JavaScript - Day 5

Function Scope Vs Block scope :-

→ we already know about function scope. Everytime we call a function we creates a function scope, and we can only access variables declared inside a function inside it.

→ We can't access them outside the function. Beco2 they are within the function scope.

So what's Block scope?

→ Block scope is the scope created inside {.....} i.e. (curly Brackets).

→ Javascript follows function scope & didn't actually have the concept of block scope, until ES6 was introduced.

```
if (true)
{
  var name = "Rahul";
}
console.log(name);
```

Output > Rahul

→ Because if is not a function right? so, name is still in global scope and is accessible.

→ so if we had

```
function sayName()
{
  var name = "Rahul";
}
console.log(name);
```

Output >> ReferenceError.

Explanation :- Basically when name is ~~defined~~ declared inside function a function scope is created and that's why global scope does not have name it is in function scope).

* However, with the introduction of let and const, they come under block scope.

```
if (true)
{
    let name = 'Rahul';
}
console.log(name);
```

Output >> Reference Error.

→ Because since let and const are block scoped using if statement is like block scope ({...}) these those ~~statement~~ variables are only accesible inside the block scope.

** var still follows function scope.

** let & const follows block scope.

** → Therefore in most situations it's better to avoid var.

Immediatly Invoked Function Expression (IIFE).

→ Before that let's understand why global variables are bad and then how IIFE can help us :)

→ We already spoke about memory leaks.

Okay, If space is the problem, can I use a few global variables.

→ let's say we have ~~introduce~~ add multiple script tags in a file.

```
<script> var count = 1 </script>
<script> var count = 2 </script>
```

→ Both files have a global variables name "count" so now your count will be overwritten by the latest count value which is 2.

THAT's BAD RIGHT?

→ As your codeBase gets larger it's will gets hard to track these name collisions.

* All script tags get combined to one execution context so count is used across all and obviosly it can't have different values for different files.

* IIFE will help us here :-

→ IIFE is a function expression that was like this,

This bracket says this is not a funtn declaration, it's a funtn expression

```
(function () {
    // anonymous function (No name).
})();
```

we are immediatly calling this funtn

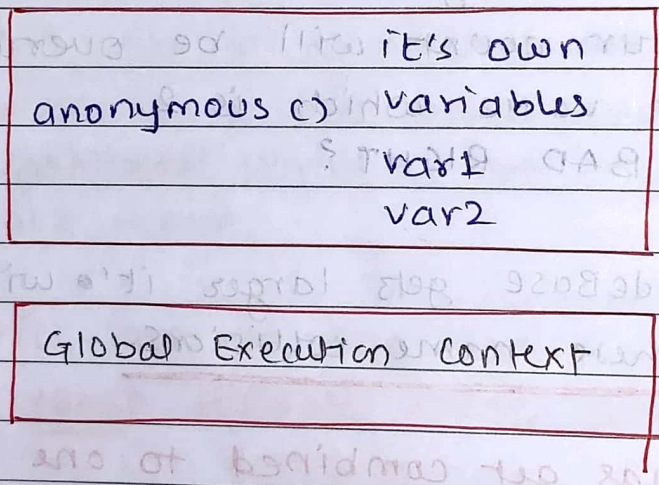
* SUMMARY :-

→ It's an function expression (we already learnt function expression don't get completely hoisted as JE does not look at them as function declaration)

→ That means whatever code we put inside this functions will not be a part of global execution context. since this function itself is not part of global execution context.

→ So far we know whatever code (variable) we declared inside this IIFE are going to be local to this functions.

→ This is How execution context looks



→ so a new execution context is created for this anonymous funation and it has it's own local variables.

→ How do we use IIFE to use data (variables or functions) through the files?

file.js

```
var file1 = (function () {
    var name = 'Rahul';
```

```
function sayHello() {
    return 'Rahul'
}
return {
    name: name,
    sayHello: sayHello,
}
})();
```

→ Since we are immediatly invoking / calling this IIFE file1 variable has the returened data or return value of this IFFE so, we can simply do.

```
file1.name
file1.sayHello()  etc
```

→ But again, we still have a global variable called file1, so these is a better solution in modern Javascript called modules. we will study about it later.