Intro – Async Execution

Ex: 01

```
var asd = 10

function xyz(params) {
    console.log(asd)
}

function abc(params) {
    console.log(asd)
}

xyz()
abc()
```

Output:

10

10

So, what we understand is both functions are able to access the variable **`asd`**. JavaScript is based on Lexical scoping; meaning whatever variables are declared along with the function in the same scope, the functions are able to access the variables.

Ex:02

```
function xyz(params) {
    var innerVariable = 10;

    function abc(params) {
        console.log(innerVariable)
    }

    abc()
}
var fn = xyz()
fn()
```

In my function **`xyz()`**, declared a variable and function **`abc()`**. The **`abc()`** function **`console.logs`** the **`innerVariable`**. This **`innerVariable`** is declared inside **`xyz()`** and not inside the **`abc()`** function. However, if we call **`xyz()`** in our global scope; this in turn will define the variable and initialize it with the value 10 and then it'll declare the **`abc()`** function inside itself and whatever function which is declared is being called.

Output:

10

Meaning, even though my function **`abc()`** doest have any declaration for this **`innerVariable`**, however, this **`abc()`** is still able to access my **`innerVariable`** from my parent function.

With all this in my mind, we can say ***a closure***, *is a function which remembers everything in its lexical scope.* Meaning, for the **`xyz()`** function, lexical scope is everything outside it. **`innerVariable`** is in the lexical scope of **`abc()`** function and not **`xyz()`** because **`xyz()`** function is in the local scope of **`abc()`**.

Ex: 03

```
function xyz(params) {
    var innerVariable = 10;

    function abc(params) {
        console.log(innerVariable)
    }
    return abc
}
var fn = xyz()
fn()
```

`abc()` function was returned in the variable **`fn`** because the moment we call **`fn`**, it will go through the entire code and since the function **`abc()`** is known it would be returning the **`abc()`** function definition will be returning. This entire definition is stored in the variable fn. The **`fn`** variable is storing a function, so we can call it by **`fn()`**.

After **`xyz()`** is executed, **`abc()`** being inside my **`xyz()`**; if a function is executed everything inside is removed from the memory. Let us add another function or replace **`abc()`** function with **`increase()`**

Ex: 04:

```
function xyz(params) {
    var innerVariable = 10;

    function increase() {
        innerVariable++
        console.log(innerVariable)
    }
    return increase
}
var fn = xyz()
fn()
```

Output:

11

When `xyz()` was called, function `increase()` got returned and then using `fn()`, the `increase()` function was called which will increase the variable 10 and then it will log the variable. Now, since all the execution is done, `fn()` is still in the global scope, referring to the `increase()`.

If we call it in the console, the output is 12. We call again the output would be 13. Even though `xyz()` is executed and removed from the memory, but the inner function was returned in the variable `fn()` and if we keep calling it, `fn()` will be calling increase function. Since `increase()` had access to the `innerVariable`, so it will remember the variable.

It will create a closure with that variable which was declared inside my function. Meaning, every function, always remembers everything which is declared in its lexical scope or in the surrounding scope.

Ex:05

```
function xyz(params) {
    var innerVariable = 10;

    function increase() {
        innerVariable++
        alert(innerVariable)
    }
    return increase
}
var fn = xyz()
fn()
```

Output:

Give an alert box:11

The var `fn` was declared in the window object. In the console, `fn` would be declared with the `increase()`. Notice, it is the same function that it was calling. Now, if we expand upon it and then in the [[Scopes]], we have two properties.

0: Closure (xyz) {innerVariable: 11}
1: Global

The main function we called was `xyz()`. In the `increase()`, we see that it is access variable which is declared in `xyz()`. That is why, if we expand ***0: Closure (xyz)***, we can see ***{innerVariable: 11}***.

If we call the `fn()` again, it will give an output 12. Meaning, it incremented the `innerVariable`. If we go back to the window object and check the scopes in the `fn`, it would show ***{innerVariable: 12}*** in the ***Closure(xyz)***. It remembers, whatever operations are done in the inner function and if it is doing any changes to the `innerVariable` in the parent function. When we return a function, also gets returned with whatever variables were inside the function.

```
function xyz(params) {
    var innerVariable = 10;
    var innerVariable2 = "My String"

    function increase() {
        innerVariable++
        alert(innerVariable)
    }
    alert(innerVariable2)
    return increase

}
var fn = xyz()
fn()
```

Ultimately, the functions would be declared along with the variables. It will execute `alert()` and then it would return the `increase()` function to `fn`. After that `fn()`

function is being executed. In **`fn(),`** it is increasing the **`innerVariable`** and then alerting that **`innerVariable`**. So, if we go back and refresh, we would get the ‘My String’ and once we click on OK, then the **`fn()`** function would be called and the alert box shows 11.

In the window object, inside the scopes of **`fn`**, only **`innerVariable`** is found, innerVariable2 is not found because the **`increase()`** is only accessing **`innerVariable`**. But if alert **`innerVariable2`** inside the **`increase(),`** then we can find **`innerVariable2`** as well inside the closure. So, whatever variables inner function needs to access, only those variables it will try to remember.

Since it was only accessing the **`innerVariable`**, only that got registered in the scope, but once we add **`innerVariable2`**, now my inner function is also trying to access **`innerVariable2`**, so now the closure saves the second variable value as well.

```
function x() {

    var text = '25'
    return function () {
        return function() {
            text + '100'
            console.log(text)

        }
    }
}
x()()()
```

Output:

25

The function x() would be read in the first pass, then it is called and executed. In execution, the var text value is declared and store the value 25 in it. Then it will return the function. The function that just came in would be executed right away that would be another function. This would be executed, and since text + 100 is not being stored and it would not do anything. Then the next statement console.log(text) would be executed and the output would be 25.

```
var card_number = '468765243218'

function pay(params) {
    //logic to complete payment
    console.log('Payment Complted with card number ' +
card_number)
}
pay()
```

Output:

Payment Completed with card number 468765243218.

If we expand on the window object, we will be able to see the card number and also its value. Another thing is pay() is also defined in the window object. So, if we scroll down fast enough, we can see the pay function as well.

In the window object if we enter "card_number" = 'zasdfweoih' and then call the pay() function, then the output would be,

Payment Completed with card number zasdfweoih.

So, if the password is being saved in the global context, then someone can actually change that or someone can write a script and get the password of that user. In order to remove this issue, we can approach or create a specific pattern of calling all these functions.

In that pattern, closure is being used as the main id. Since card_number is accessible to the surrounding scope, we can

```
var app = (function() {
    var card_number = 468765243218

    function pay() {
        console.log('Payment Complted with card number' +
card_number)

    }
    return pay
})()
```

Now if we expand the window object, we won't be able to see the **card_number**. Even if we try to search for it, the console will say it is not defined. We declared **app**(), which can be found in the window object. So, if we call **app**(), it would directly give out the console.log statement.

We called the anonymous function. Inside this function we have defined a variable and a function in it, which is using the variable **card_number** which is declared inside its parent function. Finally, we are returning the saved function. So, this entire thing is being saved in the variable app. In the closure properties **pay**() is getting called and accordingly **card_number** is getting logged. This solves privacy issues. We can have multiple functions;

```
var app = (function() {
    var card_number = 468765243218

    function pay() {
        startPayment()
        console.log('Payment Completed with card number' +
card_number)
    }

    function startPayment() {
        console.log('Payment started with card number' +
card_number)
    }

    return pay
})()
```

The **pay**() function knows about the **startPayment**(), because they are in the lexical scope. So, now if we call **app**(), we get

```
Payment Started with card number 468765243218
Payment Completed with card number 468765243218
```

If we go back again in the window object, we only get reference to the **pay**(), but it is still able to call the **startPayment**(), which behaves as a private function. We have not given access to the **startPayment**() through interface. Instead of the function if are returning a property as

```
return {
    pay: pay,
    startPayment: startPayment }
```

In the window object, app is now an object instead of a function and if we expand upon it we get 2-properties, pay and **`startPayment`**. In the other case, which was just returning only the {pay: pay}, now app is an object with only pay property defined. We are able to safe guard our variables inside a closure, as well as able to defined private functions which my outer functions would not be able to access, only whatever is put in the object or returned from the anonymous function will be exposed to the window object.

setTimeout():

The **`setTimeout()`** method is used to execute a function after waiting for specified time interval. This method returns a numeric value that represents the ID value of the timer. Unlike the **`setInterval()`** method, the **`setTimeout()`** method executes the function only once. This method can be written with or without the ***`window`*** prefix.

We can use the **`clearTimeout()`** method to stop the timeout or to prevent the execution of the function specified in the **`setTimeout()`** method. The value returned by the **`setTimeout()`** method can be used as the argument of the **`clearTimeout()`** method to cancel the timer.

Parameter:

- **function:** It is the function containing the block of code that will be executed.
- **milliseconds:** This parameter represents the time-interval after which the execution of the function takes place. The interval is in milliseconds. Its default value is 0. It defines how often the code will be executed. If it is not specified, the value **0** is used.

```
function bhookamp() {
    console.log("Chat on FIRE!!!")
}
bhookamp()
```

Let’s say we have some use case where in we wanted this function to be called at a later point in time. Instead of calling it directly, we will use

```
setTimeout(bhookamp, 5000) // 5000 → 5 seconds
```

Now, when we refresh, the script would load after 5 seconds.

setInterval ():

The **`setInterval()`** method in JavaScript is used to repeat a specified function at every given time-interval. It evaluates an expression or calls a function at given intervals. This method continues the calling of function until the window is closed or the **`clearInterval()`** method is called. This method returns a numeric value or a non-zero number that identifies the created timer.

Unlike the **`setTimeout()`** method, the **`setInterval()`** method invokes the function multiple times. This method can be written with or without the ***`window`*** prefix.

Parameter:

- **function:** It is the function containing the block of code that will be executed.
- **milliseconds:** This parameter represents the length of the time interval between each execution. The interval is in milliseconds. It defines how often the code will be executed. If its value is less than 10, the value 10 is used.

```
function bhookamp() {
    console.log("Chat on FIRE!!!")
}

setInterval(bhookamp, 5000)
```

When we refresh, the script gets executed and will keep executing after every 5 seconds unless it is stopped.

Ex:06

```
function add(a,b) {
    console.log(a+b)
}

setInterval(function() {add(20,50)},2000)
```

Ex:07

```
function add(a,b) {
    console.log(a+b)
}
var count = 0
var timerid = setInterval(function() {
    if (count < 10) {
        add(20,50);
        count++
    }else {
        clearInterval(timerid)
    }
},2000)
```

Output:

(3) 70

Even if we omit the clearInterval(), however, the setInterval() will keep using the memory. But if we use the clearInterval() then the setInterval is stopped and the memory will be cleared.