JavaScript - Day 6

THIS KEYWORD :-

→ "this" refers to the object on which we call our function :

example :-

sayHello has 'this' (just like all other functions) and it's value is obj

```
obj.sayHello(this)
```

obj is the object on which we are calling sayHello.

** Inside of a function the value of 'this' is the object we used to call the function.

* What if we simply say
> sayHello(), we are not calling sayHello on any object.
> In this case, 'this' refers to the window object.

→ So just saying sayHello() is like saying window.sayHello()

someObject.someFunction()

① Look before the dot what's there?
② It's someObject → the value of 'this'
③ If there is not a . means directly calling function, then look at it as window.someFunction()
↳ what's before the dot is window internally.

*** → On global execution context the value of 'this' is window.

* Example 1 :-

```
function sayHello () {
    console.log (this);
}

> sayHello ()
```

output > window ⇒ [Because sayHello is called w/o any object, internally it's window (window.sayHello ())].

* Example 2

```
const obj = {
    name : "Rahul",
    sayHello : function () {
        return 'Hi' + this.name;
    }
}
```

→ If we want access to name in sayHello this is how we can do it.

→ Since sayHello is a part of obj, that value of 'this' inside sayHello is obj.

→ To run it we will use

```
obj.sayHello()
```

and what's before that.? It's 'obj', that means value of 'this' inside sayHello should be 'obj'.

* Example 3 :-

→ Execute same code for multiple object (Reuse).

```
function sayHello() {
    console.log(this.name);
}
const obj1 = {
    name: 'Rahul',
    sayHello: sayHello
};
const obj2 = {
    name: 'John',
    sayHello: sayHello
};

const name = 'Lara'
```

> window.sayHello() OR sayHello()
→ Lara

> obj1.sayHello();
→ 'Rahul'

> obj2.sayHello()
→ John

* Conclusion:-

→ This gives method (functions inside your object) access to the object they are defined in.

→ let's execute same code (sayHello) by multiple objects (obj1 & obj2)

* Example4 :

```
const a = function () {
    console.log ('a', this);
    const b = function () {
        console.log ('b', this);
        const c = {
            sayHello: function () {
                console.log ('c', this);
            }
        }
        c.sayHello ();
    }
    b ();
}
a ();
```

> a ⇒ window
(because there is nothing before a (), it's effectively window.a())

> b ⇒ window
(same case as a)

NOTE :- 1) value of 'this' inside b is window means it does NOT MATTER WHERE WE DEFINED FUNCTION B, what matters is how it's called.

2) But Javascript follows lexical (static) scoping right? scope of a function is decided by where it is defined and not when it's called.

3) No wonder 'this' is like an alien, that follows dynamic scope, value of this depends on how we called the function not where it is actually defined.

> c → c

Inside sayHello the value of this is the object c because we called sayHello as

c.sayHello()

* Example 5

```
const obj = {
    name: 'Rahul'
    sayHello: function() {
        console.log("Hello", this);
        var sayBye = function() {
            console.log("Bye", this);
        }
        sayBye();
    }
}

> obj.sayHello();
```

Output > Hello obj
> Bye window

→ So how do we solve this problem and make it work as lexically scoped, value of this should depends on where the function was defined because that's how rest of Javascript works.

SOLUTION is Arrow Functions

→ If you convert sayBye to an arrow function out will be

> Hello obj
> Bye obj

• Why?

→ Because inside arrow function value of this depends on ~~what~~ & where that arrow function is present or defined and it's defined inside obj. right? so, the value of this will be obj.

Wait...!!! Arrow functions ~~were~~ were introduced in ES6 right? what did people do before that?

→ To get the same result with normal functions as arrow function you can use the bind method

sayBye.bind(this)

Your basically binding the current value of this which is (obj) to sayBye as well.

* In order to play with 'this' keyword, we have 3 methods.

call(), apply(), bind().

→ Internally every function uses call() when it's invoked.

→ So when you call a function by sayHello()
you are basically doing
sayHello.call() ⇒ [Internally]

→ Let's take an example of a game, where you have a battery life (of player) and in some situations the player can charge itself to 100%.

```
const player1 = {
    name: 'John',
    battery: 62,
    charge: function () {
        this.battery = 100;
    }
}
```

→ so basically player1 is an object with a method charge (to charge 100%)

→ To simply charge player1 we can do

→ player1.charge()

[battery : 100]

Now itself introduce a second player ⇒ player2

```
const player2 = {
    name = 'Lara',
    battery = 20
}
```

⇒ OKAY, player2 does not have a charge method, so can it borrow method, so can from player1, Also note we don't want to repeat code that can be reused.

⇒ Call the charge method of player1

player1.charge.call(player2)

↓

But call it on player2

→ We are esentially able to pass reference to player2 (which will be value of this) inside battery method.

→ So argument to call is overiding what the value of 'this' will be when that method is invoked.

→ Also you can pass arguments using call which will be received by battery method.

```
player1.charge.call(player2, 20, 30, 40)
```

arguments.

and your battery function might look like this.

```
battery: function (args1, args2, args3)
```

NOTE:- you don't need to accept player2 as an arguments 'this' already contains player2.

- **apply**

→ It's pretty much same as call but the way it takes arguments is different.

→ It takes an array of arguments.

```
player1.charge.apply(player2, [20, 30, 40]);
```

takes an array of arguments.

- **bind() :-**

→ call and apply immediatly invoke the function, where as bind returns a new function binded with a new 'this' which we can run later.

```
const callLater = player1.charge.bind(player2, 20, 30, 40);
```

[syntax same as call]

→ to actually call it we can later do

```
callLater();
```