Agenda: Destructuring and Spreads

- Introduction
- Array Destructuring
- Object Destructuring
- Mixed Destructuring
- Property renaming
- Spreads

Introduction

Destructuring is the convenient way of **extracting data** stored in objects and arrays. It can be used in the places where data is received in **LHS** of the statement.

It’s an ECMAScript 2015 feature.

Spread operator is the opposite of destructuring. It will break the array into individual components. It allows you to spread an array into another array, or an object into another object.

Let’s see how to extract values using various patterns.

Array Destructuring

The simplest form of destructuring is array destructuring assignment:

```
let input = [1, 2];
let[first, second] = input;
console.log(first); // outputs 1
console.log(second); // outputs 2
```

You can create a variable for the remaining items in a list using the syntax:

```
let [first, ...rest] = [1, 2, 3, 4];
console.log(first); // outputs 1
console.log(rest); // outputs Array with 3 elements: [ 2, 3, 4 ]
let [first] = [1, 2, 3, 4]; // You can just ignore trailing elements you don’t care about
console.log(first); // outputs 1
```

Elision lets you use the syntax of Array “holes” to skip elements during destructuring:

```
const [, , third, fourth] = ['a', 'b', 'c', 'd']; // third = 'c'; fourth= 'd'
```

exec - with regular expressions match:

```
const [totalDate, year, month, day] = /^(\d\d\d\d)-(\d\d)-(\d\d)$/.exec('2017-09-13');
console.log("totalDate - " + totalDate + "\n Individual Date -" + year + "-" + month + "-" + day);
```

exec() returns null if the regular expression doesn't match.

Unfortunately, you can't handle null via default values, which is why you must use the Or operator (||) as in example below:

```
const [totalDate, year, month, day] =    /^(\d\d\d\d)-(\d[
```

Object Destructuring

Destructuring object into variables

```
let o = {
    a: "foo",
    b: 12,
    c: "bar"
}
// Following creates new variables a and b from o.a and o.b. Notice that you can skip c if you don't need it.
let { a, b } = o; //value of o.a is assigned to a and o.b is assigned to b
let { b, a } = o; //Same as above, value of o.a is assigned to a and o.b is assigned to b
alert(a + " " + b)

Properties Renaming
let { a:X, b:Y } = o
alert(X + " " + Y)
```

You can create a variable for the remaining items in an object using the syntax:

```
let { a, ...passthrough } = o;
let total = a.length + passthrough.b + passthrough.c.length;
```

Nested Object Destructuring

This syntax is similar to the object literals, here we can navigate into nested objects and fetch or retrieve the data we want easily.

```
function nestedObjectDesctructuring() {
    let user = {
        department: "DP1",
        name: "SandeepSoni",
        favouriteCricketer: {
            first: {
                name: "Sachin"
            },
            second: {
```

```
            name: "Dhoni"
         }
      }
   };

   let { favouriteCricketer: { first, second }} = user;
   console.log(first.name);      // Sachin
   console.log(second.name);     // Dhoni
}
```

Here the destructuring pattern indicates to descend into the property **favouriteCricketer** on **user** and look for the properties **first** and **second**.

We can even declare a different variable for the local variables as below, just change the few lines from the above example

```
let { favouriteCricketer: { first:fav1, second:fav2 }} = user;
   console.log(fav1.name);      // sachin
   console.log(fav2.name);     // dhoni
```

Mixed Destructuring

Mixed Destructuring: In this pattern object and array destructuring can be used together to create complex expressions.

```
function nestedObjectDesctructuring() {
   let user = {
         department: "DP1",
         name: "SandeepSoni",
         favouriteCricketer: {
               first: {
                     name: "Sachin"
               },
               second: {
                     name: "Dhoni"
               }
         },
         hobbies: ["Playing Cricket", "Playing Chess"]
   };

   let { favouriteCricketer: { first: fav1, second: fav2 }, hobbies: [hob1, hob2]} = user;
   console.log(fav1.name);            // Sachin
   console.log(fav2.name);            // Dhoni
```

```
    console.log(hob1);                    // Playing Cricket
    console.log(hob2);                    // Playing Chess
}
nestedObjectDesctructuring();
```

You need to keep in mind that **favouriteCricketer:** and **hobbies:** in the destructured pattern are just locations that correspond to properties in the user object.

for..of iteration and destructing:

```
function forOFIteration_Destructing() {
    let users = [
        {
            department: "DP1",
            name: "Ramesh",
        },
        {
            department: "DP2",
            name: "suresh",
        },
        {
            department: "DP3",
            name: "somesh",
        }];

    for (var {name: n, department: dept} of users) {
        console.log('Name: ' + n + ', Department: ' + dept);
    }
}
forOFIteration_Destructing();
```

Optional and Default Values for Parameters

Default Values: It let you specify a default value in case a property is undefined:

```
function keepWholeObject(wholeObject: { a: string, b?: number }) {
    let { a, b = 1001 } = wholeObject;  //If b is undefined, set b = 1001 else b = wholeObject.b
}
```

keepWholeObject now has a variable for wholeObject as well as the properties a and b, even if b is undefined.

Functions with optional / default values for parameters

```
type C = { a: string, b?: number } //a is mandatory and b is optional
function f({ a, b }: C): void {
   // ...
}
```

But specifying defaults is more common for parameters, and getting defaults right with destructuring can be tricky. First of all, you need to remember to put the type before the default value.

```
function f({ a, b } = { a: "", b: 0 }): void {
   // ...
}
f(); // ok, default to { a: "", b: 0 }
```

Then, you need to remember to give a default for optional properties on the destructured property instead of the main initializer. Remember that C was defined with b optional:

```
function f({ a, b = 0 } = { a: "" }): void {
   // ...
}
f({ a: "yes" }) // ok, default b = 0
f() // ok, default to { a: "" }, which then defaults b = 0
f({}) // error, 'a' is required if you supply an argument
```

Use destructuring with care. As the previous example demonstrates, anything but the simplest destructuring expression is confusing. This is especially true with deeply nested destructuring, which gets *really* hard to understand even without piling on renaming, default values, and type annotations. Try to keep destructuring expressions small and simple. You can always write the assignments that destructuring would generate yourself.

Spread

The spread operator is the opposite of destructuring. It will break the array into individual components using (**Three-dot ellipsis (…) as a prefix to the array**).

It allows you to spread an array into another array, or an object into another object.

Array spreading

```
let first = [1, 2];
let second = [3, 4];
let combined = [0, ...first, ...second, 5];
```

This gives ***combined*** the value **[0, 1, 2, 3, 4, 5].** Spreading creates a shallow copy of first and second. They are not changed by the spread.

Object spreading:

You can also spread objects:

```
let defaults = { food: "spicy", price: "$10", ambiance: "noisy" };
let search = { ...defaults, food: "rich" };
```

Now search is { food: "rich", price: "$10", ambiance: "noisy" }. Object spreading is more complex than array spreading. Like array spreading, it proceeds from left-to-right, but the result is still an object.

This means that properties that **come later** in the spread object **overwrite** properties that come earlier.

So if we modify the previous example to spread at the end:

```
let defaults = { food: "spicy", price: "$10", ambiance: "noisy" };
let search = { food: "rich", ...defaults };
```

Then the food property in defaults overwrites food: "rich", which is not what we want in this case.

Object spread also has a limitation, you **lose methods** when you spread instances of an object:

```
class C {
  p = 12;
  m() {
  }
}
let c = new C();
let clone = { ...c };
clone.p; // ok
clone.m(); // error!
```