Lesson 5.1 - Destructuring Assignment

ES2015 comes with a brand new way to assign variables when dealing with arrays and objects. This new feature is called `destructuring assignment` and it allows you to easily assign variables from array elements or key/value pairs for an object. Let’s dive right in shall we?

Array Matching

The first type of destructuring we’re going to talk about is array matching. Put simply, this feature allows you to assign a list of variables from matching indices in arrays. Let’s take a look at this code example so that we can better see how this works.

```
> const list = [1, 2, 3]
> const [a, b, c] = list
> console.log(a, b, c)
-> 1 2 3
```

As you can see there, we were able to easily assign to three variables, a, b, and c, the values that matched on the indices of the array.

If we want to only match on some of our indices, we can use the spread operator like this:

```
> const list = [1, 2, 3, 4, 5]
> const [a, b, ...c] = list
> console.log(a, b, c)
-> 1 2 [3, 4, 5]
```

It is also possible to skip an index when doing this type of assignment.

```
> const list = [1, 2, 3]
> const [a, ,c] = list
> console.log(a, c)
-> 1 3
```

It’s as easy as just not passing in a new variable name at the index that you would like to skip, and the variable won’t be assigned.

Object Matching

Another great destructuring assignment that is now available to us is object matching. With object matching we can assign variables in much the same way that we did using array matching, but in this case we will be matching our variable names to the object keys, instead of array indices.

```
> const obj = { a: 1, b: 2, c: 3 }
> const { a, b, c } = obj
> console.log(a, b, c)
-> 1 2 3
```

As you can see, the syntax differs slightly here in that we're using curly braces for the destructuring assignment, to match the fact that we are destructuring on an object. This works equally as well for deeply nested objects.

```
> const obj = { a: { b: 1, c: { d: 2 } } }
> const { a: { b, c: { d } } } = obj
> console.log(b, d)
-> 1, 2
```

Now you can very easily extract a value from a deeply nested object using this new destructuring assignment! This is a great new feature.

We can use these same types of assignments in functions.

Parameter Context Matching

Using parameter context matching we can use this same type of destructuring assignment syntax in the definition of function parameters. Let's take a look at the following example.

```
function print({ value }) {
  console.log(value)
}
```

In this example, the `print` function is expecting to receive as an argument, an object with a key of `value`.

```
> print({value: "Hello World!"})
-> Hello World!
```

You can also extend this to work in various ways:

```
> const obj = { value: 'Hello' }
> print(obj)
-> Hello
```

Here we just passed in a predefined object that fits what your function is looking for. But what if we had a variable named `value`, that already contains the value that we want to print? ES2015 has something for that too!

```
> const value = 'Hello World!'
> print({ value })
-> Hello World!
```

That's really powerful and flexible! This allows you to write a lot less code, and to easily refactor existing code.

You might think that this has all been really amazing, but we're not done yet! There's one more great feature, and it really puts the cherry on top of everything we have covered so far.

Fail Soft Destructuring

When attempting to assign a variable via destructuring, if the indice or object key does not exist, that variable will simply be `undefined`, instead of throwing an error. That's not all though, we can also assign a default value if we want! Let's see this in action.

```
> const list = [1, 2]
> const [a = 10, b = 6, c = 40, d] = list
> console.log(a, b, c, d)
-> 1 2 40 undefined
```

This also works for objects:

```
> const obj = { a: 1, b: 2 }
> const { a = 10, b = 5, c = 8, d }
> console.log(a, b, c, d)
-> 1 2 8 undefined
```

Awesome! This is going to make variable assignment a breeze!