Object and Array Destructuring

Let’s check out the ES2015 feature called destructuring. To better understand it, let's take a look at some of the basics of Javascript objects. To add a single property to an object, you use dot notation. With dot notation, you can only add properties to an object one at a time. The same syntax can be used to extract data, again, one property at a time.

```
const user = {};
user.name = "Tyler McGinnis";
user.handle = "@tylermcginnis";
user.location = "Eden, Utah";

const name = user.name;
const handle = user.handle;
```

If you wanted to add multiple properties to an object at the same time, you would need to use JavaScript's "object literal notation" when you initialize the object.

```
const user = {
  name: 'Tyler McGinnis',
  handle: '@tylermcginnis',
  location: 'Eden, Utah',
};

const name = user.name;
const handle = user.handle;
```

There's a way to add properties one at a time, extract properties one at a time, add multiple properties at the same time, but unfortunately, there's no comparable way to extract multiple properties from an object at the same time. That is, until "destructuring" was introduced in ES2015. Destructuring allows us to extract multiple properties from an object. This can drastically decrease the amount of code we need to write when we want to extract data from an object, because what used to look like this,

```
const name = user.name;
const handle = user.handle;
const location = user.location;

can now look like this,

const { name, handle, location } = user;
```

The syntax can be a little bit weird but know that these two blocks of code are identical in that they both create and initialize three new variables. You can think of it like this, if you want to add properties to an object, do it as you are used to, on the right-hand side of the equal sign. If you want to extract properties from an object, do it on the left-hand side of the equal sign.

Destructuring also allows you to destructure the results of function invocations. For example, below we have a function called getUser() which returns the user object. Rather than invoking getUser() and grabbing all of the properties off of it one by one, we could get the same result by destructuring the result of that invocation.

```
function getUser() {
  return {
    name: "Tyler McGinnis",
    handle: "@tylermcginnis",
    location: "Eden, Utah",
  };
}

const { name, handle, location } = getUser();
```

Up until this point we've talked about how destructuring helps us extract data from objects, but what about arrays? Though not as common as object destructuring, array destructuring is a thing and it is still pretty useful in certain circumstances, specifically when the location of an item in the array is the main differentiator for that item. So here we have a user array with each item being a unique piece of information about the user,

```
const user = ["Tyler McGinnis", "@tylermcginnis", "Eden, Utah"];
```

You'll notice that this array probably should just be an object. But sometimes you have to take what you can get from weird external API's. Typically if we want to better identify each item in the array we need to create a variable for each item.

```
const name = user[0];
const handle = user[1];
const location = user[2];
```

However just like with objects, array destructuring allows us to more effectively extract items from an array so the above code, can now look like the code below.

```
const [name, handle, location] = user;
```

Just as we saw from objects you can use array destructuring with function invocations. For example, below "split" is going to return an array with each item in the array being a specific property of the car.

```
const cvs = "1997,Ford,F350,MustSell!";
const [year, make, model, description] = csv.split(",");
```

By using array destructuring, we are able to extract each property into their own, user readable variable. So that's it in regards to the basics of destructuring, again destructuring allows us to easily extract data from an object or an array. There are, however, what I'd consider to be more advanced features of destructuring that are worth taking a look at.

For example, what if when we do destructure an object, we wanted the variable name to be different than the property name on that object. So say we had an object that looked like this,

```
const user = {
  n: "Tyler McGinnis",
  h: "@tylermcginnis",
  l: "Eden, Utah",
};
```

Since we are not masochists and we actually like the other developers on our team, we don't want to make three one letter variable names. Instead, we can have the property names on the left of the colon and the new variable names on the right. Now, we are not only destructuring the user object, but we are also renaming the poorly named properties into more easily understood variable names.

```
const { n: name, h: handle, l: location } = user;
console.log(name); // Tyler McGinnis
console.log(handle); // @tylermcginnis
console.log(location); // Eden, Utah
```

This may seem like a rarely used feature, but it is actually pretty common. To find a real world example we don't have to look very far. This is the implementation of the render method in React Router Native's Link component. Note how we're renaming component with a lowercase "c" to Component with a capitalized "c".

```
render () {
  const { component: Component, to , replace, ...rest } = this.props
  return <Component {...rest} onPress={this.handlePress}/>
}
```

Next, let's talk about function arguments and parameters. Below we have a fetchRepos() function which is going to be in charge of fetching a group of repositories from the Github API.

```
function fetchRepos(
  language,
  minStars,
  maxStars,
```

```
  createdBefore,
  createAfter,
) {}
```

The first thing you'll notice is that we have a lot of control over the type of repositories that we will be fetching. Fortunately, this leads to a stupid amount of arguments that can be passed into the function. Currently when we invoke our fetchRepos() function, we have two issues. First, we need to remember or look up which arguments go in which order. Second, we need to read and hope that the documentation has instructions for what to do with our arguments that we do not care about. In this case, we will just use null and hope for the best.

```
function fetchRepos(
  language,
  minStars,
  maxStars,
  createdBefore,
  createAfter,
) {}

fetchRepos(
  "JavaScript",
  100,
  null,
  new Date("01.01.2017").getTime(),
  null,
);
```

The good news is that destructuring helps us with both of these problems. First, let's solve the positional parameters problem. What if instead of passing in each argument one by one, we pass in an object instead? Now, before we ever need to look at the function definition of fetchRepos, we know exactly what information it needs. Even more important, order no longer matters.

```
function fetchRepos(
  language,
  minStars,
  maxStars,
  createdBefore,
  createAfter,
) {}

fetchRepos({
  language: "JavaScript",
  maxStars: null,
  createdAfter: null,
  createdBefore: new Date("01/01/2017").getTime(),
```

```
  minStars: 100,
});
```

Now we need to modify the fetchRepos function definition. This is where destructuring comes into play. Because we are receiving an object as the argument to the function, we can destructure it. So now the code above, can be changed to this.

```
function fetchRepos({
  language,
  minStars,
  maxStars,
  createdBefore,
  createAfter,
}) {}

fetchRepos({
  language: "JavaScript",
  maxStars: null,
  createdAfter: null,
  createdBefore: new Date("01/01/2017").getTime(),
  minStars: 100,
});
```

Again, the biggest benefit here is that we have removed the order out of the equation entirely, so that's one less thing we have to worry about. The second problem we had earlier with our code was that we needed to figure out what to do with the arguments we did not care about. Before we just passed in null, but now that we are passing in an object rather than arguments one by one, we can actually just remove the null values altogether and that will give us a function invocation that looks like this.

```
function fetchRepos({
  language,
  minStars,
  maxStars,
  createdBefore,
  createAfter,
}) {}

fetchRepos({
  language: "JavaScript",
  createdBefore: new Date("01/01/2017").getTime(),
  minStars: 100,
});
```

This now leads us back to our function definition of fetchRepos. We need a way to establish default values for any properties that aren't on the arguments object when the function is invoked. Typically that would look like this.

```
function fetchRepos({
  language,
  minStars,
  maxStars,
  createdBefore,
  createAfter,
}) {
  language = language || All;
  minStars = minStars || 0;
  maxStars = maxStars || "";
  createdBefore = createdBefore || "";
  createdAfter = createdAfter || "";
}
fetchRepos({
  language: "JavaScript",
  createdBefore: new Date("01/01/2017").getTime(),
  minStars: 100,
});
```

For each different possible property, we'd set the value of that property to itself or a default value if the original value was undefined. Luckily for us, another feature of destructuring is it allows you to set default values for any properties. If a partially destructured value is undefined, it will default to whatever you specify. What that means is that the ugly code above can be transformed into this,

```
function fetchRepos({
  language = "All",
  minStars = 0,
  maxStars = "",
  createdBefore = "",
  createdAfter = "",
}) {}
```

We set the default value of each property in the same place where we just destructured the parameters. Now that we've seen the power of using object destructuring to destructure an object's parameters, can the same thing be done with array destructuring? Turns out, it can.

My favorite example of this is with Promise.all. Below we have a getUserData function.

```
function getUserData(player) {
  return Promise.all([getProfile(player), getRepos(player)]).then(
    function (data) {
      const profile = data[0];
```

```
      const repos = data[1];

      return {
        profile: profile,
        repos: repos,
      };
    },
  );
}
```

Notice it's taking in a player and returning us the invocation of calling Promise.all. Both getProfile and getRepos return a promise. The whole point of this getUserData function is that it's going to take in a player and return an object with that player's profile as well as that player's repositories. If you're not familiar yet with the Promise.all API, what's going to happen here is getProfile and getRepos are both asynchronous functions. When those promises resolve (or when we get that information back from the Github API), the function that we passed to then is going to be invoked receiving an array (in this case we are calling it data). The first element in that array is going to be the user's profile and the second item in the array is going to be the user's repositories. You'll notice that order matters here. For example, if we were to pass another invocation to Promise.all, say getUsersFollowers, then the third item in our data array would be their followers.

The first update we can make to this code is we can destructure our data array. Now we still have our profile and repos variables, but instead of plucking out the items one by one, we destructure them.

```
function getUserData(player) {
  return Promise.all([getProfile(player), getRepos(player)]).then(
    function (data) {
      const [profile, repos] = data;
      return {
        profile: profile,
        repos: repos,
      };
    },
  );
}
```

Now just as we saw with objects, we can move that destructuring into the parameter itself.

```
function getUserData(player) {
  return Promise.all([getProfile(player), getRepos(player)]).then(
    ([profile, repos]) => {
      return {
        profile: profile,
```

```
        repos: repos,
      };
    },
  );
}
```

Now we still have profile and repos, but those are being created with array destructuring inside of the function's parameters.