Destructuring in JavaScript

§

Made with Obsidian

Type deep-dive | Category computer-science | Technologies JavaScript, VS Code

Website Post Link

Destructuring is a very powerful & widely-used syntactic construct that provides us with the ability to decompose a given object or structure into smaller components. This is specially useful when we have, for example, objects with nested elements and would like to extract all or some of them and assign them to variables, without having to write multiple lines of code to do so.

Destructuring can be used in 3 main JavaScript structures:

- Arrays
- Objects
- Function parameters

In this Deep Dive, we'll discuss what destructuring is, how can it be applied to nested structures such as multi-dimensional arrays and objects, how can it be applied to function parameters, and some of its main use cases. We'll also introduce some examples that will hopefully help illustrate this concept better.

We'll be using JavaScript scripts which can be found in the Deep Dive Repo.

§

Table of Contents

- What is destructuring?
- Why is it useful?
  - Working with APIs
  - Simplified code
  - Power
- General syntax
  - Arrays
  - Objects
  - Function parameters
- Main features
  - Skipping elements
  - Using starter values
  - Renaming variables
- Some practical examples

    - Deeply nested objects
    - Extracting intermittently from an unknown array
    - An object containing multiple argument groups
- Conclusions
- References
- Copyright

§

What is destructuring?

Simply put, destructuring, also called matching, is a syntactic construct introduced in ES6 (*ECMAScript 2015*), that lets us deconstruct structures, assigning the nested components to variables in one single line of code.

We can see destructuring as a two-step process:

1. Destructuring the structure
2. Assigning the resulting components to variables

This might sound really simple, but it's actually an extremely powerful feature. Let us understand why.

Let us imagine we have an object with multiple key-value pairs, and some of the values are arrays. We want to take that object, go for some of the values that have array structures, and assign a variable for each element inside these arrays.

Previous to ES6, this was possible by indexing:

Code

```
const myObj = {
    fName: "Emma",
    lName: "Miller",
    interests: ["Reading", "Writing", "Sleeping"]
};

// Indexing method
const mainInterest = myObj.interests[0];
const secondaryInterest = myObj.interests[1];
const tertiaryInterest = myObj.interests[2];

console.log(mainInterest, secondaryInterest, tertiaryInterest);
```

Output

```
Reading Writing Sleeping
```

This is a complete waste of resources and space, since we're instancing our object on each line, and simply changing the index.

ES6 then introduces a new & more efficient way to handle these types of operations:

Code

```
// Destructuring (ES6) method
const [Int1, Int2, Int3] = myObj.interests
console.log(Int1, Int2, Int3);
```

Which gives the exact same result:

Output

```
Reading Writing Sleeping
```

This was just an example using arrays, but destructuring can also be applied to objects and function parameters. We can see that the technique is extremely useful.

§

Why is it useful?

Destructuring patterns in general, is an extremely useful feature; many languages implement variations of this technique:

- **Scala** has the pattern matching construct, where we can even deconstruct at the type level.
- **Python** also supports deconstruction through its unpacking syntax, which allows us to assign values from a sequence (*such as a list or tuple*) to multiple variables simultaneously
- **Ruby** has support for deconstruction assignment, commonly known as "*destructuring*" or "*parallel assignment*". It allows us to assign values from arrays or hashes to multiple variables.
- Rust, similar to Scala, provides a pattern matching deconstruction feature, which allows us to extract and bind values from data structures, such as tuples or structs, to variables.

Fair, deconstructing is popular, but why is it so valuable? Well, simply put, it allows us to extract nested elements from sometimes complex structures, without indexing, and using considerably less code than if we were to index values in a loop, or even worst, one by one.

One of the main use cases for deconstruction, specially in JavaScript, is when we're working with APIs, but of course, that's not the only one.

1. Working with APIs

When working with external APIs, we have no guarantee that the structures returned will be simple, easy to understand, and well documented. Deconstruction can facilitate interaction with APIs that return complex data structures by easily extracting and assigning the relevant values, while keeping our code clean and readable.

2. Simplified code

When using deconstruction, we're effectively removing the intermediaries; we remove the use of any temporary variable. With deconstruction, we can directly assign values to variables, making the code more concise and easier to understand.

Additionally, it's extremely flexible; if we're not comfortable performing the full deconstruction in one line of code, we can separate our transformations in multiple lines, by going one level at a time, similar to what we would do in a `for` loop. We are in control of the depth of deconstruction, and can even stay at top levels if we don't need the full granularity.

3. Power

There are languages that have more powerful features when talking about deconstruction constructs. For example, Scala, a strongly-typed language, lets us deconstruct at type level, opening the possibilities for performing even the most granular operations, while maintaining our code safe.

Even though JavaScript is a dynamically-typed language, and there's no need to think of use cases regarding types, it still provides very powerful use cases, for example, when we're trying to unpack complex structures containing multiple nested objects. Just imagine if we were to use a `for` loop to achieve this.

§

General syntax

The destructuring syntax is analogous to the one we use to define the structures in the first place. For example, an array destructuring will involve square brackets `[]`, while an object destructuring will involve curly brackets `{}`. Let us look at the three in more detail.

1. Arrays

We can destructure an array by using the following syntax:

Code

```
const myArr = [1, 2, 3]
const [val1, val2, val3] = myArr
console.log(val1, val2, val3);
```

Output

```
1 2 3
```

We can also destructure nested arrays using a similar approach:

Code

```
const myNestedArr = [[1, 2], [3, 4], [5, 6], [7, 8]]
const [[val1,], [val2,], [val3,], [val4,]] = myNestedArr
console.log(val1, val2, val3, val4);
```

Output

```
1 3 5 7
```

In this example we introduced skipping elements, which we'll review shortly, but the idea is that we can go as deep as we'd like in terms of structure nesting, simply by mimicking the structure we're trying to deconstruct in the left side of the operation.

2. Objects

The mechanics are similar with objects. The main difference is that, with the latter, we're not working with an iterable object; an object does not store keys in an ordered structure, instead, we purposefully provide a unique name for each value.

Thus, we cannot skip elements or use other names to assign our variables. Instead, we must use the actual object key's names:

Code

```
const groundSupportPlane = {
    model: "A-10 Warthog",
    dimensions: { len: "53ft", wingspan: "57ft", height: "14ft" },
    power: { engines: "Twin General Electric TF34-GE-100 turbofans", thrust: "9,065 pounds (40.32 kilonewtons) of thrust per engine" },
    primaryWeapon: "GAU-8/A Avenger 30mm Gatling"
};

const { model, dimensions, power, primaryWeapon } = groundSupportPlane;
console.log(model, dimensions, power, primaryWeapon);
```

Output

```
A-10 Warthog { len: '53ft', wingspan: '57ft', height: '14ft' } {
  engines: 'Twin General Electric TF34-GE-100 turbofans',
  thrust: '9,065 pounds (40.32 kilonewtons) of thrust per engine'
} GAU-8/A Avenger 30mm Gatling
```

Once we have our variables assigned, we can rename them in the same expression. However, we'll leave this for later.

3. Function parameters

Another extremely useful application consists of designing objects containing function parameters, and then deconstructing them inside the actual function. This is called as parameter packing/unpacking, and can be of use whenever we have functions with a considerable amount of arguments.

We can declare a simple function, and pass its parameters in an object by using the following syntax:

Code

```
function getPlaneSpecs(args) {
    const { model, dimensions, power, primaryWeapon } = args
    console.log(`The ${model} is a ground support plane built by the US military.`);
    console.log(`Its dimensions are: ${dimensions.len} by ${dimensions.wingspan} by
${dimensions.height}.`);
    console.log(`The plane is equipped with a ${primaryWeapon} cannon, and is powered by two
${power.engines}.`);
}

// Declaring the function parameters object
const groundSupportPlane = {
    model: "A-10 Warthog",
    dimensions: { len: "53ft", wingspan: "57ft", height: "14ft" },
    power: { engines: "Twin General Electric TF34-GE-100 turbofans", thrust: "9,065 pounds (40.32
kilonewtons) of thrust per engine" },
    primaryWeapon: "GAU-8/A Avenger 30mm Gatling"
};

getPlaneSpecs(groundSupportPlane)
```

Output

```
The A-10 Warthog is a ground support plane built by the US military.
Its dimensions are: 53ft by 57ft by 14ft.
The plane is equipped with a GAU-8/A Avenger 30mm Gatling cannon, and is powered by two Twin
General Electric TF34-GE-100 turbofans.
```

As we can see, we can leave out parameters that are not important to us, and not even extract them from the original object.

§

Main features

Destructuring can be mainly applied to three different structures:

- Arrays
- Objects
- Function parameters

1. Skipping elements

Additionally, we can also skip elements, depending if we're not interested on returning a given element. For example, following the `myObj` implementation we reviewed earlier, we can extract only the first and third interest by using the following syntax:

Code

```
const [Int1, , Int3] = myObj.interests
console.log(Int1, Int3);
```

Output

```
Reading Sleeping
```

If we look closely, we skipped the middle element by simply not defining any variable in its place.

2. Using starter values

There are times when we don't know if an index position will be there (*in the structure we're working on*). In those cases, it's useful to have some kind of fallback value; otherwise, JavaScript will return `undefined`, which may lead to unexpected behavior and hard-to-debug errors, if we don't handle these occurrences carefully.

Destructuring lets us set predefined values for a given variable. For example, we might not know that the array inside the object we're trying to access contains only 3 values:

Code

```
const [val1 = "", val2 = "", val3 = "", val4 = ""] = myObj.interests
console.log(val1, val2, val3, val4);
```

Output

```
Reading Writing Sleeping
```

Otherwise, we would get the following:

Code

```
const [val1, val2, val3, val4] = myObj.interests
console.log(val1, val2, val3, val4);
```

Output

```
Reading Writing Sleeping undefined
```

This also applies for objects and function parameters:

Code

```
// Default values for objects
const { fName = "", lName = "", interests = "", favFood = "" } = myObj
console.log(fName, lName, interests, favFood);
```

Output

```
Emma Miller [ 'Reading', 'Writing', 'Sleeping' ]
```

Code

```
// Default values for function parameters
function myFun(params) {
    const { Country = "", City = "", Coords = "", Temp = "" } = params
    console.log(`Country : ${Country} \nCity: ${City} \nCoordinates: ${Coords} \nTemperature:
${Temp}`);
}

const funParams = {
    Country: "Austria",
    City: "Vienna",
    Coords: [48.2082, 16.3738]
}

myFun(funParams)
```

Output

```
Country : Austria
City: Vienna
Coordinates: 48.2082,16.3738
Temperature:
```

3. Renaming variables

We mentioned that, when destructuring objects, we needed to use the exact same variable names we declared when creating our object. For example, the destructuring pattern below would not work:

Code

```
const myObj = {
    val1: "JavaScript",
    val2: "is",
    val3: "weird"
}

const { word1, word2, word3 } = myObj
console.log(word1, word2, word3);
```

Output

```
undefined undefined undefined
```

The only way to do this, would be by referencing the keys by their actual name:

Code

```
const { val1, val2, val3 } = myObj
console.log(val1, val2, val3);
```

Output

```
JavaScript is weird
```

However, there is one trick we can use in order to reassign variable names:

Code

```
const { val1: word1, val2: word2, val3: word3 } = myObj
console.log(word1, word2, word3);
```

Output

```
JavaScript is weird
```

Note that we still need to use the original variable names, but now, we simply reassign them to other variables, effectively changing the name in the same operation.

§

Some practical examples

1. Deeply nested objects

Let us imagine we just got a database entry containing the information of a potential client we would like to contact in order to offer some of our services. The problem is that we have limited resources and would like to only get the most relevant client fields.

The object has the following structure:

Code

```
// Define an object
const clientInfo = {
    fName: "John Doe",
    age: 30,
    contact: {
        email: "johndoe@example.com",
        phone: {
            countryCode: "+1",
            number: "555-1234"
        },
        address: {
            street: "123 Main Street",
            city: "New York",
            state: "NY",
            zipCode: "10001"
        }
    },
    preferences: {
        language: "English",
        theme: "Dark",
        notifications: {
            email: true,
            sms: false
        }
    }
};
```

We can extract only the interesting fields, by extracting with the appropriate depth, and only selecting the required values:

Code

```
// Deconstruct the object by extracting the most relevant information
const { fName, age, contact: { email: address }, contact: { phone: { number }, address: { state
}, address: { city } }, preferences: { language }, preferences: { notifications: { email:
emailContactPermission } } } = clientInfo

// Log the values
console.log(fName, age, address, number, state, city, language, emailContactPermission);
```

We can see that we had to access deeply nested objects by using the following strategy:

```
level_1 : { level_2: { level_3: level_3_renamed } }
```

We also had to rename one variable, since we had repeated nested elements (`email`). However, the effort is worth it since we now have a nice set of our client's most relevant fields:

Output

```
John Doe 30 johndoe@example.com 555-1234 NY New York English true
```

2. Extracting intermittently from an unknown array

Let us imagine a scenario where we have an array with two levels of depth, and would like to extract the first, third and fifth elements from the first level, and all the elements except the first one, from the second level. However, the number of elements in the second level are unknown to us (*we only know that there's at least two elements*).

The structure is something as such:

Code

```
const myArr = [[1, 2, 3], [4, 5, 6], [7, 8, 9], ['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h',
'i']]
```

So the strategy would consist of the following:

- Get the required first-level elements using skipping.
- Get the required second-level elements using skipping, followed by a spread operator to include the rest of the elements.

We have not yet reviewed the rest operator. However, it's fairly simple; it includes all the elements of a given array, and is denoted by three dots `...`.

So, now that we have our strategy, we can define our destructuring pattern:

Code

```
const [[,  ...aa], , [,  ...bb], , [,  ...cc]] = myArr
console.log(aa, bb, cc);
```

Output

```
[ 2, 3 ] [ 8, 9 ] [ 'e', 'f' ]
```

3. An object containing multiple argument groups

Let us think of an example where we have a function that calculates a person's age by using their birth date, and then prints out a set of useful values about the person in question.

We can first abstract a person by using an object:

Code

```
// Define an object
const person = {
    fName: "Emma",
    lName: "Lovelace",
    birthDate: 1992,
    address: {
        street: "36, Obere Amtshausgasse",
        suburb: "Reinprechtsdorf",
        city: "Vienna",
        postCode: 1050, country: "Austria"
    },
    occupation: "Artist",
    family: {
        husband: "Peter",
        daughter: "Emma",
        son: "Christoph"
    }
}
```

We can then create a simple function for displaying basic data about this person, and returning a new object with only the fields we would like to preserve:

Code

```
// Defining a function
function introducePerson(args) {
    const currYear = new Date().getFullYear();

    // Since we do not want the person's address, we skip it
    const { fName, lName, birthDate, occupation, family: { husband } } = args;

    // Assign a new key-value pair including the person's calculated age
    const age = currYear - birthDate;
    person.age = age

    // Build a template literal
    const introduction = `Name: ${fName}\nLast Name: ${lName}\nAge: ${age}\nOccupation:
${occupation}\nHusband: ${husband}`
    console.log(introduction);

    // Build a new object with interesting fields
    const cleanPerson = {
        fName: fName,
        lName: lName,
        birthDate: birthDate,
        occupation: occupation,
        husband: husband
    }

    return cleanPerson
}

cleanPerson = introducePerson(person)
console.log(cleanPerson);
```

Output

```
Name: Emma
Last Name: Lovelace
Age: 31
Occupation: Artist
Husband: Peter

{
  fName: 'Emma',
  lName: 'Lovelace',
  birthDate: 1992,
  occupation: 'Artist',
  husband: 'Peter'
}
```

§

Conclusions

In this segment, we discussed what deconstruction is in the context of JavaScript, why is it useful, how can it be applied, to which structures can we apply it, its main syntax depending on the structure we're trying to deconstruct, some of the main features of this construct, and its main advantages. We also went over some simple practical examples in order to illustrate why this technique is so useful for some scenarios.

Deconstruction is a powerful construct

§

References

- Destructuring assignment, MDN Web Docs
- Destructuring Function Parameters in JavaScript, Medium
- A Brief Introduction to Array Destructuring in ES6, FreeCodeCamp

§

Copyright