Siddharth Singha Roy
Winter 19-20

Next Generation JavaScript

Up until this point we where using ES5. In this section we will be using ES6 or ES2015 and the more recent versions.

Currently we can use ES6 and newer features because they are converted into ES5 by a process known as polyfilling and transpiling.

LET AND CONST

Instead of var now we have two new ways in which we can initialise a variable

1. let : Like the old var
2. const : Whose value we do not want to change

Var is function scoped whereas let and const our blocked scope.

Block means anything that is wrapped between two curly braces. Earlier the scoped change only when a function was called. However now it has changed. Even if/else has a different scope.

```
const check;
check = 5;
```

This is not allowed for const (allowed for let).

In ES5 we had code hoisting where we could access a variable even before it was declared. However in ES6 although we have code hoisting this is not allowed. A variable cannot be accessed before it has been defined. This is called the temporal dead zone.

```
let i = 23;
for(let i = 0; i<5;i++)
    console.log(i);
console.log(i);
```

The last console.log will print 23. This is because let and const are block scope. So the I declared in the for loop has a different scope than the I declared in the window object.

Use const for variables that would not change their value over the execution course.

Since let and const are block scoped, then cannot be accessed in the global scope ie. the window object.

Blocks and IIFEs

In ES6 we have block scoping which can be used to improve data privacy. To create a new block simply wrap them inside a curly brace. Thus we do not need to have a for loop or if/else to create a block, just a pair of curly braces.

```
{
        // New block is created
}
```

This simply replaces IIFE.

Strings

<u>Template Literals</u>

Instead of writing variables and the string separately we can combine them into one using this.

```
`${name} is a good boy. He is ${calcAge()} years old.`
```

We now have four new important methods

```
1. console.log(name.startsWith("Sid"));
2. console.log(name.endsWith("oy"));
3. console.log(name.includes("ing"));
4.      console.log(name.repeat(5));
        console.log(`${name} `.repeat(5));
```

Arrow

```
map(function(ele) {   // Callback function
        return ele-20;
});
```

```
map(ele => ele-20);
```

Here we are passing only **one argument** and then **returning it in one line.**
We cannot do this for other cases.

```
map((ele, index) => ele-index);
```

2 arguments and **returning it in one line.**

If we only have one line of code inside the function then the return will implicit.

```
map((ele,index) => {
        const date = new Date().getFullYear();
        return `${date} and ${ele-index}`;
});
```

2 arguments and multiple lines of code.

Even for 0 arguments we need a ()

Here we need to write the return keyword.

This in arrow functions

Arrow functions in JavaScript do not have their own **this** keyword. They simply use the this keyword of the function that they are written in. So we say they have a **lexical this** keyword.

```
var obj6 = {
  color:'green',
  calc:(function() {
    return() => {
      console.log("Hi   " + this.color); // this will refer to the function inside which we have
the nested function
    }
  })
};

obj6.calc()();
```

We are able to access this inside the nested function because of lexical this keyword. It will use the this keyword of the function in which it is defined. The function in which it is defined has access to the this instance of the object because it is an object method call.

```
var obj6 = {
        color:'green',
        calc: () => {
            return() => {
                console.log("Hi    " + this.color); // this will refer to the function inside which we
have the nested function
            }
        }
    };

    obj6.calc()();
```

Now this.color will return undefined. Since we change the calc to an arrow function it no longer will take the this keyword from the object instance but will now use the lexical this keyword. It will use the this keyword of the function in which it is defined. Thus the this will now point to the global object.

A callback function inside an object method will also not have access to this. (ES5)

Destructuring

```
var arr = ['John', 18];
const [name, age] = arr;
console.log(name, age);
```

This is destructuring.

Destructuring can also work with objects.However here we need to make sure that the keys of the object match with the variable names.

To destructure arrays we use [] . To destructure objects we have to use the {}.

```
const mark = {
    name: "Sid",
    age: 19,
    fn: ()=>17,
    arr:[1,2,3],
}

const {name, fn, age, arr} = mark;
console.log(name, age, fn, arr);
```

To destructure objects we need to ensure that the object key and the variable name are same. If we do not want this we can use

```
const {firstName : a, lastName : b} = mark;
```

Now we can access the values using a and b.

To return multiple values from a function we can return an object or an array and then destructure the result to get the individual value.

Spread Operator

Expand elements of an array.

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

ages = [1,2,3,4];
console.log(addFour.apply(this, ages));
```

This is how we use the apply method. The **this** can be replaced with null as we are not interested in it. So what the apply method did was take the array and then passed each element of the array into the addFour function. This is how it is different from the call method.

But we have a better way of doing it in ES6. What we will do use the spread operator. What the spread operator does is it will resolve the array into its individual components.

```
ages = [1,2,3,4];
console.log(addFour(…ages));
```

Adding 2 arrays using spread operator

```
const arr1 = [1,2,3,4];
const arr2 = [5,6,7,8];

const arr = [...arr1,...arr2];
console.log(arr);
```

Rest Parameters

Rest Parameters allow us to pass an arbitrary number of elements into a function.

The rest operator has the same notation as that of spread however their functionalities are completely the opposite. The spread operator expands an array into its individual components whereas rest operator converts individual components into an array.

```
function sum(...arr) {
   let sums = 0;
   for(const ele of arr)
   {
      sums+=ele;
   }
   console.log(sums);
}
sum(1,2,3,4,5);
```

The rest parameter is used in the function declaration to accept an arbitrary number of elements. The spread operator is used in function call.

```
const checkAge = (limit, ...years) => {
    years.forEach((ele) => {
        console.log(ele>=limit);
    })
}

checkAge(19,16,12,18,14,20,23,26,85);
```

Default Parameters

```
const details = (name, age, birthYear = new Date().getFullYear(), nation = "India") =>
console.log(name ,age, birthYear, nation);

details("Siddharth", 19);
```

MAPS

Maps is a new data structure that has been added in ES6. It is a key-value structure which earlier could only be done via objects. The advantage of hash maps over objects is that we can have the key of the map as anything we want. In objects we are limited to string as keys. Here in maps we can have number, boolean, objects and even functions as key.

Maps are also utterable unlike objects ie. we can loop through maps.

```
const question = new Map();
question.set("question", "What is 1+1");
question.set(1, "2");
question.set(2, "0");
question.set(3, "I will be deleted");
question.set(true, 1);
question.set(false, "Try again");

// console.log(question.get("question"));
// console.log(question.get(1));
// if(question.has(3))
// {
//     console.log(question.get(3));
//     question.delete(3);
// }
// question.clear();
// console.log(question.size);

// question.forEach((value, key) => {
//     console.log(key, value);
// })

for( const [key, value] of question.entries()) // .entries() specifies that we want the key-value
pair
{
    if(typeof(key) === 'number')
        console.log(key,parseInt(value)); // Convert to num
}
```

Classes

This is a new addition to the language however does not add anything new. What it does is make the syntax shorter for inheritance.

We have something known as static methods inside the classes. These methods a re not inherited by the instances of the class. These are attached to the class definition.

One of the major difference between function construction and classes is that classes is not hoisted. We cannot access class before its declaration.

We can only add methods to classes not properties. But this is not a problem at all. Because inheriting property to object instances is not the best practice anyway.

```
// var Person = function (name, birthYear) {
//     this.name = name;
//     this.birthYear = birthYear;
// }

// Person.prototype.calcAge = function() {
//     this.age = new Date().getFullYear() - this.birthYear;
// }

// var Athlete = function ( name, birthYear, medals) {
//     Person.call(this, name, birthYear);
//     this.medals = medals;
// }

// // First use Object.create then create a new prototype  for athlete

// Athlete.prototype = Object.create(Person.prototype);

// Athlete.prototype.addMedal = function() {
```

```
//     this.medals++;
// }

// var mark = new Athlete("Mark", 1993, 23);
// mark.calcAge();
// mark.addMedal();
// console.log(mark);


class Person {
    constructor(name, birthYear) {
        this.name = name;
        this.birthYear = birthYear;
    }

    calcAge() {
        this.age = new Date().getFullYear() - this.birthYear;
    }

    static greet() {
        console.log("Hiiiii From Person");
    }
}

class Athlete extends Person {
    constructor(name, age, medals){
        super(name, age);
        this.medals = medals;
    }
    addMedals() {
        this.medals++;
    }
}
```

```
const john = new Athlete("John", 19, 23);
john.calcAge();
john.addMedals();
// john.greet();  Static method is not inherited
console.log(john);
```