)[destructuring](https://dmitripavlutin.com/tag/destructuring/)

[48 Comments](https://dmitripavlutin.com/javascript-object-destructuring/#comments)

Object destructuring is a useful JavaScript feature to extract properties from objects and bind them to variables.

Even better, object destructuring can extract multiple properties in a single statement, can access properties from nested objects, and can set a default value if the property doesn't exist.

This post will help you understand how to use object destructuring in JavaScript.

**Table of Contents**

* [1. The need for object destructuring](https://dmitripavlutin.com/javascript-object-destructuring/#1-the-need-for-object-destructuring)
* [2. Extracting a property](https://dmitripavlutin.com/javascript-object-destructuring/#2-extracting-a-property)
* [3. Extracting multiple properties](https://dmitripavlutin.com/javascript-object-destructuring/#3-extracting-multiple-properties)
* [4. Default values](https://dmitripavlutin.com/javascript-object-destructuring/#4-default-values)
* [5. Aliases](https://dmitripavlutin.com/javascript-object-destructuring/#5-aliases)
* [6. Extracting properties from nested objects](https://dmitripavlutin.com/javascript-object-destructuring/#6-extracting-properties-from-nested-objects)
* [7. Extracting a dynamic name property](https://dmitripavlutin.com/javascript-object-destructuring/#7-extracting-a-dynamic-name-property)
* [8. Rest object after destructuring](https://dmitripavlutin.com/javascript-object-destructuring/#8-rest-object-after-destructuring)
* [9. Common use cases](https://dmitripavlutin.com/javascript-object-destructuring/#9-common-use-cases)
  + [9.1 Bind properties to variables](https://dmitripavlutin.com/javascript-object-destructuring/#91-bind-properties-to-variables)
  + [9.2 Function parameter destructuring](https://dmitripavlutin.com/javascript-object-destructuring/#92-function-parameter-destructuring)
* [10. Summary](https://dmitripavlutin.com/javascript-object-destructuring/#10-summary)

**1. The need for object destructuring**

Imagine you'd like to extract some properties of an object. In a pre-ES2015 environment, you'd need to write the following code:

var hero = {

name: 'Batman',

realName: 'Bruce Wayne'

};

var name = hero.name;

var realName = hero.realName;

console.log(name); *// => 'Batman',*

console.log(realName); *// => 'Bruce Wayne'*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/pr9o2fgx/)

The property hero.name value is assigned to the variable name. Same way hero.realName value is assigned to realName.

By writing var name = hero.name, you have to mention the name binding 2 times, and the same for realName. This way of accessing properties and assigning them to variables requires boilerplate code.

That's where the object destructuring syntax is useful: you can read a property and assign its value to a variable without duplicating the property name. What is more, you can read multiple properties from the same object in just one statement!

Let's refactor the above script and apply the object destructuring to access the properties name and realName:

const hero = {

name: 'Batman',

realName: 'Bruce Wayne'

};

const { name, realName } = hero;

console.log(name); *// => 'Batman',*

console.log(realName); *// => 'Bruce Wayne'*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/6jbehpf4/)

const { name, realName } = hero is an object destructuring assignment. This statement defines the variables name and realName, then assigns to them the values of properties hero.name and hero.realName correspondingly.

Compare two approaches to accessing the object properties:

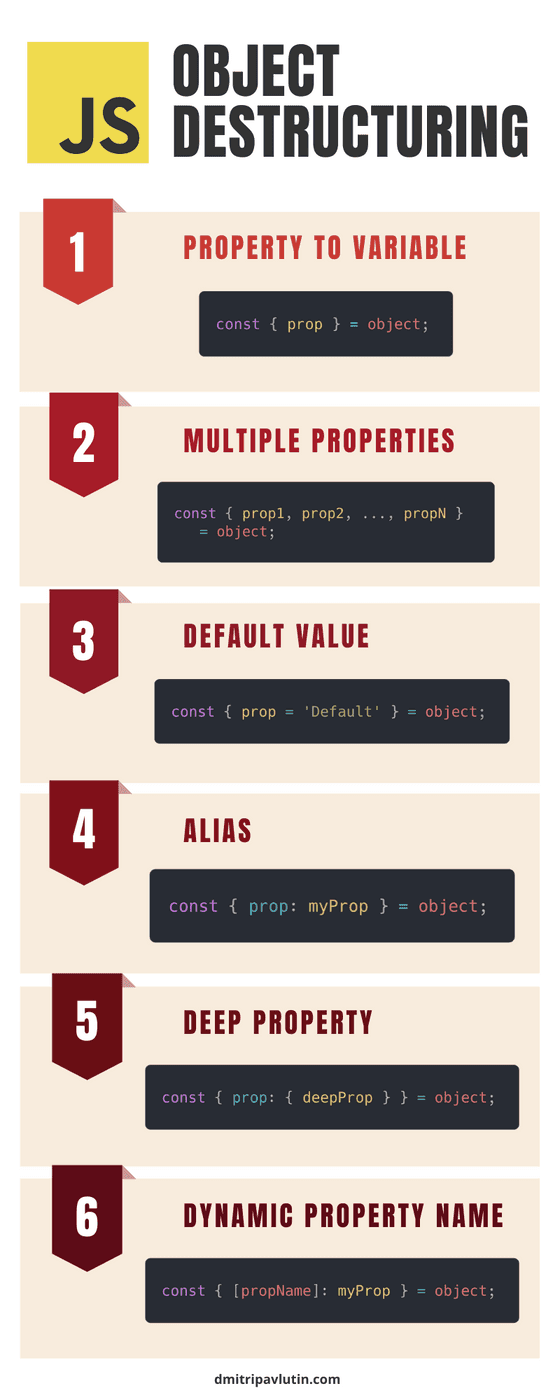
const name = hero.name;

const realName = hero.realName;

*// is equivalent to:*

const { name, realName } = hero;

it's visible that object destructuring is handier because neither the property names nor the object variable is duplicated.

[](https://dmitripavlutin.com/static/c6d9d0af2d1b6e8d0c88dad3f41f55f4/36e56/javascript-object-destructuring-infographic-2.png)

**2. Extracting a property**

The syntax of object destructuring is pretty simple:

const { identifier } = expression;

Where identifier is the name of the property to access and expression should evaluate to an object. After the destructuring, the variable identifier contains the property value.

Here's the equivalent code using a [property accessor](https://dmitripavlutin.com/access-object-properties-javascript/#1-dot-property-accessor):

const identifier = expression.identifier;

Let's try object destructuring in practice:

const hero = {

name: 'Batman',

realName: 'Bruce Wayne'

};

const { name } = hero;

console.log(name); *// => 'Batman'*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/Lxs8jk3t/)

The statement const { name } = hero defines the variable name and initializes it with the value of hero.name property.

**3. Extracting multiple properties**

To destructure the object into multiple properties, enumerate as many properties as you like adding commas , in-between:

const { identifier1, identifier2, ..., identifierN } = expression;

Where identifier1, ..., identifierN are names of properties to access, and expression should evaluate to an object. After the destructuring, the variables identifier1, ..., identifierN contain corresponding properties values.

Here's the equivalent code:

const identifier1 = expression.identifier1;

const identifier2 = expression.identifier2;

*// ...*

const identifierN = expression.identifierN;

Let's take a look again at the example from the first section, where 2 properties are extracted:

const hero = {

name: 'Batman',

realName: 'Bruce Wayne'

};

const { name, realName } = hero;

console.log(name); *// => 'Batman',*

console.log(realName); *// => 'Bruce Wayne'*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/6jbehpf4/)

const { name, realName } = hero creates 2 variables name and realName assigned with values of corresponding properties hero.name and hero.realName.

**4. Default values**

If the destructured object doesn't have the property specified in the destructuring assignment, then the variable is assigned with undefined. Let's see how it happens:

const hero = {

name: 'Batman',

realName: 'Bruce Wayne'

};

const { enemies } = hero;

console.log(enemies); *// => undefined*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/8spL1mkc/)

After destructuring the variable enemies is undefined because the property enemies doesn't exist in the object hero.

Fortunately, you can set a default value if the property doesn't exist in the destructured object:

const { identifier = defaultValue } = expression;

Where identifier is the name of the property to access and expression should evaluate to an object. After destructuring, the variable identifier contains the property value or is assigned with defaultValue if the property identifier doesn't exist.

Here's the equivalent code:

const identifier = expression.identifier === undefined ?

defaultValue : expression.identifier;

Let's change the previous code sample, and use the default value feature:

const hero = {

name: 'Batman',

realName: 'Bruce Wayne'

};

const { enemies = ['Joker'] } = hero;

console.log(enemies); *// => ['Joker']*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/vt8gcL2d/)

Now, instead of being undefined, the variable enemies defaults to ['Joker'].

**5. Aliases**

To create variables of different names than the properties you can use the aliasing feature of object destructuring.

const { identifier: aliasIdentifier } = expression;

identifier is the name of the property to access, aliasIdentifier is the variable name, and expression should evaluate to an object. After destructuring, the variable aliasIdentifier contains the property value.

The equivalent code:

const aliasIdentifier = expression.identifier;

Here's an example of an object destructuring alias feature:

const hero = {

name: 'Batman',

realName: 'Bruce Wayne'

};

const { realName: secretName } = hero;

console.log(secretName); *// => 'Bruce Wayne'*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/pbuehm5t/)

Looking at const { realName: secretName } = hero, the destructuring defines a new variable secretName (alias variable) and assigns to it the value of hero.realName.

**6. Extracting properties from nested objects**

In the previous examples, the objects were plain: the properties have primitive data types (e.g. strings).

But objects can be nested in other objects. In other words, some properties can contain objects.

In such a case, you still can use the object destructuring and access properties from deep. Here's the basic syntax:

const { nestedObjectProp: { identifier } } = expression;

nestedObjectProp is the name of the property that holds a nested object. identifier is the property name to access from the nested object. expression should evaluate to the destructured object.

After destructuring, the variable identifier contains the property value of the nested object.

The above syntax is equivalent to:

const identifier = expression.nestedObjectProp.identifier;

The level of nesting to extract properties from is unlimited. If you want to extract properties from deep, just add more nested curly braces:

const { propA: { propB: { propC: { .... } } } } = object;

For example, the object hero contains a nested object { city: 'Gotham'}.

const hero = {

name: 'Batman',

realName: 'Bruce Wayne',

address: {

city: 'Gotham'

}

};

*// Object destructuring:*

const { address: { city } } = hero;

console.log(city); *// => 'Gotham'*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/2hpvoazL/)

The object destructuring const { address: { city } } = hero accesses the property city from the nested object and creates a variable city having the property value.

**7. Extracting a dynamic name property**

You can extract into variables properties with a dynamic name (the property name is known at runtime):

const { [propName]: identifier } = expression;

propName expression should evaluate to a property name (usually a string), and the identifier should indicate the variable name created after destructuring. expression should evaluate to the object you'd like to destructure.

An equivalent code without object destructuring:

const identifier = expression[propName];

Let's look at an example where prop holds the property name:

const hero = {

name: 'Batman',

realName: 'Bruce Wayne'

};

const prop = 'name';

const { [prop]: name } = hero;

console.log(name); *// => 'Batman'*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/31w04z6p/)

const { [prop]: name } = hero is an object destructuring that assigns to variable name the value hero[prop], where prop is a variable holding the property name.

**8. Rest object after destructuring**

The rest syntax is useful to collect the remaining properties after destructuring:

const { identifier, ...rest } = expression;

Where identifier is the name of the property to access and expression should evaluate to an object. After destructuring, the variable identifier contains the property value. rest variable is a plain object with the remaining properties.

For example, let's extract the property name, but collect the rest of the properties into a variable rest:

const hero = {

name: 'Batman',

realName: 'Bruce Wayne',

company: 'WayneCorp'

};

const { name, ...rest } = hero;

console.log(rest); *// => { realName: 'Bruce Wayne', company: 'WayneCorp' }*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/yuab78Lr/2/)

The destructuring const { name, ...realHero } = hero extracts the property name. Also, the remaining properties (realName and company) are collected into rest.

**9. Common use cases**

**9.1 Bind properties to variables**

As seen in many examples before, the object destructuring binds property values to variables.

The object destructuring can assign values to variables declared using const, let, and var. Or even assign to an already existing variable.

For example, here's how to destructure using let statement:

*// let*

const hero = {

name: 'Batman',

};

let { name } = hero;

console.log(name); *// => 'Batman'*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/3pv740e9/)

How to destructure using var statement:

*// var*

const hero = {

name: 'Batman',

};

var { name } = hero;

console.log(name); *// => 'Batman'*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/pkoqud4x/)

And how to destructure to an already declared variable:

*// existing variable*

let name;

const hero = {

name: 'Batman',

};

({ name } = hero);

console.log(name); *// => 'Batman'*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/6bc4oL0a/)

I find it satisfying to combine for..of cycle with object destructuring to extract the property right away:

const heroes = [

{ name: 'Batman' },

{ name: 'Joker' }

];

for (const { name } of heroes) {

console.log(name); *// logs 'Batman', 'Joker'*

}

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/vb9z6qhu/)

**9.2 Function parameter destructuring**

Object destructuring can be placed anywhere where an assignment happens.

For example, you could destruct an object right inside the parameter of a function:

const heroes = [

{ name: 'Batman' },

{ name: 'Joker' }

];

const names = heroes.map(

function({ *name* }) {

return name;

}

);

console.log(names); *// => ['Batman', 'Joker']*

[Open the demo.](https://jsfiddle.net/dmitri_pavlutin/gmtbs5ny/)

function({ name }) destructures the function parameter and creates a variable name holding the value of name property.

**10. Summary**

Object destructuring is a powerful feature to extract properties from an object and bind these values to variables.

I like object destructuring due to the concise syntax and the ability to extract multiple variables in one statement.

*Have questions regarding object destructuring? Ask in a comment below!*

**Like the post? Please share!**

[Suggest Improve](https://github.com/panzerdp/dmitripavlutin.com/edit/master/posts/079-object-destructuring/index.md)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Exercises -- Now for you to practice:**

**Fill in this chart**

**Destructuring syntax**

|  |  |
| --- | --- |
| **Code** | **Explanation** |
| **const {prop} = object;** | The constant prop gets assigned the value of object.prop |
| **const {prop1, …propN}   = object;** |  |
| **const {prop =’default’}   = object;** |  |
| **const {prop: myProp}  = object;** |  |
| **const {prop:{deeper}}  = object;** |  |
| **const {[propName]:myProp = object;** |  |

**What do the following do: (Check it with console.log of the expected values)**

const user1 = {name:'Lynn Perry Wooten',  
 address: {street:'300 The Fenway',  
 city:'Boston',  
 state:'MA',  
 zip:'02115'  
 },  
 title:'President'  
}

let {name, title} = user1;  
let {salutation = 'Dr.'} = user1;  
let {address:{city}} = user1;  
const {name: userName} = user1;  
NOTE: It appears to not be possible to use both nested destructuring and aliasing – i.e. I was not able to destructure as we did in ***let {address:{city}} = user1;***  and also alias it the way did in the line just above.  
  
**Comment—there is also a short hand syntax to “undo” the destructuring. Here we take variables and store their values in a new object that uses the names of the variables as the keys:**let university = 'Simmoms University';  
let leader = user1;  
const Simmons = {university, leader}; //new object with keys university, leader  
If we examine Simmons we find: **console.log(Simmons);  
 //VM650:1 {university: 'Simmoms University', leader: {…}}**

**console.log(Simmons.leader);**

**//VM769:1 {name: 'Lynn Perry Wooten', address: {…}, title: 'President'}**

**You will see this short hand used in defining properties in objects.  
  
There is also array destructuring  
Read about it at** [**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring\_assignment**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) **and bring your questions to class.** (Note: This also makes use of the rest operator, which is discussed in ***JavaScript gets fancy – spread and rest operators.***) ***Warning***When this page comes to use destructuring and the rest operator there are two examples  
First Example   
 const { a, ...others } = { a: 1, b: 2, c: 3 };  
 console.log(others); // { b: 2, c: 3 }  
This gives you exactly what you would expect.  
  
Second Example  
 const {a, b, …{length}) = [1, 2, 3]  
 console.log(a, b, length); //1 2 1  
 To see what is really going on here, modify this a bit to   
 const {a, b, …{length}) = [10, 20, 30, 40]  
 console.log(a, b, length); //10 20 2  
  
 Okay: a and b are as expected, but look at the red parens ---- we have a nested destructuring of the   
 (unnamed) array [10, 20, 30, 40] and so we are getting unnamed\_array.length  
  
 To make this even clearer suppose we had  
 const A = [10, 20, 30, 40];  
 const {a, b, …{length}) = A  
 console.log(a, b, length); //10 20 2

**What does the following do:**  
   
{ name: 'Alice', age: 30 }  
{ name: 'Bob', age: 25 }  
];  
  
for (const { name, age } of users) {  
console.log(`${name} is ${age} years old`);  
}

**What does the following do:**

Users is an array of objects, each of the form   
 {name: *someName,* email: *someEmail,* password: *somePassword}*Someone has tried to log on to your site with emailTried and passwordTried

found = false;  
continue = true;  
let j = 0;  
while (continue) {  
 if (emailTried = Users[j].email)  
 {if (passwordTried = Users.[j].password)  
 { let {name} = Users[j];  
 //display welcome message  
 found = true;  
 continue = false;  
 } else {  
 // display invalid password message  
 //return user to login screen  
 }  
 } else { //this is not Users[j]  
 j++;  
 continue = (j < Users.length);  
 }  
 }  
 if (!found) { //display invalid user name message }  
  
**What does the following do:**

In h.s. we all used the Periodic Table (e.g. at <https://en.wikipedia.org/wiki/Periodic_table> ), but you can find it as an array of json elements at <https://github.com/Bowserinator/Periodic-Table-JSON/blob/master/PeriodicTableJSON.json> Of course, as arrays start indexing at 0, the element with atomic number 1 (which is hydrogen) is at index 0, the element with atomic number 2 (which is helium) is at index 1, etc.   
Suppose you have downloaded this array and stored it in elements.  
You have a variable atomicNumber  
  
let {name, number, melt} = elements[atomicNumber – 1];  
   
**Write code to do the following:**Using the elements array above, modify the code so that the variables you set up are named   
elementName, elementNumber, and elementMeltingPoint.

Using the elements array above, suppose you have been given an array ofInterest which has a bunch of atomic numbers. Create a new array ofInterestDetails which, for each of those elements holds an object with the atomic name, symbol, number and atomic mass and also displays those items in a pop-up in a nice sentence. You should use template literals to form the sentence.

Now modify your code to include the year each element was discovered.

When we looked at the nullish coalescing operator we learned how to handle null values. Modify your code so that the pop-up includes the name of the discoverer or ‘unknown’ if that value is null.

Getting even fancier – go thru

<https://javascript.plainenglish.io/harnessing-javascript-object-destructuring-and-spread-syntax-use-cases-and-best-practices-257c731122ec>

**I also need an example or two from importing modules**

From Stackoverflow:

What is the difference between const and const {} in JavaScript  
Ask Question  
Asked 6 years, 3 months ago  
Modified 27 days ago  
Viewed 85k times  
  
150  
  
  
When I study electron, I found 2 ways of getting BrowserWindow object.  
  
const {BrowserWindow} = require('electron')  
and  
  
const electron = require('electron')  
const BrowserWindow = electron.BrowserWindow  
What is the difference between const and const {} in JavaScript?  
  
I can't understand why the const {} can work. Do I miss anything important about JS?  
  
javascriptnode.jsdestructuring  
Share  
Edit  
Follow  
Flag  
edited Mar 6 at 20:35  
  
customcommander  
17k55 gold badges5858 silver badges7878 bronze badges  
asked Dec 9, 2016 at 10:43  
  
Kevin00000000  
1,68922 gold badges1212 silver badges1111 bronze badges  
Add a comment  
4 Answers  
Sorted by:  
  
  
239  
  
  
The two pieces of code are equivalent but the first one is using the ES6 destructuring assignment to be shorter.