ADVANCED js

Fat arrow functions (the new anonymous)

Now in ES6 the old anonymous functions are changed..

OLD:

Var a = function () {awesome function};

NEW:

Var a = () => {}

If you want to take any parameter as input you put it in the () parenthesis.

Now there are some issues with it which we will learn about in the callback topic.

SetTimeout(func,time)/SetInterval(func,ms)

These functions only accept functions and ( , ) time in milliseconds

Set Timeout returns a numeric value which is a unique handler.

Now suppose you have a timeout which is active but somehow the purpose of your timeout is achieved before it even executed or you simply want to cancel the timeout because the user cancelled it too. So how do you do that?

Basically JS remembers your timeout by the specific handler that it generated. If you want to clear the timeout simply use..

clearTimeout(handler);

Same applies for setInterval too.. clearInterval().

CallBacks:

Suppose we have a program which looks like this:

var x = 1, y = 2;

var b = calc();

function calc() {

var a = null;

setTimeout(()=>{

a = x + y;

return a;

},50);

}

Now when we run this it returns undefined because the function is void and timeout did not run until the function ran.

If we place the return outside the timeout it returns null.

IN COMES THE CALLBACK.

We create such a function (read callback) which takes the calc function as input and calls the calc function from within the timeout.

The callback is basically just a function that takes the answer and returns it.

var num1 = 1

var num2 = 2

var num3 = calc(showTotal);

function calc(myCallBack) {

var total = null;

setTimeout(() => {

console.log('adding async')

total = num1 + num2;

num3 = total;

myCallBack();

}, 50);

}

function showTotal() {

console.log('SUM Result ', num3);

}

Calback can also take a value as input and return any operations performed on it

Now suppose you are performing a network call and you do not know the number of arguments (parameters) passed in the callback.

Spread Operator (three dots prefix to the variable)

Var myCb = (…arguments){

Console.log(…arguments)

}

It takes all the data and displays it in a NEW and open array..

Like if a was an array and b was an array it concatenates them..

Var c = […a,…b];

Similarly for objects too.

We could also make a new array by using forEach and push but the problem is that it takes a lot of time for longer arrays/objects.

In ES5 you would use slice for arrays. Or .toString and then split(‘,’)

In ES6

For objects we Use

Var a = Object.assign(b);

This will reference the b object in the new object a;

Can also concatenate 2 arrays in a reference.

But if we want to create a new object or array which is not referenced then we define the data type required at the end (breaking the reference)

Var a = Object.assign(b,c, []);

Old browsers do not run es6 so we have libraries that convert es6 into es5.

JSON.stringify() bhi use kr skte hain.

TO convert object values to values..

Var { name, age } = obj // which is an object having name, age and location as properties.

Var { name, age , …extra} = obj // this saves all the leftover properties in the extra variable which is an array

API Call (network calls)

2 methods used in HTML GET & POST

Data from a API call is usually in JSON (objects & arrays) and XML( old apis ).

Call an API and you get an object.

api.github.com/users/hassan-imran/

api.github.com/users/hassan-imran/followers/ //etc

# Old Method – XHR (returns JSON file and callback)

Then

# JQUERY – library (returns CallBack, promise)

First call jquery library from the internet (script src in absolute path (also called CDN paths) from jquery.com)

$ 🡺 represents the usage of Jquery

$.get(“link”, cb); // get can be called by browser and POST,PUT, cannot be called by browser

Where var cb = (a,b) {

Console.log(a,b); // in a it gives the JSON data (object) and b gives success or error //(promise)

}

This takes place in the form of asynchronous programming (the rest of the program runs while it also listens to the callback at the same time)

But now In JS we have fetch (Almost all browsers but mostly chrome)

Higher order functions

Returns a functions from within a function

Function myFunc () {

Return () => {

Return 123;

}

}

Either call in 2 steps.

Var a = myFunc();

a();

or simply call

myFunc()();

eg multiplier(2)(3); // returns 6

now if we want to use a multiplier with a specific value we use..

var double = multiplier(2);

now when we call double (3).. it doubles 2

shorthand:

function multiplier(x) => return (factor) => return (x\*factor)

Closure (kinda like local vars)

If we store a function in a variable and the same function in another variable..

The value in the function is not overwritten.. but the var holds, stores the parameter passed in the functions in that specific variable.

Closure issue

Var arr = [5,4,3,2,1];

For (var i=0; i<arr.length;i++) {  
 var x = setTimeout

}

Referential transparency

Does not change any value (in the data) (it does the work on the values but does not change it) and return the same data.

This

Lambda functions => fat arrow funcs

Fat arrow function also uses the this of its parent (baap)

In normal functions this means of that function only not of its parent like we saw in constructor functions

Functional Programming

No shared data

No mutable data (changeable data)

No side-effect (like in async programming)

HTML writing

${} 🡪 This represents an id within the curlies

Like ${container}

Now when we have to concatenate we do not use quotes (“) and plus(+) sign anywhere but simply use

(``) back ticks and write all HTML within it.. the only difference is that we write the IDs in the above manner.

Period sign (.) represents classes

# Declarative programming (intelligent programming)

It has the computations (functions) of all possible scenarios covered. But it does not give the flow that when which function to execute.

# Imperative Programming (unintelligent)

It describes the flow and order about how the functions should run and does not contradict that order.

MAP

It is used to iterate on each element of an array and return a new corrected array whereas the old data (array) also exists

var newArr = myArray.map(any function here to perform on each element);

The function that is passed within the map function must have 2 parameters..

1. Array element (lets say x)
2. Index of the array (lets say y)

ForEach

Same as map but does not return new array and works on the same array.

Like it’s a for loop

Filter

Closures

A closure is an inner function that has access to the outer (enclosing) function’s variables—scope chain. The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function’s variables, and it has access to the global variables.

**A Basic Example of Closures in JavaScript:**

function showName (firstName, lastName) {

var nameIntro = "Your name is ";

// this inner function has access to the outer function's variables, including the parameter

function makeFullName () {

return nameIntro + firstName + " " + lastName;

}

return makeFullName ();

}

showName ("Michael", "Jackson"); // Your name is Michael Jackson

**Closures’ Rules and Side Effects**

1. **Closures have access to the outer function’s variable even after the outer function returns:**  
   One of the most important and ticklish features with closures is that the inner function still has access to the outer function’s variables even after the outer function has returned. Yep, you read that correctly. When functions in JavaScript execute, they use the same scope chain that was in effect when they were created. This means that even after the outer function has returned, the inner function still has access to the outer function’s variables. Therefore, you can call the inner function later in your program
2. **Closures store references to the outer function’s variables**; they do not store the actual value.  Closures get more interesting when the value of the outer function’s variable changes before the closure is called. And this powerful feature can be harnessed in creative ways, such as this private variables
3. **Closures Gone Awry**  
    Because closures have access to the updated values of the outer function’s variables, they can also lead to bugs when the outer function’s variable changes with a for loop

To fix this side effect (bug) in closures, you can use an **Immediately Invoked Function Expression** (IIFE)

Put () at the end of the closure so it returns the value immediately

Objects

Mutable data (that can be changed)

e.g. var and let

Immutable data (cannot be changed)

e.g. constt

Object.freeze

Spread operator

Object.assign

Promise