Destructuring assignment

The two most used data structures in JavaScript are `Object` and `Array` .

Objects allow us to pack many pieces of information into a single entity and arrays allow us to store ordered collections. So we can make an object or an array and handle it as a single entity, or maybe pass it to a function call.

Destructuring assignment is a special syntax that allows us to “unpack” arrays or objects into a bunch of variables, as sometimes they are more convenient. Destructuring also works great with complex functions that have a lot of parameters, default values, and soon we’ll see how these are handled too.

Array destructuring

An example of how the array is destructured into variables:

```
// we have an array with the name and surname
let arr = ["Ilya", "Kantor"]

// destructuring assignment
let [firstName, surname] = arr;

alert(firstName); // Ilya
alert(surname);  // Kantor
```

Now we can work with variables instead of array members.

It looks great when combined with `split` or other array-returning methods:

```
let [firstName, surname] = "Ilya Kantor".split(' ');
```

“Destructuring” does not mean “destructive”.

It’s called “destructuring assignment,” because it “destructurizes” by copying items into variables. But the array itself is not modified.

It’s just a shorter way to write:

```
// let [firstName, surname] = arr;
let firstName = arr[0];
let surname = arr[1];
```

Ignore first elements

Unwanted elements of the array can also be thrown away via an extra comma:

```
// first and second elements are not needed
let [, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert( title ); // Consul
```

In the code above, although the first and second elements of the array are skipped, the third one is assigned to `title` , and the rest are also skipped.

Works with any iterable on the right-side

...Actually, we can use it with any iterable, not only arrays:

```
let [a, b, c] = "abc"; // ["a", "b", "c"]
let [one, two, three] = new Set([1, 2, 3]);
```

Assign to anything at the left-side

We can use any "assignables" at the left side.

For instance, an object property:

```
let user = {};
[user.name, user.surname] = "Ilya Kantor".split(' ');

alert(user.name); // Ilya
```

Looping with .entries()

In the previous chapter we saw the Object.entries(obj) method.

We can use it with destructuring to loop over keys-and-values of an object:

```
let user = {
  name: "John",
  age: 30
};

// loop over keys-and-values
for (let [key, value] of Object.entries(user)) {
  alert(`${key}:${value}`); // name:John, then age:30
}
```

...And the same for a map:

```
let user = new Map();
user.set("name", "John");
user.set("age", "30");

for (let [key, value] of user.entries()) {
  alert(`${key}:${value}`); // name:John, then age:30
}
```

The rest '...'

If we want not just to get first values, but also to gather all that follows – we can add one more parameter that gets "the rest" using three dots `"..."`:

```
let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(name1); // Julius
alert(name2); // Caesar

// Note that type of `rest` is Array.
alert(rest[0]); // Consul
alert(rest[1]); // of the Roman Republic
alert(rest.length); // 2
```

The value of `rest` is the array of the remaining array elements. We can use any other variable name in place of `rest`, just make sure it has three dots before it and goes last in the destructuring assignment.

Default values

If there are fewer values in the array than variables in the assignment, there will be no error. Absent values are considered undefined:

```
let [firstName, surname] = [];

alert(firstName); // undefined
alert(surname); // undefined
```

If we want a “default” value to replace the missing one, we can provide it using `=` :

```
// default values
let [name = "Guest", surname = "Anonymous"] = ["Julius"];

alert(name);    // Julius (from array)
alert(surname); // Anonymous (default used)
```

Default values can be more complex expressions or even function calls. They are evaluated only if the value is not provided.

For instance, here we use the `prompt` function for two defaults. But it will run only for the missing one:

```
// runs only prompt for surname
let [name = prompt('name?'), surname = prompt('surname?')] = ["Julius"];

alert(name);    // Julius (from array)
alert(surname); // whatever prompt gets
```

Object destructuring

The destructuring assignment also works with objects.

The basic syntax is:

```
let {var1, var2} = {var1:…, var2…}
```

We have an existing object at the right side, that we want to split into variables. The left side contains a “pattern” for corresponding properties. In the simple case, that’s a list of variable names in `{...}` .

For instance:

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

let {title, width, height} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

Properties `options.title` , `options.width` and `options.height` are assigned to the corresponding variables. The order does not matter. This works too:

```
// changed the order of properties in let {...}
let {height, width, title} = { title: "Menu", height: 200, width: 100 }
```

The pattern on the left side may be more complex and specify the mapping between properties and variables.

If we want to assign a property to a variable with another name, for instance, `options.width` to go into the variable named `w` , then we can set it using a colon:

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};

// { sourceProperty: targetVariable }
let {width: w, height: h, title} = options;

// width -> w
// height -> h
// title -> title

alert(title);  // Menu
alert(w);      // 100
alert(h);      // 200
```

The colon shows "what : goes where". In the example above the property `width` goes to `w`, property `height` goes to `h`, and `title` is assigned to the same name.

For potentially missing properties we can set default values using `"="`, like this:

```
let options = {
  title: "Menu"
};

let {width = 100, height = 200, title} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

Just like with arrays or function parameters, default values can be any expressions or even function calls. They will be evaluated if the value is not provided.

The code below asks for width, but not the title.

```
let options = {
  title: "Menu"
};

let {width = prompt("width?"), title = prompt("title?")} = options;

alert(title);  // Menu
alert(width);  // (whatever you the result of prompt is)
```

We also can combine both the colon and equality:

```
let options = {
  title: "Menu"
};

let {width: w = 100, height: h = 200, title} = options;

alert(title);  // Menu
alert(w);      // 100
alert(h);      // 200
```

The rest operator

What if the object has more properties than we have variables? Can we take some and then assign the "rest" somewhere?

The specification for using the rest operator (three dots) here is almost in the standard, but most browsers do not support it yet.

It looks like this:

```
let options = {
  title: "Menu",
  height: 200,
  width: 100
};

let {title, ...rest} = options;

// now title="Menu", rest={height: 200, width: 100}
alert(rest.height);  // 200
alert(rest.width);   // 100
```

Gotcha without `let`

In the examples above variables were declared right before the assignment: `let {…} = {…}` . Of course, we could use existing variables too. But there's a catch.

This won't work:

```
let title, width, height;

// error in this line
{title, width, height} = {title: "Menu", width: 200, height: 100};
```

The problem is that JavaScript treats `{...}` in the main code flow (not inside another expression) as a code block. Such code blocks can be used to group statements, like this:

```
{
  // a code block
  let message = "Hello";
  // ...
  alert( message );
}
```

To show JavaScript that it's not a code block, we can wrap the whole assignment in parentheses `(...)` :

```
let title, width, height;

// okay now
({title, width, height} = {title: "Menu", width: 200, height: 100});

alert( title ); // Menu
```

Nested destructuring

If an object or an array contain other objects and arrays, we can use more complex left-side patterns to extract deeper portions.

In the code below `options` has another object in the property `size` and an array in the property `items` . The pattern at the left side of the assignment has the same structure:

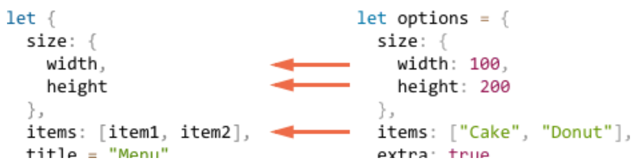
```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true    // something extra that we will not destruct
};

// destructuring assignment on multiple lines for clarity
let {
  size: { // put size here
    width,
    height
  },
  items: [item1, item2], // assign items here
  title = "Menu" // not present in the object (default value is used)
} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
alert(item1);  // Cake
alert(item2);  // Donut
```

The whole `options` object except `extra` that was not mentioned, is assigned to corresponding variables.

Note that `size` and `items` itself is not destructured.



```
let {
  size: {
    width,
    height
  },
  items: [item1, item2],
  title = "Menu"
```

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
}                                  }
```

Finally, we have `width`, `height`, `item1`, `item2` and `title` from the default value.

That often happens with destructuring assignments. We have a complex object with many properties and want to extract only what we need.

Even here it happens:

```
// take size as a whole into a variable, ignore the rest
let { size } = options;
```

Smart function parameters

There are times when a function may have many parameters, most of which are optional. That's especially true for user interfaces. Imagine a function that creates a menu. It may have a width, a height, a title, items list and so on.

Here's a bad way to write such function:

```
function showMenu(title = "Untitled", width = 200, height = 100, items = []) {
  // ...
}
```

In real-life, the problem is how to remember the order of arguments. Usually IDEs try to help us, especially if the code is well-documented, but still… Another problem is how to call a function when most parameters are ok by default.

Like this?

```
showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])
```

That's ugly. And becomes unreadable when we deal with more parameters.

Destructuring comes to the rescue!

We can pass parameters as an object, and the function immediately destructurizes them into variables:

```
// we pass object to function
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

// ...and it immediately expands it to variables
function showMenu({title = "Untitled", width = 200, height = 100, items = []}) {
  // title, items – taken from options,
  // width, height – defaults used
  alert( `${title} ${width} ${height}` ); // My Menu 200 100
  alert( items ); // Item1, Item2
}

showMenu(options);
```

We can also use more complex destructuring with nested objects and colon mappings:

```
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

function showMenu({
  title = "Untitled",
  width: w = 100,  // width goes to w
  height: h = 200, // height goes to h
  items: [item1, item2] // items first element goes to item1, second to item2
}) {
  alert( `${title} ${w} ${h}` ); // My Menu 100 200
  alert( item1 ); // Item1
  alert( item2 ); // Item2
}

showMenu(options);
```

The syntax is the same as for a destructuring assignment:

```
function({
  incomingProperty: parameterName = defaultValue
  ...
})
```

Please note that such destructuring assumes that `showMenu()` does have an argument. If we want all values by default, then we should specify an empty object:

```
showMenu({});


showMenu(); // this would give an error
```

We can fix this by making `{}` the default value for the whole destructuring thing:

```
// simplified parameters a bit for clarity
function showMenu({ title = "Menu", width = 100, height = 200 } = {}) {
  alert( `${title} ${width} ${height}` );
}

showMenu(); // Menu 100 200
```

In the code above, the whole arguments object is `{}` by default, so there's always something to destructurize.

Summary

- Destructuring assignment allows for instantly mapping an object or array onto many variables.
- The object syntax:

  ```
  let {prop : varName = default, ...} = object
  ```

  This means that property `prop` should go into the variable `varName` and, if no such property exists, then the `default` value should be used.
- The array syntax:

  ```
  let [item1 = default, item2, ...rest] = array
  ```

  The first item goes to `item1`; the second goes into `item2`, all the rest makes the array `rest`.
- For more complex cases, the left side must have the same structure as the right one.

Tasks

Destructuring assignment

importance: 5

We have an object:

```
let user = {
  name: "John",
  years: 30
};
```

Write the destructuring assignment that reads:

- `name` property into the variable `name`.
- `years` property into the variable `age`.
- `isAdmin` property into the variable `isAdmin` (false if absent)

The values after the assignment should be:

```
let user = { name: "John", years: 30 };

// your code to the left side:
// ... = user

alert( name ); // John
alert( age ); // 30
alert( isAdmin ); // false
```

solution

```
let user = {
  name: "John",
  years: 30
};

let {name, years: age, isAdmin = false} = user;

alert( name ); // John
alert( age ); // 30
alert( isAdmin ); // false
```

The maximal salary

importance: 5

There is a `salaries` object:

```
let salaries = {
  "John": 100,
  "Pete": 300,
  "Mary": 250
};
```

Create the function `topSalary(salaries)` that returns the name of the top-paid person.

- If `salaries` is empty, it should return `null` .
- If there are multiple top-paid persons, return any of them.

P.S. Use `Object.entries` and destructuring to iterate over key/value pairs.

Open a sandbox with tests.

solution

Open the solution with tests in a sandbox.

Previous lesson Next lesson

Share Tutorial map

Comments

- You're welcome to post additions, questions to the articles and answers to them.