ES2015 introduced a new feature called Destructuring.

***“Destructuring allows to extract data more easily from an object or an array.”***

**BASICS:**

In JavaScript we have ways to add properties to an object one at a time, extract properties one at a time and add multiple properties at the same time but unfortunately, there’s no comparable way to extract multiple properties from an object at same time. That is until De-structuring ES2015 introduced.

Add properties to an object one at a time, using dot ( . ) notation,

Example: 1

var user = {};

user.name = ‘Tyler’;

user.handle = ‘@tyler123’

Extract properties one at a time, using same dot ( . ) notation,

Example: 2

var name = user.name

var handle = user.handle

Add multiple properties at the same time, using JavaScripts object literal notation,

Example: 3

var user = {

name: ‘tyler’,

handle: ‘@tyler123’

}

Just like object literal notation allows us to ADD multiple properties to an object, Destructuring allows us to EXTRACT multiple properties from an object at the same time.

Example: 4

var user = {

name: ‘tyler’,

handle: ‘@tyler123’,

location: ‘Eden, Utah’

}

// var name = user.name;

// var handle = user.handle;

// var location = user.location;

var { name, handle, location } = user; // using destructuring

The last line in above code initializes three new variables.

**Tip:**

If you want to assign properties to an object we do it at right hand side of the equal sign.

If we want to **extract** properties from an object we do it on **left hand** side of the equal sign.

**Another feature -------------------------------------------------------------------------------------------------------------**

You can also destructure the results of function invocation, by destructuring the results of that function invocation,

Example: 5

function getUser() {

return {

name: ‘tyler’,

handle: ‘@tyler123’,

location: ‘eden, utah’

};

}

var { name, handle, location } = getUser();

**Another feature is Array Destructuring-------------------------------------------------------------------------------**

Array Destructuring allows us to more effectively extract the items from an array.

Example: 6

var user = [‘tyler’, ‘@tyler123’, ‘Eden, Utah’];

// var name = user[0];

// var handle = user[1];

// var location = user.[2];

var [ user, handle, location ] = user;

Example: 6

var carcsv = “1997, Ford, F350, Must Sell”;

var [ year, make, model, description ] = carcsv.split(‘,’);

ADVANCED FEATURES OF DESTRUCTURING

1. **Renaming**

When we do destructure an object we want new variable names to be different than property names on that object, i.e, new variable names =/= property names

var user = {

n: ‘tyler’,

h: ‘@tyler123’,

l: ‘Eden, Utah’

}

With property name on the left of the colon and new variable name on right of the object.

property\_name : new\_variable\_name

var { n:name, h: handle, l:location } = user;

*Here we renamed poorly named properties into more easily understood variable names.*

**Application:** Desctructuring this.props in ReactJS,

render () {

const { component : Component, to, replace, …rest } = this.props;

// ..do something

}

Here, we are renaming component with lower case ‘c’ to Component with upper case ‘C’ because in React, the component name needs to be capitalize.

1. **Destructuring function’s parameters**

**Note:**

**The PARAMETERS are used to *define* the method signature.**

**The ARGUMENTS are the actual values of a *particular call* to a function.**

var foo = function( a, b, c ) {}; // a, b, and c are the parameters

foo( 1, 2, 3 ); // 1, 2, and 3 are the arguments

function fetchRepos ( language, minStars, maxStars, createdBefore, createdAfter) {

}

In this function, you’ll notice that this leads to stupid amount of arguments that can be passes to this function.Now when we invoke fetchRepos function, we have two issues:

1. Order of the arguments passes needs to be taken care
2. What to do with the arguments we don’t care about

**INVOKING WITHOUT DESTRUCTURING:**

fetchRepo ( ‘JavaScript’, 100, null, new Date{‘01/01/2019’).getTime(), null )

**INVOKING WITH DESTRUCTURING:**

The order no longer matters ( + ) we know what information we need

fetchRepo ( {

language: ‘JavaScript’,

maxStars: null,

createdAfter: null,

createdBefore: new Date{‘01/01/2019’).getTime(),

minStars: 100

})

***ISSUE 1 ADDRESSED***

**DEFINITION WITH DESTRUCTURING:**

*Here we destructured the parameters which removed the order of the equation entirely*

function fetchRepos (**{** language, minStars, maxStars, createdBefore, createdAfter}) {

}

***ISSUE 2 ADDRESSED***

Establish default values for any property that isn’t in the argument:

**DEFINITION WITHOUT DESTRUCTURING:**

function fetchRepos ({ language, minStars, maxStars, createdBefore, createdAfter}) {

language: language || ‘All’;

minStars: minStars || 0;

maxStars: maxStars || ‘ ’;

createdBefore : createdBefore || ‘ ’;

createdAfter : createdAfter || ‘ ’

}

**DEFINITION WITH DESTRUCTURING:**

*Here we destructure the parameters ( + ) we set default values for each property*

function fetchRepos (**{ language= ‘All’ , minStars = 0, maxStars = ‘ ’, createdBefore = ‘ ’, createdAfter = ‘ ’})** {

}

Invoking:

fetchRepo ( {

language: ‘JavaScript’,

createdBefore: new Date{‘01/01/2019’).getTime(),

minStars: 100

})

1. **Array Destructuring**

getProfile() and getRepo() are two asynchronous functions, when these functions are resolved or when we get the information from github api, the function that we passes to .then is going to be invoked and is going to be passes an array, in this case we call it ‘data’. The first item in data array is going to be user’s profile and second item is going to be user’s repositories. Notice: Order matters here.

**a. NORMAL:**

function getUserData( player ){

return Promise.All ( [

getProfile(player);

getRepo(player);

]).then (function(data) {

**var profile = data[0];**

**var repo = data[2];**

return {

profile: profile;

repo: repo;

}})

**c. DESTRUCTURING INTO PARAMETERS:**

function getUserData( player ){

return Promise.All ( [

getProfile(player);

getRepo(player);

]).then (function**([ profile, repo])** {

return {

profile: **profile**;

repo: **repo**;

}

})

**b.DESTRUCTURING DATA ARRAY:**

function getUserData( player ){

return Promise.All ( [

getProfile(player);

getRepo(player);

]).then (function(data) {

***var [ profile, repo] = data*;**

return {

profile: profile;

repo: repo;

}})

**d. MORE CONDENSED:**

function getUserData( player ){

return Promise.All ( [

getProfile(player);

getRepo(player);

]).then ( **([ profile, repo])** =>

**[ profile, repo]**

})

**Shallow copy and Deep Copy**

First of all, what is a copy?

A copy just looks like the old thing, but isn’t. When you change the copy, you expect the original thing to stay the same, whereas the copy changes.

In programming, we store values in variables. Making a copy means that you initiate a new variable with the same value(s). However, there is a big potential pitfall to consider: **deep copying** vs. **shallow copying**.

A deep copy means that all of the values of the new variable are copied and **disconnected from the original**variable.

A shallow copy means that certain (sub-)values are **still connected** to the original variable.

To really understand copying, you have to get into how JavaScript stores values.

**Primitive data types**

Primitive data types include the following:

1. Number — e.g. 1
2. String — e.g. 'Hello'
3. Boolean — e.g. true
4. undefined
5. null

When you create these values, they are tightly coupled with the variable they are assigned to. They only exist once. When you make a copy, it will be a real copy. Let’s see an example:

const a = 5

let b = a // this is the copy

b = 6

console.log(b) // 6

console.log(a) // 5

By executing b = a , you make the copy. Now, when you reassign a new value to b, the value of b changes, but not of a.

**Composite data types — Objects and Arrays**

Technically, arrays are also objects, so they behave in the same way. I will go through both of them in detail later.

Here it gets more interesting. These values are actually stored just once when instantiated, and assigning a variable just creates **a pointer (reference) to that value**. That is why, if we make any change to new object or array the original array also gets same changes.

**Array:**

Now, if we make a copy b = a , and change some nested value in b, it actually changes a’s nested value as well, since a and b actually point to the same thing. Example:

**var** a = [ 'Punjab', 'Delhi', 'Mumbai', 'Bangalore' ];

var b = a;

b[0] = 'J&K';

console.log(b) // ["J&K", "Delhi", "Mumbai", "Bangalore"]

console.log(a) // ["J&K", "Delhi", "Mumbai", "Bangalore"]

*Using ‘const’ to declare original variable also doesn’t prevent its mutation.*

**const** d = [ 'Punjab', 'Delhi', 'Mumbai', 'Bangalore' ];

var b = d;

b[0] = 'J&K';

console.log(b) // ["J&K", "Delhi", "Mumbai", "Bangalore"]

console.log(d) // ["J&K", "Delhi", "Mumbai", "Bangalore"]

**Object:**

const car = {

color: 'Red',

model: 'F350',

year: '2017'

}

let newcar = car;

newcar.color = 'White';

console.log(car) // {color: "**White**", model: "F350", year: "2017"}

console.log(newcar) // {color: "**White**", model: "F350", year: "2017"}

**Shallow copying**

An object / Array is said to be shallow copied when the source top-level properties are copied without any reference, but the nested or sub properties are copied with reference.

**Shallow copy in Arrays**

1. Concate

var copyArray = origArray.concate( oldArray )

1. From

var copyArray = Array.from( origArray)

1. Slice

var copyArray = origArray.slice( 0)

1. Spread Operator

var copyArray = [ …origArray]

1. Array functions — map, filter, reduce

Example:

const originalLocation = ['Delhi','Mumbai', 'Kolkata','Chennai', {'state': 'Gujrat'}];

var copiedLocation = Array.from(originalLocation);

copiedLocation[0] = 'Ghaziabad'; // changing top level element

copiedLocation[4]['state'] = 'Punjab'; // changing Nested element

console.log(originalLocation)

// ["Delhi", "Mumbai", "Kolkata", "Chennai", {state: "Punjab"}]

console.log(copiedLocation)

// ["Ghaziabad", "Mumbai", "Kolkata", "Chennai", {state: "Punjab"}]

Here, we can see that top-level elements in original array is not impacted.

However, the nested element’s value is changed by changing copied array.

**Shallow copy in Objects**

1. Assign

Var newObject = Object.assign( {}, sourceObject )

1. Spread Operator

Var newObject = { … sourceObject }

**Deep Copying**

A deep copy will duplicate every object it encounters ( both top-level and nested ). The copy and the original object will not share anything.

### Using JSON.parse(JSON.stringify(object));

**Deep copy in Objects**

let origObject = {

a: 1,

b: {

c: 2,

},

}

let newObject = JSON.parse(JSON.stringify(origObject));

newObject.a = 5;

newObject.b.c = 20;

console.log(origObject); // {a: 1, b: {c: 2}} (Original Object Intact!)

console.log(newObject); // {a: 5, b: {c: 20}}

**Deep copy in Arrays**

let originalArr = ['Delhi', 'Mumbai', {'state': 'Punjab'}];

let newArr = JSON.parse(JSON.stringify(originalArr));

newArr[0] = 'Chandigarh';

newArr[2]['state'] = 'Rajasthan';

console.log(originalArr);

// ["Delhi", "Mumbai", {state: "Punjab"}] (Original Array Intact!)

console.log(newArr); // ["Chandigarh", "Mumbai", {state: "Rajasthan"}]