Chapter 7: Next Generation JavaScript: Intro to ES6 / ES2015:

3. What's new in ES6 / ES2015:

JavaScript Versions:

ES6 / ES2015	→ Supported in all modern browsers
ES7 / ES2016	→ No support in older browsers
ES8 / ES2017	→ Can use most features in production with transpiling and polyfilling (converting to ES5).

Now ES6 Features We'll Cover in this Section:

→ variable declarations with let and const.

→ Blocks and IIFEs

→ Strings

→ Arrow Functions

→ Destructuring

→ Arrays

→ The Spread Operator

→ Rest and Default Parameters

→ Maps

→ Classes and Subclasses.

Later....

→ Promises

→ Native modules.

4. Variable Declarations with let and const:

Use the starter project: 7-EcmaScript2015

script.js:

```
// let and const.

// ES5
var name5 = 'Jane Smith';
var age5 = 23
name5 = 'Jane Miller';
console.log(name5);

//ES6
Const name6 = 'Jane Smith';
let age6 = 23;
name6 = 'Jane Miller';
console.log(name6);
```

Output:

Jane Miller

▶ Uncaught TypeError: .....

The error is because the const keyword is used to declare constants, variables that are immutable, which we cannot change and if we attempt to change the value of a constant, it gives an error.

In ES6, we no longer use var, but instead we use const if we have a variable that's not gonna change its value over time and we use let if we want to change the value of the variable.

That's not the only thing that's new with let and const. That's because variables declared with var in ES5 are function-scoped but variables declared with let and

const are block-scoped. We'll see what that means with this example:

script.js:

```
//ES5
function driversLicense5(passedTest) {
    if (passedTest) {
        var firstName = 'John';
        var yearOfBirth = 1990;
        console.log(firstName + ', born in '
            + yearOfBirth + ', is now officially
            allowed to drive a car.');
    }
}
driversLicence5(true);

// ES6
function driversLicence6(passedTest) {
    if (passedTest) {
        let firstName = 'John';
        const yearOfBirth = 1990;
```

```
        console.log (firstName + ....);
    }
}
driversLicence6 (true);
```

Output:

John, born in 1990, is now officially allowed to drive a car.

John, born in " " " ".

If we now move the console.log outside of the if block:

script.js:

```
//ES5
function driversLicence5(passedTest) {
    if (passedTest) {
        var firstName = 'John';
        var yearOfBirth = 1990;
    }
```

```
    console.log(firstName + ....);
}

driversLicence5(true);

//ES6
function driversLicence6(passedTest) {
    if (passedTest) {
        let firstName = 'John';
        const yearOfBirth = 1990;
    }
    console.log(firstName + ....);
}
driversLicence6(true);
```

Output:

```
John, born in 1990, ....
▶ Uncaught ReferenceError: firstName not defined....
```

· We got this error because with let and const, the variables are not function-scoped but block-scoped. A block is all the code that is wrapped between the curly braces of if condition `if { .... block... }`. So each time we have an if statement or a for block of a while block, we're actually creating a new block and the variables declared with let and const are only accessible by the code that are inside of the same block. With var, we have access if it's in the same function.

To make this work, in ES6, we have to declare the variables outside of the block:

script.js:

```
//ES6
function driversLicence6(passedTest) {
```

```
let firstName;
const yearOfBirth = 1990;
if (passedTest) {
    firstName = 'John';
}
console.log (firstName + ....);
}
driversLicence6(true);
```

for const, we can't declare the variable here and assign a value afterwards.

If we do this:

For variables:

```
console.log (firstName);
var firstName = 'John';
```

Output: undefined.

For let and const:

```
Console.log (firstName);
let firstName;
firstName = 'John';
```

Output: ▸ Uncaught Ref.Error: firstName not defined.

So, we can't use a variable before it is declared if we use let and const. This is a good thing bcoz it prevents errors.

This happens because of something called Temporal - dead Zone: This means that the variables are hoisted but we still cannot access them before they are declared

Lets look at another example:

script.js:

```
let i = 23;

for (let i=0; i<5; i++){
    console.log (i);
}
console.log(i);
```

Output:

```
0
1
2
3
4
23
```

This means assigning other values to i in for-loop doesn't change the i variable that we defined in the beginning. with a let. That's because these variables are block-scoped. So they are two completely different variables.

If we use var instead of let, we'll get this output:

0
1
2
3
4
5

So, if you want to start using ES6, then the best practice is to use let for variables that will change the value over time and const for variables that cannot be reassigned.

5. Blocks and IIFEs:

Up until this point, we used IIFEs for data privacy. But ES6 provides a much simpler way of achieving data privacy. All we have to do is to use a block.

script.js:

```
// Blocks and IIFEs.
{
    const a = 1;
    let b = 2;
}
console.log ( a + b);
```

Output: ▶ Uncaught ReferenceError : a is not defined...

In ES5, we did it this way!

script.js:

```
(function () {
    var c = 3;
})();
console.log(c);
```

6. Strings in ES6 / ES2015:

script.js:

```
// Strings: Template Literals.

let firstName = 'John';

let lastName = 'Smith';

const yearOfBirth = 1990;

function calcAge(year) {

    return 2016 - year;

}

//ES5

console.log('This is ' + firstName + ' ' +

lastName + '. He was born in ' +

yearOfBirth + '. Today, he is ' +

calcAge(yearOfBirth) + ' years old.');

//ES6                    This is a backtick

console.log(`This is ${firstName} ${lastName}.

He was born in ${yearOfBirth}. Today,

he is ${calcAge(yearOfBirth)} years old.`);
```

```
Const n = `${firstName} ${lastName}`;
console.log(n.startsWith('J'));
console.log(n.endsWith('Sm'));
console.log(n.includes('oh'));
Console.log(`${firstName} `.repeat(5));
```

Output:

This is John Smith. He was born in 1990.
Today, he is 26 years old.

This is " " " " "
" " "

true
false
true
John John John John John.

7. Arrow Functions: Basics:

script.js:

```
//Arrow functions.

const years = [1990, 1965, 1982, 1937];

//ES5

var ages5 = years.map(function(el) {

    return 2016-el;

});
console.log(ages5);

//ES6: One line & one argument:

let ages6 = years.map(el => 2016 - el);
console.log(ages6);
// Two arguments, we have to use parentheses.
ages6 = years.map((el, index) => `Age

    element ${index+1}: {2016-el}.`);
Console.log(ages6);
// More lines of code, use curly braces & return keyword
ages6 = years.map((el, index) => {

    const now = new Date().getFullYear();
```

In map method, we have access to current element, current index and entire years array.

at the end.

```
    const age = now - el;
    return `Age element ${index + 1}:
                        ${age}.`
});
console.log(ages 6);
```

Output:

- [26, 51, 34, 79]
- [26, 51, 34, 79]
- ["Age element 1: 26.", "Aget element 2: 51.", "Age element 3: 34.", "Age element 4: 79."]
-

8. Arrow Functions: Lexical 'this' Keyword:

script.js:

```
// Arrow functions 2:

// ES5

var box5 = {
    color: 'green',
    position: 1,
```

```
clickMe: function() {
    document.querySelector('.green').addEventListener('click', function() {
        var str = 'This is box number ' +
            this.position + ' and it is ' +
            this.color;
        alert(str);
    });
}
}
box5.clickMe();
```

If we run this and click on the green box, we'll get this in alert:

This is box number undefined and it is undefined.

The reason for this is that only in a method call, the 'this' keyword points to that object. But in a regular function call, the 'this' keyword will always point to the global object, which in the case of the

browser, is the window object.

To avoid this, we have to create a new variable in the method and store the 'this' variable in it. It looks like this.

script.js:

```
//ES5
var box5 = {
    :

    clickMe: function (){

        var self = this;

        document..... ('click', function(){
            &
            var str = 'This is box number '
            + self.position + ' and it is '
            + self.color;

        alert(str);

        });

    }

}

box5.clickMe();
```

And now, it will work. This is a simple hack

The arrow functions share the surrounding this keywood. Let's see how we can use arrow functions instead of 'self'.hack.

script.js:

```
//ES6
const box6 = {
    color: 'green',
    position: 1,
    clickMe: function() {
        document.querySelector('.green').addEvent-
        -Listener('click', () => {
            var str = 'This is box number ' +
            this.position + ' and it is ' +
            this.color ;
            alert(str);
        });
    }
}
box6.clickMe();
```

Then we'll get the required output. It is better

to use arrow functions when you need to preserve the value of the 'this' keyword.

Let's look at another example:

~~//ES5~~
script.js:

```
function Person(name) {
    this.name = name;
}

//ES5
Person.prototype.myFriends5 = function(friends){
    var arr = friends.map(function(el) {
        return this.name + ' is friends
        with ' + el;
    });

    console.log(arr);
}

var friends = ['Bob', 'Jane', 'Mark'];
new Person('John').myFriends5(friends);
```

Output:

▶ [" is friends with Bob", " is friends with Janie",
" is friends with Mark"]

The name is not defined here. This is because of the same reason as before.

The myFriends5 method, we have access to the 'this' variable and it points to the name of the person, which is 'John'. But, we call another function inside of the method. So in this, the 'this' keyword is not going to point to the object, it's going to point to the global object, which is window. We can use the trick we used in the last example: the 'self' variable hack. But there's an another way of doing this.

Remember the bind, call and apply methods which we can use to set the 'this' keyword manually. Bind creates a copy of

the function while 'call' actually calls it immediately. This is how it looks:

script.js:

```
//ES5
Person.proto.....
    :
    var arr = .... {
        return .....
    }.bind(this));
    console.log(arr);
}
:
```

Now, it will work. We created a copy of the function with the 'this' variable set to the anonymous function. So we have created a copy of the function using bind method and we set the 'this' variable to 'this'.

Let's do the same thing in ES6:

script.js:

```
//ES6
Person.prototype.myFriends6 = function (friends){
    var arr = friends.map (el => `${this.name}
    is friends with ${el}`);
    console.log (arr);
}
new Person ('Mike'). myFriends6 (friends);
```

We'll get the required output

9. Destructuring:

Destructuring gives us a very convenient way to extract data from a data structure like an object or an array.

Imagine that we had an array filled with some data and we want to store each of the elements of that array in a single variable.

script.js:

```
//ES5
var john = ['john', 26];
var name = john[0];
var age = john[1];

//ES6
const [name, age] = ['John', 26];
console.log(name);
console.log(age);
```

This also works with objects.

```
//ES6
const obj = {
    firstName: 'John',
    lastName: 'Smith'
};

const { firstName, lastName } = obj;
console.log(firstName);
console.log(lastName);
```

→ These names should be same

We can rename the data of an object.

script.js:

```
// ES6

const obj = {
    firstName: 'John',
    lastName: 'Smith'
};

const { firstName: a, lastName: b } = obj;

console.log(a);
console.log(b);
```

(new names ← a, b)

Let's now try to return multiple values from a function

script.js:

```
// ES6

function calcAgeRetirement(year) {
    const age = new Date().getFullYear() - year;
    return [age, 65 - age];
}
```

```
const [age2, retirement] = calcAgeRetirement(1990);

console.log(age);
console.log(retirement);
```

10. Arrays in ES6/ES2015:

A lot of things were added to arrays in ES6.

script.js:

```
//Arrays.

const boxes = document.querySelectorAll('.box');

//ES5
var boxArr5 = Array.prototype.slice.call(boxes);
boxArr5.forEach(function(cur) {
    cur.style.backgroundColor = 'dodgerblue';
});
```

The querySelectorAll method doesn't return an array. It returns a node list. So we transformed it into an array.

There's a better way in ES6.

script.js:

```
// ES6.
Const boxesArr6 = Array.from(boxes);
Array.from(boxes).forEach (cur =>
    cur.style.backgroundColor = 'dodgerblue');
```

Let's now look at loops.

- When we want to loop over an array, we use the for each or map method. The problem with them is that we cannot break from them. We can't use the continue statement as well.

- Suppose that we wanted to change the text in the boxes, let's write that in ES5 & ES6 ways.

script.js:

```
//ES5
for (var i=0; i< boxesArr5.length; i++){
    if (boxesArr5[i].className === 'box blue'){
        continue;
    }
```

```
    boxesArr5[i].textContent = 'I changed
        to blue!';
}

//ES6
for (const cur of boxesArr6) {
    if(cur.classList.contains('blue')) {
        continue;
    }
    cur.textContent = 'I changed to blue!';
}
```

This is a for-of loop.

And also, there are two new array methods in ES6 that allow us to find elements in an array.

Suppose that we have a group of children and we know that only one of them is of full age. Let's now find out who and

how old that person is.

script.js:

```
//ES5
var ages = [12, 17, 8, -21, 14, 11];
var full = ages.map (function (cur) {
    return cur >= 18;
});
Console.log (full);
console.log (full.indexOf (true));
console.log (ages [full.indexOf (true)]);
```

Output:

▶ [false, false, false, true, false, false]

3

21

findIndex returns the index of the array where the callback function returns true.

Find returns the actual value

```
//ES6
console.log (ages.findIndex (cur => cur >= 18));
console.log (ages.find (cur => cur >= 18));
```

Output: 3

21

11. The Spread Operator:

The Spread operator is used to expand elements of an array in places like arguments and function calls.

Let's create a simple function which adds four values.

script.js:

```
function addFourAges(a, b, c, d) {
    return a + b + c + d;
}

var sum1 = addFourAges(18, 30, 12, 21);
console.log(sum1);
```

Imagine that we had these four numbers in an array. How would we pass that entire array into the function.

```
//ES5
var ages = [18, 30, 12, 21];
var sum2 = addFourAges.apply(null, ages);
```

'this' variable is null here.

```
Console.log (sum2);

//ES6
Const  sum3 = addFourAges (... ages );
console.log (sum3);

Const  familySmith = ['John', 'Jane', 'Mark'];
Const  familyMiller = ['Mary; 'Bob', 'Ann'];

const  bigFamily = [... familySmith, ... familyMiller];
Console.log (bigFamily);
```

We can add another element in the middle if we want:

```
Const bigFamily = [... familySmith, 'Lily', ...familyMiller];
```

Consider we want to change the textColor of the page we're working on:

```
const  h = document. querySelector ('h1');
Const boxes = document.querySelectorAll ('.box');
const  all = [h, ... boxes ];
Array. from (all). forEach ( cur => cur.style.color
                                  = 'purple');
```

We put all the tags in the same structure with spread operator.

We transformed the node list generated by the spread operator into an array and we looped through it. with forEach method.

```
Array.from(all).forEach(cur => cur.style.color='purple')
```

from(all) — converts into an array

cur — argument to the function

cur.style.color='purple' — changing the color of cur.

12. Rest Parameters :

Rest parameters allow us to pass an arbitrary number of arguments into a function and use these arguments in that function. Rest parameters look exactly like the spread operator. The spread operator takes an array and transforms it into single values while the rest parameters receive a couple of single values and transforms them into an array. when we call a function with multiple parameters.

Suppose we want to create a function that receives an arbitrary number of years and then prints to the console whether each person corresponding to these years is of full age or not.

script.js:

```
//ES5
function isFullAge5() {
    var argsArr = Array.prototype.slice.call(
        arguments);
    argsArr.forEach(function (cur) {
        console.log((2016 - cur) >= 18);
    })
}
isFullAge5(1990, 1999, 1965);
isFullAge5(1990, 1999, 1965, 2016, 1987);

//ES6
function isFullAge6(...years) {
    years.forEach(cur => console.log(2016- cur)>=18));
}
```

isFullAge6(1990, 1999, 1965, 2016, 1987):

The big difference between the spread operator and the rest parameters is actually the place in which we use each of them. The spread operator is used in the function call. The rest operator is used in function declaration. to accept an arbitrary number of parameters.

We'll add something to the code. We'll accept another parameter which will act as the age limit. So instead of having the 18 here, we will pass a parameter which is going to tell us at which age a person becomes full of age.

Scoipt.js:

```
//ES5
function isFullAge5(limit){
    Var argsArr = Array.prototype.slice.call(
        arguments, 1); This takes the arguments from index 1.
```

```
    argsArr.forEach(function(cur) {
        console.log((2016 - cur) >= limit);
    })
}

isFullAge5(16, 1990, 1999, 1965);

//ES6
function isFullAge6(limit, ...years) {
    years.forEach(cur => console.log(2016 - cur)
                                >= limit));
}

isFullAge6(16, 1990, 1999, 1965, 2016, 1987);
```

13. Default Parameters:

Default parameters are used whenever we want one or more parameters of a function to be preset, when we want them to have a default value.

Let's look at how this looks like in ES5 and ES6.

script.js:

```
//ES5

function SmithPerson(firstName, yearOfBirth, lastName,
                                                nationality) {
    lastName === undefined ? lastName = 'Smith':
        lastName = lastName;
    nationality === undefined ? nationality = 'american':
        nationality = nationality;

    this.firstName = firstName;
    this.lastName = lastName;
    this.yearOfBirth = yearOfBirth;
    this.nationality = nationality;
}

var john = new SmithPerson('John', 1990);
var emily = new SmithPerson('Emily', 1983, 'Diaz',
                                                'spanish');
```

```
//ES6
function SmithPerson(firstName, yearOfBirth,
    lastName = 'Smith', nationality = 'american'){
  this.firstName = firstName;
  this.lastName = lastName;
  this.yearOfBirth = yearOfBirth;
  this.nationality = nationality;
}

var john = new SmithPerson('John', 1990);
var emily = new SmithPerson('Emily', 1983,
                'Diaz', 'spanish');
```

14. Maps:

Map is a new data structure in ES6. A map is a key-value data structure. In objects, we use only strings as keys. But in maps, we can use any datatype for keys. We can even use functions or objects as keys.

script.js:

```
//Maps.

const question = new Map();

question.set('question', 'What is the official name of the latest major Javascript version?');

question.set(1, 'ES5');
question.set(2, 'ES6');
question.set(3, 'ES2015');
question.set(4, 'ES7');
question.set('correct', 3);
question.set(true, 'Correct answer :D');
question.set(false, 'Wrong answer, please try again!');

console.log(question.get('question'));
console.log(question.size);

// question.delete(4); -> This deletes '4'.

if (question.has(4)) {
    console.log('Answer 4 is here')
}
```

```
//question.clear(); →This clears all data
                     from the map.

//question.forEach((value, key) => console.log(
  `This is ${key}, and it's set to ${value}`));

                  deconstructor
                                  returns all entries of
                                  our question map.
for (let [key, value] of question.entries()){
  if (typeof(key) === 'number') {
    console.log(`Answer ${key}: ${value}`);
  }
}

const ans = parseInt(prompt('Write the
    correct answer'));
console.log(question.get(ans === question.get(
                                'correct')));
```

15. Classes:

Classes make it easier to implement inheritance and to create objects based on blueprints. In ES5, these blueprints are called function constructors

script.js:

```
//ES5
var Person5 = function(name, yearOfBirth, job){
    this.name = name;
    this.yearOfBirth = yearOfBirth;
    this.job = job;
}

Person5.prototype.calculateAge = function(){
    var age = new Date().getFullYear - this.
                                    yearOfBirth;
    console.log(age);

var john5 = new Person5('John', 1990, 'teacher');

//ES6
class Person6 {
    constructor (name, yearOfBirth, job) {
        this.name = name;
        this.yearOfBirth = yearOfBirth;
        this.job = job;
    }
```

```
    calculateAge () {
        var age = new Date().getFullYear -
                                this.yearOfBirth;
        console.log(age);
    }

    static greeting () {
        console.log('Hey there!');
    }
}

Const john6 = new Person6 ('John', 1990, 'teacher')

Person6.greeting ();
```

The static greeting()... is a static method. Static methods are methods that are attached to the class but not inherited by the class instances ie., by objects that we create through that class.

Things to note:

→ Class definitions are not hoisted. So unlike function constructors, we need to first implement a class and only later in our code, we can start using it.

→ We can only add methods to classes but not properties. This isn't a problem because inheriting properties through the object instances is not a best practice anyway.

16. Classes with subclasses:

Let's implement inheritance between classes ~~and~~ using subclasses.

Consider the slide of Person and Athlete when we first talked about inheritance. We had an example where we had two function constructors that can inherit from one another. We have the more generic class person for all persons and

Then we have the more specific subclass for an athlete. Because an athlete is also a person but with some more specific attributes and methods.

script.js:

```
//ES5
var Person5 = function (name, yearOfBirth, job){
    this.name = name;
    this.yearOfBirth = yearOfBirth;
    this.job = job;
}

Person5.prototype.calculateAge = function(){
    var age = new Date().getFullYear() -
                this.yearOfBirth;
    console.log(age);
}

var Athlete5 = function (name, yearOfBirth,
    job, olympicGames, medals) {
    Person5.call(this, name, yearOfBirth, job);
```

```
    this.olympicGames = olympicGames;
    this.medals = medals;
}

Athlete5.prototype = Object.create(Person5.
                                   prototype);

Athlete5.prototype.wonMedal = function(){
    this.medals++;
    console.log(this.medals);
}

var johnAthlete5 = new Athlete5('John', 1990,
    'swimmer', 3, 10);

johnAthlete5.calculateAge();
johnAthlete5.wonMedal();
```

Output: 26

11

Have a look at this line:

```
Person5.call(this, name, yearOfBirth, job);
```

To understand what we are doing here, we need to remember how the 'new' operator works, which is the operator that we use to create a new instance. So, when creating a new athlete object, 'new' creates a new empty object, calls the athlete function constructor and sets the this keyword to the newly created empty objects. So, in execution context that we're in here, the 'this' keyword will point to the new empty object. Now, if we want a person property's name, year and job to be set on the new athlete object, then we need to call the person function constructor with the this keyword also set to our newly created athlete object. After this, all the properties will be set in the new

athlete object that's created by the new operator and that's why we need to call it here and to set the 'this' variable to 'this'.

And then, to create the correct prototype chain, we used object.create. Object.create allows us to manually set the prototype of an object and we want the prototype of the athlete to be the prototype of the person so that they will be connected.

Let's now do this in ES6.

script.js:

```
//ES6
class Person6 {
    constructor (name, yearOfBirth, job) {
        this.name = name;
        this.yearOfBirth = yearOfBirth;
        this.job = job;
    }
```

```
    calculateAge() {
        var age = new Date().getFullYear() -
                  this.yearOfBirth;
        console.log(age);
    }
}

class Athlete6 extends Person6 {
    constructor(name, yearOfBirth, job, olympicGames,
        medals) {
        super(name, yearOfBirth, job);
        this.olympicGames = olympicGames;
        this.medals = medals;
    }

    wonMedal() {
        this.medals++;
        console.log(this.medals);
    }
}
```

```
const jonhAthlete6 = new Athlete6 ('John',
                    1990, 'swimmer', 3, 10);
johnAthlete6.wonMedal();
johnAthlete6.calculateAge();
```

Output: 11

26.

Here, 'super' will simply call the superclass. We don't have to manually set any 'this' variable or anything. Then we've added our method: wonMedal().

17. Coding Challenge 8:

Do it!!