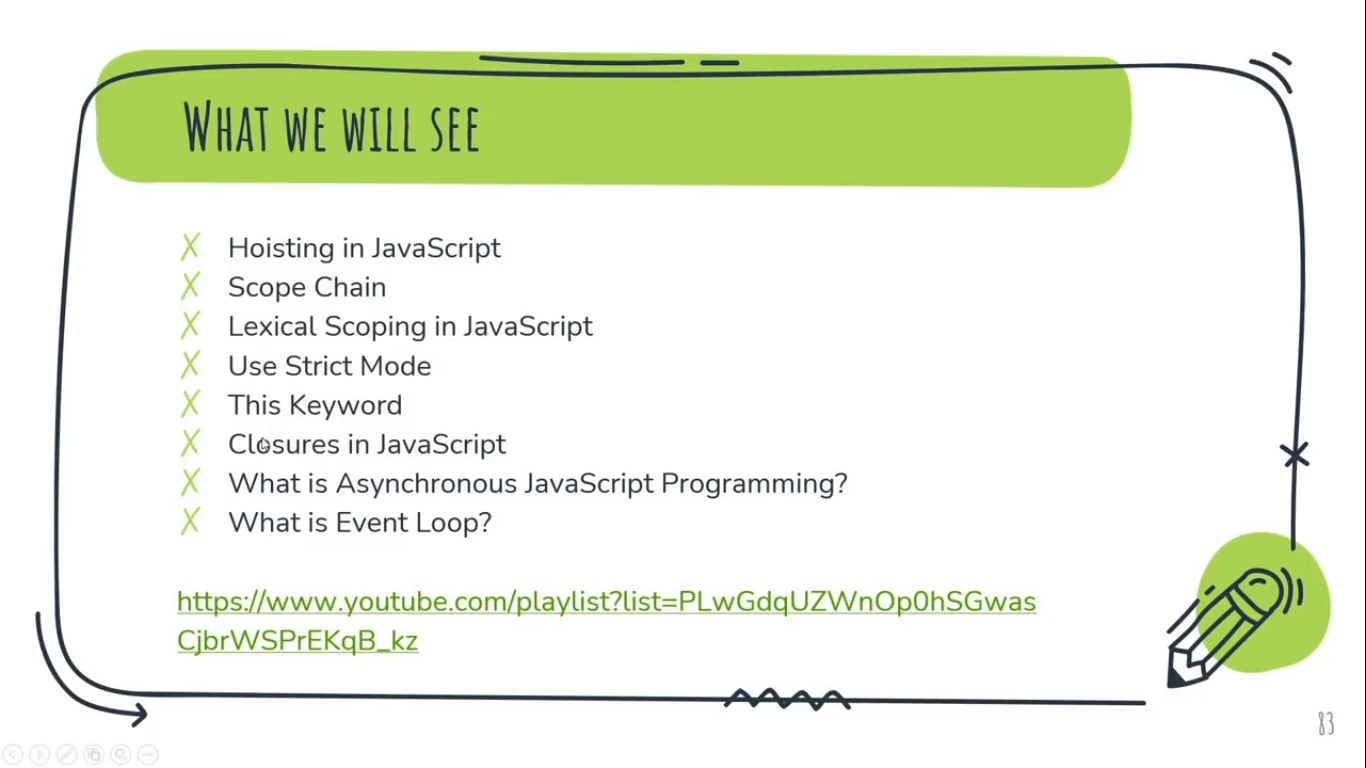
**Asynchronous JavaScript**



**Hoisting in JavaScript:**

We have a creation phase and execution phase.  
Before execution of any code, creation phase comes first then execution phase come in JavaScript.

Hoisting in JavaScript is a mechanism where variables and functions declaration are moved to the top of their scope before the code execute.

**Example:**

console.log(myName);

var myName = “Kumar”; //If nothing is initializing then JS by default initializes ‘undefined’.

myName= "Chandan";

**Output:**  
undefined

**//How it will be in output during creation phase**

**//1st.** var myName ; **//**As per definitation, here first variable is declaring  
**//2nd.** Console.log(myName);  
**//3rd.** myName = “Chandan”; //There is no variable declaring

**//Then comes execution phase, which will execute the following statement.**

**//1st.** var myName ; **//**As per definitation, here first variable is declaring  
**//2nd.** Console.log(myName);

**Note:**

After ES2015 i.e. ES16, hoisting is avoided by using the let keyword instead of var. (The other difference is that variables declared with let are local to the surrounding block, not the entire function.)

So, Hoisting fails with let or const variable and return an error.  
**Example:**

console.log(myName);  
let myName = "Chandan";  
myName= "Chandan";

**Output:**Uncaught ReferenceError: Cannot access 'myName' before initialization

**What is Scope Chain and Lexical Scoping JavaScript?**

The **Scope Chain** is used to resolve the value of variable names in JS.  
Scope chain in JS is lexically defined, which means that we can see what the scope chain will be by looking at the code.  
**Example:**  
const PI = 3.42   
**//Here, we can say by looking at the above code that there is going to assin value *3.42*  at the constant variable *‘PI’*.**

**Lexical Scoping** means Now, the inner function can get access to their parent functions variables. But the vice-versa is not true i.e. We can’t access child/inner function from the parent/outer function.

**Example: // Example of Lexical Scoping**let a = " Hi!"; **//Global Scope**

const first = ()=>{ **//Parent Function**  
 let b = " How are You?";

const second = ()=>{ **//Child Function**  
 let c = " This is Chandan.";  
 console.log(a + b + c); **//Because of Lexical Scoping we can access all**   
 }

second();

}

first();

**Output:**Hi! How are You? This is Chandan.

**Example: // Without Lexical Scoping**let a = " Hi!"; **//Global Scope**

const first = ()=>{ **//Parent Function**  
 let b = " How are You?";

const second = ()=>{ **//Child Function**  
 let c = " This is Chandan.";  
 }  
console.log(a + b + c); **//We can’t access the variable of child fun outside its scope**

second();

}

first();

**Output:**Uncaught ReferenceError: c is not defined

**Closure in JavaScript**

A closure is an inner function that has access to the outer Function’s variables / parameters as well as the Global Variables.  
 OR

A closure is the combination of a function and the lexical environment within which that function was declared.

OR

A closure is a function having access to the parent scope. It preserve the data from outside.

For every clousre we have three scopes:-

* Local Scope
* Outer Function’s Scope
* Global Scope

**Example: //Sum of Three Numbers using Closure**

let c = 10; **//Global Scopes**

const outerFun = (a)=>{ **//Parent Scope or Outer Function’s Scope**

let b = 10;

const innerFun = ()=>{ **//Inner Function’s Scope**  
 let sum = a+b+c;  
 console.log(`The sum of Three Number is ${sum}`);

}

return innerFun; **//Returning Only Inner Function**

}

let inner = outerFun(5);

console.dir(inner);  
inner ();

**Output:**innerFun() **//Returned inner Function**  
The sum of Three Number is 25 **//Result of Sum of Three Nos.**

**//**Whenever, we call a function, an execution context created. Just like here in the above example, When outerFun Call, it assign the value of ‘a’ and ‘b’ in the execution context and directly return the innerFun and moveout from the execution context or say from the program. (Because when a funtion return any statement, means there will be no furhter execution of that function takes place.)  
Then after calling of innerFun as inner(); takes place gives the result of sum of three number (a, b, c) i.e. ‘The sum of Two Number is 25’.  
Now, A question is arrise, after moving out of outerFun from the execution context, then How I got the value to ‘a’ and ‘b’ from outerFun to the innerFun to gives the result of sum of three number (a, b, c) i.e. ‘The sum of Two Number is 25’, means innerFun accessing the data/variable of already executed outer function from the program.   
 So, this is where ‘**Closure’** comes into play because after complete execution of outerFun from the programe it get stored (along with their data or say Variables and their values) under the Closure like this:   
Closure (outerFun)  
a: 5  
b: 10

**Note:**

* The return statement does not execute the inner function. Function is executed only when followed by (), but the return statement returns the entire body of the function.
* Closure is very much similar to the Lexical Scoping. Moreover in the both, Parent scope cannot access inner function’s scope.

**I.Q. Difference Between Synchronous and Asynchronous JavaScript?**

**Ans:-**Synchronous JavaScript:  
When two works are executing at a time then one work will not start execution until another one completely executed means one work has to wait until other one completely executed. It follow First In First Out (FIFO) Algorithm. The first code come in the queue will firstly executed. It let works to be wait or say stop until other not fully executed.   
**Example:**const fun2 = ()=>{  
 console.log('This is Function-2');  
}

const fun1 =()=>{  
 console.log('Thsi is Function-1');  
 fun2();  
 console.log('This is Function-1 Again');  
 }

fun1(); **//This Function will Call First becuase, this lying at Global Scope**

**Output:**Thsi is Function-1  
This is Function-2  
This is Function-1 Again

Asynchronous JavaScript:  
When two works are executing at a time, then work with less execution time will execute first than the other one means one work doesn’t have to wait until the other work completed its execution. Asynchronous JS doesn’t let any work to wait or stop even for a sec.  
**Example:**const fun2 = ()=>{  
 setTimeout(()=>{  
 console.log('This is Function-2');  
}, 2000);   
}

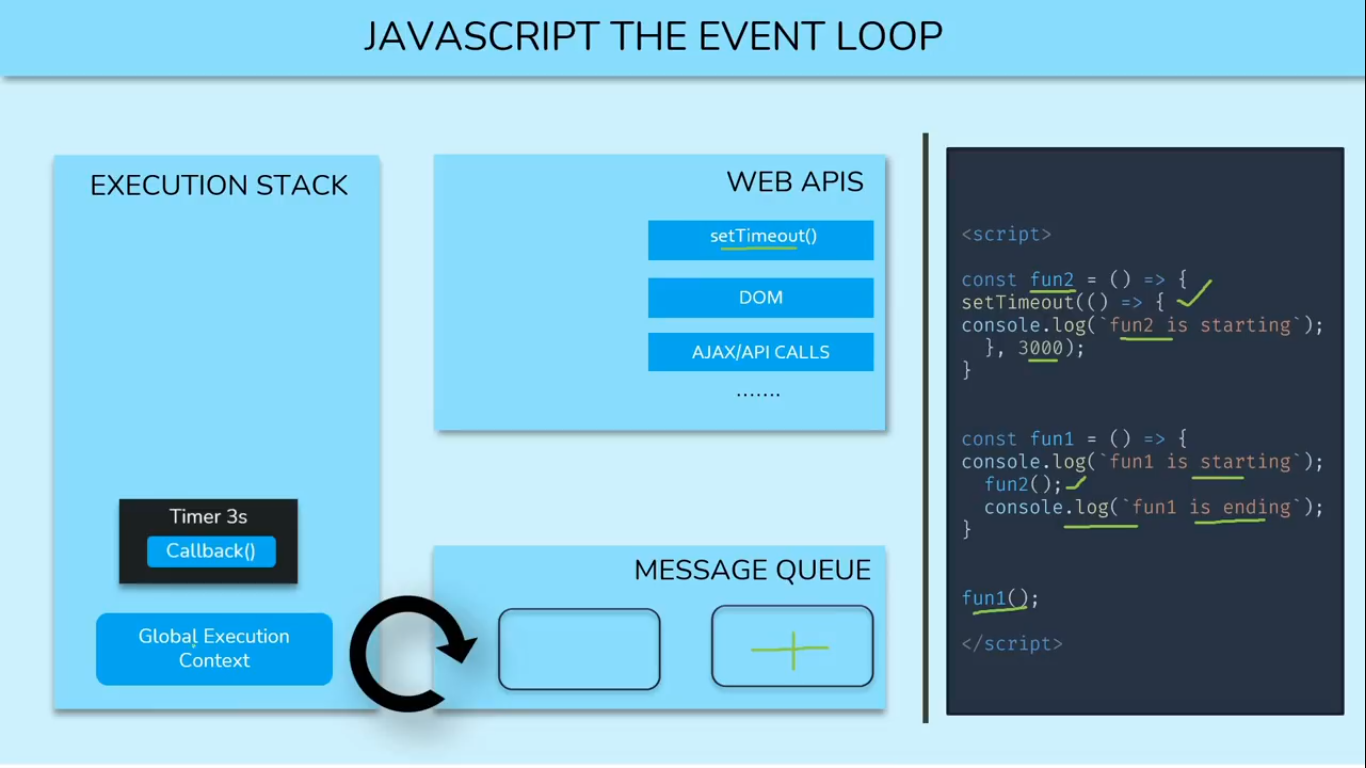
const fun1 =()=>{  
 console.log('Thsi is Function-1');  
 fun2();  
 console.log('This is Function-1 Again');  
 }

fun1(); **//This Function will Call First becuase, this lying at Global Scope.**

**Output:**Thsi is Function-1  
This is Function-1 Again  
This is Function-2

**//**Here, In the Output ‘This is Function-1 Again’ print/come first before ‘This is Function-2’ which means, it doesn’ t wait for the ‘This is Function-2’ for 2 seconds to be executed. This is what Asynchronous Programming.

**Q. What is Event Loop?**

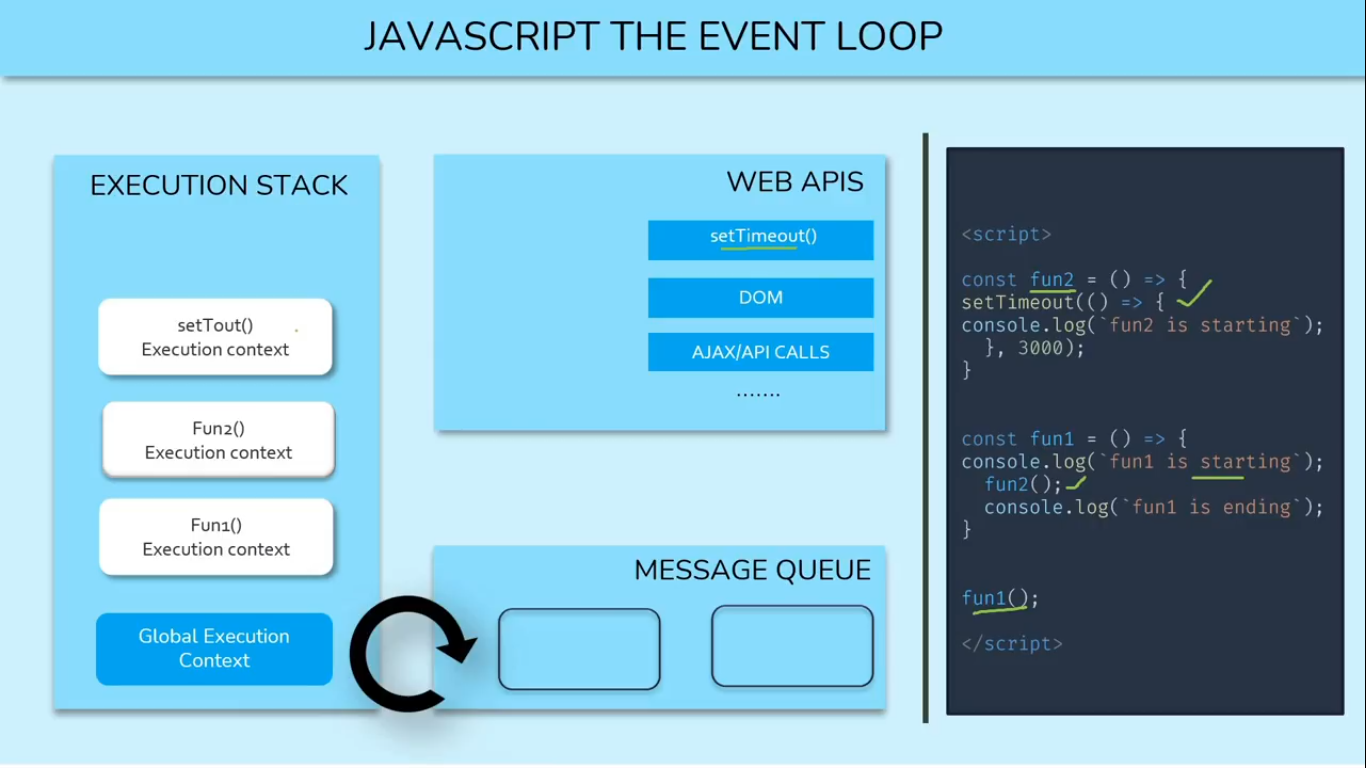


Event loop is the loop which transfer the required instruction from the Message Queue taken from the Web APIs to the Execution Stack for its execution.

Lets discuss with an Asynchronous Example,  
**Example:**const fun2 = ()=>{  
 setTimeout(()=>{  
 console.log('This is Function-2');  
}, 3000);   
}

const fun1 =()=>{  
 console.log('Thsi is Function-1');  
 fun2();  
 console.log('This is Function-1 Again');  
 }

fun1(); **//This Function will Call First becuase, this lying at Global Scope.**

**Output:**Thsi is Function-1  
This is Function-1 Again  
This is Function-2  
  
**Discuss:**   
As we already know, whenever a funciton call, it creates an execution context in the Execution Stack.  
So when fun1 call it creates an execution context in the execution stack. Then after since console.log() is itself a function or method, so now it will creates another execution context above fun1’s execution context in the execution stack. When it executed successfully, it get removed from the execution stack.  
Now, fun2 will call and creates another execution context in the execution stack above the executon context of fun1. Now setTimout is a method of window object which also work as function, so its execution context will also create in the execution stack.  
**Note:**siteTimeout() method, All DOM methods, AJAX/ API Calls contain by Web APIs.

So, Now setTimeout from the execution stack to the Web APIs. But Since setTimeout is a Event timer method/function which call its function after some given time in its second parameter. So timer i.e. (3sec) of setTimeout for calling back to its inner function i.e. (CallBack function) will pass to the Web APIs and execution context of setTimeout will remove from the execution context.

Now, the remain execution context in the execution stact will be of fun2 and then fun1. Now, fun2 which will further check, if there exist any instruction remain in it to execute or not. If not it will get remove from the execution stack.   
Now, the remaining execution context in the execution stack would be of fun1. Now, fun1 which will further check, if there exist any instruction remain in it to execute or not. But Here, another instruction is remain in fun1 i.e. console.log() function, so it will create another execution context above the fun1 in the Execution Stack then execute it and then remove its execution context from the execution stack.  
Now, fun1 which will again check, if there exist any instruction remain in it to execute or not. If not it will get remove from the execution stack.

Now, there will be nothing remain in the execution stack but On the other hand, Web APIs still keep holding the timer of setTimeout method. At the moment when timer finishes its time, Web API immedietely pass it next to the Message Queue.

**Note:** Queue is a Data Storage in Data structure which follow First Come First Serve (FCFS) algorithm.

Now, Message Queue will check is there something exist in Execution Stack or not. If not It will pass timer(Containing callback Function) next to Execution Stack for its execution using “**Event Loop”**. Now the execution context of console.log() method/function (log is the method of console object, which contains some function to simply print the result into console) containing by callback function of setTimeout method will create in the Execution Stack. After its successful execution on the console it will remove from the execution stack.

And this is how we got the final output like the following:  
**Output:**Thsi is Function-1  
This is Function-1 Again  
This is Function-2