05.00

06.00

07.00

JS:- CallBacks and Promises

✉ @ ☎

```
function fun(x, fn){
    for(let i=0; i<x; i++){
        console.log(i);
    }
    fn();
}
```

```
fun(10, function fn(){
    console.log("I am also exec");
});
```

fun() is a HOF

fn() is a callback function.

HOF consumes some function as an argument.

If the func. that you pass as an argument is a callback function.

i.e. the func. that I am passing as an argument inside the HOF is a CallBack function.

Problems with CallBacks:—

1. Inversion of control
2. CallBack Hell.
   - Readability problem
     (The code becomes very hard to understand what it actually wants to do)

Inversion of Control

Discussed and Implemented later on.

2017 WEDNESDAY

02 AUGUST

32nd Week • 214-151

AUGUST 2017

Wk	M	T	W	T	F	S	S
32		1	2	3	4	5	6
33	7	8	9	10	11	12	13
34	14	15	16	17	18	19	20
35	21	22	23	24	25	26	27
36	28	29	30	31			

Async Programming with JS:-

09.00

① JS is Synchronous in nature

② JS is single threaded.

10.00

11.00

12.00

→ Default nature of JS is that it behaves synchronously.

01.00

point 1

JS is going to execute code line by line. And if there is some piece of code that is going to take a lot of time, JS is going to stop there give that code whatever amount of time it needs to execute and then only move forward.

02.00

03.00

All of this is valid if we execute valid ecmascript code which is given in the standards.

04.00

05.00

point 2 ex.

```
console.log("The Start");

for(let i=0; i<100000000; i++){
    Some task
}

console.log("the End")
```

07.00

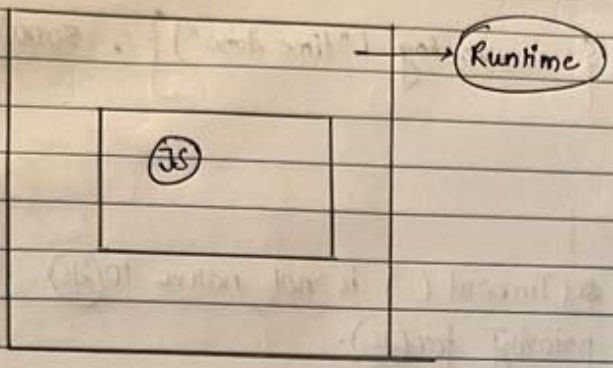
o/p

The Start

for loop executing → so JS will wait.

The End

✉ @ ☎



Whenever we run a valid (JS) code, there is (Runtime) that is reqd to execute this piece of code.

Runtime:-

(1) In order to run a valid piece of (JS) code. There are a lot of libraries that are reqd.
All of these libraries are given to (JS) by the run-time.

(2) (Browser) is a runtime.

⊙ Browser provides a lot of features those are not native to JS.

⊙ Using JS, we can modify html, provide animations, provide css of a webpage.

All of these functions are **not native to (JS).**

All of these functions are being provided to (JS) by (Browser)

That is why, (JS) is synchronous in nature for the piece of code which is native to (JS).

```
console.log("Hi");
setTimeout(function(){ console.log("time done")}, 5000);
console.log("Bye");
```

o/p
Hi
Bye
time done

- setTimeout() is not native to (JS). Its not the priority for (JS).
- (JS) Knows how to execute this.
- (Runtime) feeds this functionality to (JS).

```
function timeConsumingByLoop(){
    console.log("loop Start");
    for(let i=0; i<10000000000; i++){
        # Some Task;
    }
    console.log("loop ends");
}

function timeConsumingByRunTime(){
    console.log("Starting Timer");
    setTimeout(function(){
        console.log("completed the Timer");
    }, 2000);
}
```

```
console.log("Hi");
Time Consuming By Loop();
Time Consuming By Run Time();
Time Consuming By Loop();
console.log("Bye");
```

O/P
Hi
Loop Starts
Loop Ends
Starting Timer
Loop Starts
Loop Ends
Bye
Completed the Timer.

Few pointers before we move forward.

(A) Event Loop:- Event Loop is an infinite loop that will keep on checking

(1) if our callstack is empty or not

(2) Global piece of code is done or not.

(B) If Run-time has completed its task, we do not care. if we have something left in our callstack or there is some global piece of code still left to be executed.

SUNDAY 06

Therefore, if Runtime complete its task, then what is going to be expected to be done will wait inside Event Queue.

(C) When everything is done (callstack empty & no global code left to be executed). Then we are going to pick one by one from the Event Queue push it in callstack and execute it.

2017 MONDAY

07 33rd Week • 219-146 AUGUST

AUGUST 2017

Wk	M	T	W	T	F	S	S
32		1	2	3	4	5	6
33	7	8	9	10	11	12	13
34	14	15	16	17	18	19	20
35	21	22	23	24	25	26	27
36	28	29	30	31			

Dry Running :—

Example - 01

```
(1) function timeConsumingByLoop() {
09.00
    (1A) console.log("Loop Starts");
    (1B) for(let i=0; i<10000000000; i++){
             # Do something
         }
11.00
    (1C) console.log("Loop ends");
12.00
}
01.00
(2) function timeConsumingByRunTime(){
02.00
    (2A) console.log("Starting Timer");
03.00
    (2B) setTimeOut(function exec(){
04.00
             console.log("Completed the Timer");
         }, 5000);
05.00
}
06.00
~~console.log("Hi");~~ (3) Global piece of code
07.00
(Hi) gets printed.

(3) console.log("Hi");
(4) timeConsumingByLoop();
(5) timeConsumingByRunTime();
(6) timeConsumingByLoop();
(7) console.log("Bye");
```

O/P

Hi
loop starts
Loop ends.
Starting timer
Loop starts.
Loop ends.
Bye
Completed the Timer

(4) We are making a function call.

Inside our callstack a new entry will be made.

Once, this entry is made, we go from line (4) to line (1).

timeConsumingByLoop

callstack

(1) Inside (1), we have (1A) ~~console.log~~ "loop starts" gets printed.

SEPTEMBER 2017

	M	T	W	T	F	S	S
36					1	2	3
37	4	5	6	7	8	9	10
38	11	12	13	14	15	16	17
39	18	19	20	21	22	23	24
40	25	26	27	28	29	30	

TUESDAY 2017

33rd Week • 220-145

AUGUST 08

09.00 (1B) It is a (FOR) loop. This is a native piece of JS code.
This piece of code will take 2~3 secs to execute.

10.00 Since, this is a Native piece of (JS) code.

11.00 JS will block here. Till the time the (FOR) loop is done executing.

12.00 ie JS will wait here and not move forward.

01.00 Once, the (FOR) loop ends. We move forward.

02.00 (1C) We will print "Loop Ends".

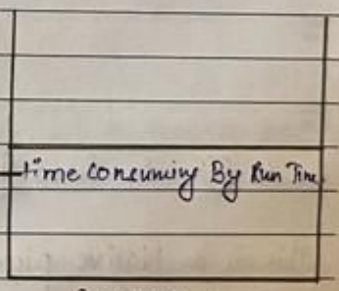
03.00 And after this, the function timeConsumingBy Loop is done executing.

04.00 ∴ We remove the entry of this function from the callStack.

05.00 → CallStack now becomes empty.

(5) This is another function call.
∴ We create an entry of this function in our call stack.

07.00 Once, this entry has been made we will go to line (2).



CallStack.

(2) Inside (2), we have (2A).
∴ Starting Timer gets printed.

(2B) This piece of code is NOT Native to (JS). This feature is given to me by my

✉ @ ☎ Run-time.

What (JS) will do is. (JS) will notify (Run-time) that (JS) has gotten an instruction to start a timer. So, can (Run-time) start a timer of 5 secs. Post which (JS) is going to execute Excel().

And then, (JS) comes back.

Now, behind the scene a timer will start in the Run-time.

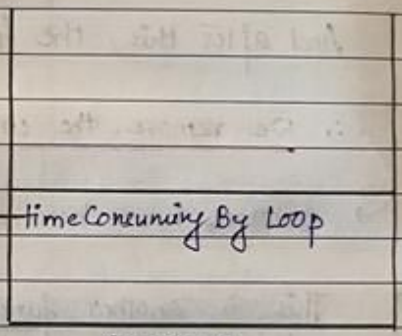
After (2B), the function timeConsumingByRunTime ends.

Simultaneously, we remove the entry from the callstack. And so, our callstack gets empty.

(6) Another function call.

We will make an entry corresponding to this func call in our callstack.

Once, this entry has been made we go to Line (1).



callstack.

(1) Inside (1), we have (1A). (Loop Start) gets printed.

(1B). This is a Native piece of code known to (JS).

∴ (JS) will block here. And it will not move forward until and unless this (FOR) loop is done executing.

Now, let's say at this time the timer has struck 5s (let's say the timer struck 5s & still the FOR loop is executing)

Even, then JS will not leave the (FOR) loop in between.

SEPTEMBER 2017

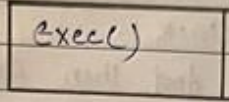
WK	M	T	W	T	F	S	S
36					1	2	3
37	4	5	6	7	8	9	10
38	11	12	13	14	15	16	17
39	18	19	20	21	22	23	24
40	25	26	27	28	29	30	

THURSDAY 2017

33rd Week • 222-143

AUGUST 10

So, what Run-Time will do is, (Run Time) will send the CallBack to the Event Queue.



Event Queue.

⇒ This will help (JS) to keep track of things that are still to be executed.

Now, the FOR Loop ends.

"Loop Ends" gets printed.

Now, the function time consuming By loop is done executing.

Simultaneously, we remove it's entry from our callstack. So, our callstack gets empty.

Now, at this point our callstack is empty.

But, Event Loop finds out that a global piece of code line (7) is yet to be executed. So, "By" gets printed.

After this callstack is empty and there is no global piece of code to be executed.

Note Whatever CallBacks we have in our Event Queue, will be executed. if and only if there is empty call stack and there is no global piece of code left.

Now,

⊙ Event Loop is going to pick one callBack at a time from the eventQueue.

⊙ Push it to the callstack.

And then starts its execution.

∴ Completed the Timer gets printed.

Note

If we have other callBacks waiting in the EventQueue. They will keep on waiting until & unless, the current callback is done executing and the callstack gets empty.

—: Example-02 :—

```
① function timeConsuming By Loop () {
    (1A) console.log ("Loop Starts");
    (1B) for (let i=0 ; i<10000000000 ; i++) {
            # Do something   (10sec)
         }
    (1C) console.log ("Loop Ends");
}

② function TimeConsuming By RunTime0 () {
    (2A) console.log ("Starting Timer");
    (2B) setTimeout ( " function execute ) {
            (2C) console.log ("Completed Timer 0");
            (2D) for (let i=0 ; i<10000000000 ; i++) {
                    # some task
                 }
         }, 5000 );   (5sec timer)
}

③ function timeconsumRunTime1 () {
    (3A) console.log ("Starting Timer");
    (3B) setTimeOut (func.execute) {
    (3C) console.log ("completed time1");
         }, 0 );   (0sec timer)
}

④ function timeconsumRunTime2 () {
    (4A) console.log ("Starting Timer");
    (4B) setTimeOut (func.execute) {
    (4C) console.log ("completed time 2");
         }, 200 );   (200 ms timer)
}

⑤ console.log ("Hi");
⑥ timeConsuming By Loop();
⑦ timeConsuming By RunTime0();
⑧ time Consuming By RunTime1();
⑨ time Consuming By RunTime2();
⑩ time Consuming By Loop();
⑪ console.log ("Bye");
```

SEPTEMBER 2017

WK	M	T	W	T	F	S	S
36					1	2	3
37	4	5	6	7	8	9	10
38	11	12	13	14	15	16	17
39	18	19	20	21	22	23	24
40	25	26	27	28	29	30	

SATURDAY 2017

33rd Week • 224 - 141

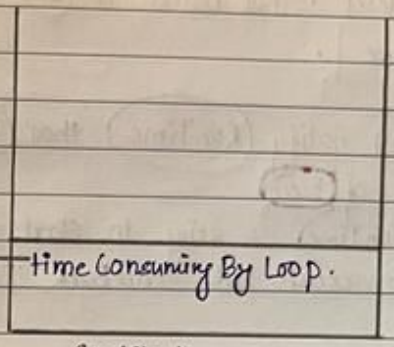
AUGUST 12

09.00 (5) Global piece of code :- (Hi) gets printed

O|P

10.00 (6) Function call :- We will make an entry to the callstack.

Hi



11.00

12.00

01.00 Once, this entry has been made, we will go to line (1).

02.00

03.00 Inside (1) we have (1A). (Loop Starts) gets printed.

Loop Starts.

04.00 (1B) We have a (FOR) loop. It is native to (JS). So (JS) will block here. Meaning, unless and until this (FOR) loop is done executing (JS) will not move forward.

05.00

06.00 After this (FOR) loop ends.

07.00 (1C) (Loop Ends) gets printed.

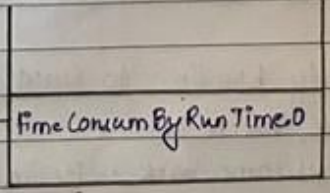
Loop Ends.

After this, we are done executing the func. timeConsuming By Loop. Simultaneously, we will delete the entry corresponding to this function from the callstack.

SUNDAY 13

Now, our callstack is empty.

(7) Function call :- We will make an entry corresponding to this func. call in our callstack.



Once, this entry has been made we will go to line (2).

2017 MONDAY

14 34th Week • 226-139 AUGUST

AUGUST 2017

Wk	M	T	W	T	F	S	S
32		1	2	3	4	5	6
33	7	8	9	10	11	12	13
34	14	15	16	17	18	19	20
35	21	22	23	24	25	26	27
36	28	29	30	31			

O/P

Starting Timer.

(2) Inside (2), we have (2A) ; (Starting Timer) gets printed.

(2B). This codeset is not native to (JS). This is a feature given to (JS) by [Run-Time].

(JS) will notify (RunTime) that (JS) has gotten an instruction to start a timer of (5sec)

Now, (RunTime) is going to start a timer of (5sec). Post which (JS) is going to execute the callBack ExecO().

Now, (JS) will come back. Behind the scenes a (5S) timer has started.

Now, the function execution is done.
So, we will remove this entry from our CallStack.
Now, our callStack gets empty.

(B). Function Call :— We will make an entry to our callstack.

time Consuming By RunTime !

Once, the entry has been made we will go to line (3).

CallStack.

(3). Inside (3), we have (3A) ; (Storing Timer) gets printed.

Starting Timer

(3B). This codeset is not native to (JS). This is a feature given to (JS) by (Run-Time).

(JS) will notify Run'time to start a timer of (0sec) Post which (JS) is going to execute Excel().
Now, JS will come back. Behind the scenes a (0s) timer has started.
The moment we come back, the runtime has struck 0s.

∴ Run time has done completing its task of OS.

∴ Excel is pushed to the EventQueue.

↳ We pushed it to EventQueue and not execute it because there is still global piece of code left.

~~Completed Timer gets printed.~~

The function is done executing. We will remove the entry from our callstack. So, our callstack gets empty.

(9) Function call:- We will make an entry to our callstack.

Once, this entry has been made, we will go to line (4).

time Consuming By RunTime 2

Callstack.

(4). Inside (4), we have (4A). (Starting Timer) gets printed.

(4B). This piece of code is not native to (JS). This is a feature provided by (Run-time).

(JS) will notify (RunTime) to start a timer of (200ms). Post which (JS) is going to execute the callBack Exec2().

Now, (JS) will come back. Behind the scenes 200ms timer has started.

The func. is done executing. We will remove its entry from call stack. Our call stack gets empty.

(10). Function call:- We will make an entry to our callstack.

Once this entry has been made we will go to line (1).

time Consuming By Loop.

CallStack.

O/P.

~~Completed Timer~~

Starting Timer.

AUGUST 2017

Wk	M	T	W	T	F	S	S
32		1	2	3	4	5	6
33	7	8	9	10	11	12	13
34	14	15	16	17	18	19	20
35	21	22	23	24	25	26	27
36	28	29	30	31			

o/p

① 09.00 Inside ①, we have (1A). (Loop Starts) gets printed.

Loop Starts.

(1B). This is a native piece of code known to (JS).

(JS) will block here for (10s) and not move forward until and 11.00 unless the (FOR) loop is done executing.

12.00 But at this point the timer of (5secs) and timer of (200ms) is done executing.

01.00

So, their corresponding callBacks will be passed inside the Event Queue.

02.00

Exec2() correspondiy to 200ms will be pushed first.

03.00

Exec0() correspondiy to 5secs will be pushed second.

04.00

∴ Our Event Queue will currently have the following arrangement.

05.00

Exec1()	Exec2()	Exec0()	

06.00

Event Queue.

07.00

(1C) - (Loop ends) gets printed.

Loop Ends.

This func. is done executing. We will remove its instance entry from our callstack.

Now our callstack gets empty.

At this point, Event loop will check the following :-

① Is our callstack Empty? (Yes)

② Is there any Global piece of code left? (Yes)

∴ (Yes) we will still not start executing from Event Queue.

O/P
Bye.

09.00 (11). (Bye) gets printed.

10.00 Event Loop checks the following :—

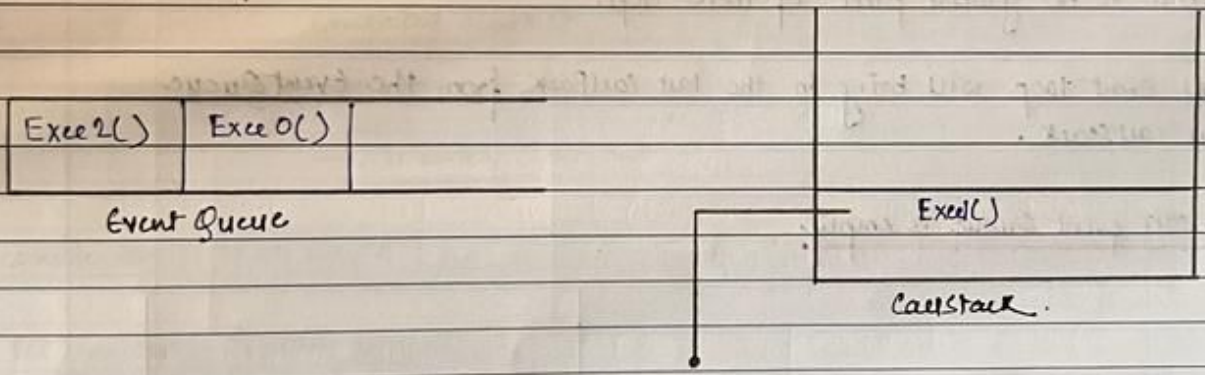
11.00 (1) Is our call stack empty? (YES)

12.00 (2) Are there any global piece of code left? (NO).

01.00

Now, at this point Event Loop will bring the callbacks one by one from the
02.00 Event Queue to the callstack.

03.00



06.00

07.00 After this entry has been made in our callstack.
we will go inside (3B).

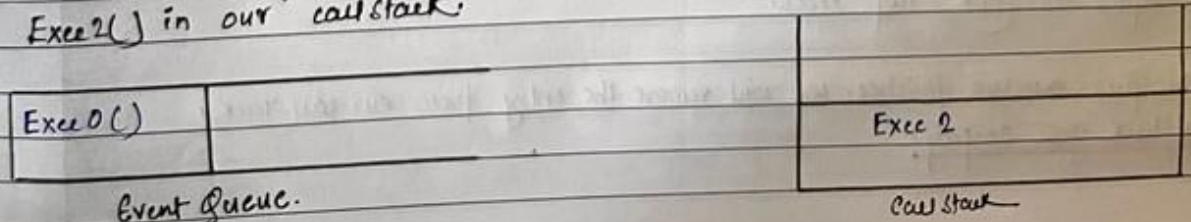
Inside (3B), we have (3C); Completed Time1 gets printed. ✓

Completed Time

After that, the function gets executed. We will remove the entry from call stack.

The callstack gets empty.

∵ Callstack is empty & no global piece of code left. Event loop will bring in
✉ @ ☎ Exec2() in our callstack.



AUGUST 2017

Wk	M	T	W	T	F	S	S
32		1	2	3	4	5	6
33	7	8	9	10	11	12	13
34	14	15	16	17	18	19	20
35	21	22	23	24	25	26	27
36	28	29	30	31			

O/P.

09.00 Entry of exec2() has been made

10.00 We will go inside (4B).

11.00 Inside (4B) we have (4C). Completed Timer 2 gets printed. → Completed Timer 2.

12.00 After this function execution ends. So, we will remove its entry from our callstack.

01.00 Now, our callstack gets empty.

02.00 The event loop at this point found out that our callstack is empty and there is no global piece of code left.

03.00

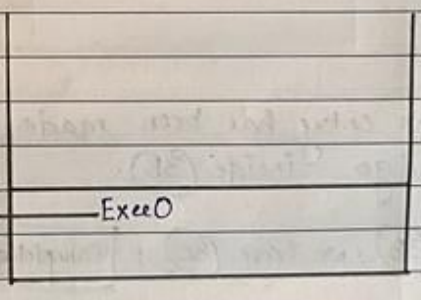
So, ~~can~~ event loop will bring in the last callBack from the EventQueue

04.00 in our callstack.

05.00 Now, our event Queue is empty.

06.00

07.00



Once, this entry has been made we will go inside (2B).

(2C) Completed Timer 0 gets printed. → Completed Timer 0.

(2D). This piece of code is native to (JS). So, (JS) will block here. for (10s). (JS) will not move forward unless and until the (FOR) loop is done executing.

✉ @ ☎ After this (FOR) loop ends.

After this func. execution finishes. We will remove the entry from our callstack.

Callstack gets empty.

SEPTEMBER 2017

Wk	M	T	W	T	F	S	S
36					1	2	3
37	4	5	6	7	8	9	10
38	11	12	13	14	15	16	17
39	18	19	20	21	22	23	24
40	25	26	27	28	29	30	

SATURDAY 2017

34th Week • 231 - 134

AUGUST 19

09.00

∴ Final O|P:—

Hi
Loop Starts.
Loop Ends
Starting Timer
Starting Timer
Starting Timer
Loop Starts.
Loop Ends.
Bye
Completed Timer1
Completed Timer2
Completed Timer0.

—: Interview Questions :—

O|P

SUNDAY 20

(1) console.log("Hello World");

(2) setTimeout(function exec() {
(2A) console.log("Timer done");
}, 0);

(3) console.log("End");

Hello World.

(1). Global piece of code :- Hello World gets printed

(2). This piece of code is NOT Native to (JS). This is a feature provided by the Run-time.

(JS) will notify the (Runtime) to start a timer of 0s. Post which (JS) is going to execute the call Back.

Now, (JS) will come back. In the background a timer of 0s started.

But, the moment (JS) comes back, the runtime has struck (0s).

But, (JS) will not execute (2A) because global piece of code (3) is still left.

AUGUST 2017

Wk	M	T	W	T	F	S	S
32		1	2	3	4	5	6
33	7	8	9	10	11	12	13
34	14	15	16	17	18	19	20
35	21	22	23	24	25	26	27
36	28	29	30	31			

∴ CallBack Exec() is being pushed to the EventQueue.

o/p

09.00

③ Global piece of code :- (End) gets printed.

End.

10.00

Now, Event loop will check the following :-

11.00

① Is the callstack empty? (Yes)

12.00

② Are there any global piece of code left? (No)

01.00

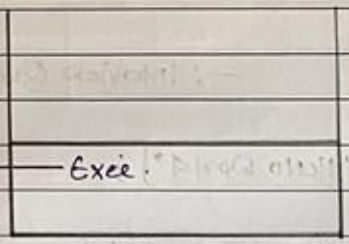
Now, Event loop will pick the callbacks from the EventQueue one by one and push it in our callstack.

By doing so, our EventQueue gets empty.

04.00

05.00

Once, this entry has been made we will go inside ②.



07.00 Inside ②; we have (2A)

(Timer done) gets printed.

Timer done.

Final o/p :-

Hello World
End
Timer done.

✉ @ ☎

SEPTEMBER 2017

W	M	T	W	T	F	S	S
36					1	2	3
37	4	5	6	7	8	9	10
38	11	12	13	14	15	16	17
39	18	19	20	21	22	23	24
40	25	26	27	28	29	30	

TUESDAY 2017

35th Week • 234 - 131

AUGUST 22

#(2). (A) console.log ("Hello World");

09.00

(B) setTimeout (function exec () {

10.00

(i) console.log ("Timer done");

11.00

}, 0);

12.00 (C) for (let i=0; i<10000000000; i++) {

some Task

01.00

}

02.00 (D). console.log ("end")

03.00 (A). Global piece of code :- (Hello World) gets printed

Hello World.

04.00 (B). This piece of code is NOT native to (JS) This feature has been provided by the Run Time.

05.00

(JS) asks Run Time to start a timer of 0s. Post which (JS) is going to execute the callBack.

06.00

07.00 Now, (JS) comes back. And in the background a timer of (0s) has started.

The moment (JS) comes back. RunTime has already struck 0s.

Now,

At this point Event-Loop will check the following :-

⊙ Is the call stack Empty? Yes

⊙ Are there any global piece of code left? Yes.

∴ Yes, Run Time will push the call back Exec() in the Event Queue.

✉ @ ☎ (C). This (FOR) loop is native to (JS). So, (JS) will block here and will not move forward unless and until the (FOR) loop completes.

(D) Global piece of code :- (End) gets printed

(End).

Now,

O/P

At this point Event Loop will check the following :—

① Is the CallStack Empty ? YES

② Are there any global piece of code left ? NO.

∴ The Event Loop will one by one pick up a CallBack at a time from the EventQueue and push it inside the CallStack.

The Event Loop pushes the only CallBack Exec() from EventQueue to CallStack.
∴ EventQueue gets empty.

Once, this entry has been made, we go to inside of (B).

Exec

CallStack.

Inside (B), we have B(i). (Timer done) gets printed.

Timer done

Final O/P :—

Hello World
end
Timer done.

#3

```
① console.log("Hello World");
② for(let i=0; i<3; i++){
    A setTimeout(function exec(){
        ① console.log("Timer done");
    }, 10);
}
③ for(let i=0; i<10000000000; i++){
    # Some Task
}
④. console.log("End");
```

O/P
Hello World.

① Global piece of code :- (Hello World) gets executed.

② This piece of code is NATIVE to (JS).

∴ JS will not move forward unless and until the FOR loop is done executing.

0th iteration :-

(A) This piece of code is NOT Native to (JS). This is a feature being provided by the Run-time.

(JS) will ask Run Time to initiate a timer of 10s. Post which (JS) is going to execute the callBack Exec().

Now, (JS) is going to comeback. In the background, a timer of 10ms. has been started.

1st iteration :-

Same thing is going to happen as that of the 0th iteration.

2nd iteration :-

Same thing is going to happen as that of the 0th & 1st iteration.

After this the (FOR) loop ends and the Event Loop found out that the callStack is empty But there are global pieces of code left.

③. This piece of code is Native to (JS).

∴ (JS) will block here and not move forward unless and until the (FOR) loop is done executing.

In the mid way, all the timers in the Run time must have struck 10s.

But, (JS) is in the middle of executing the FOR loop. But, due to its Synchronous nature, (JS) will not pause. It will keep on running the FOR loop. and all the callBacks inside the Run time will one by one pushed in the Event Queue.

At this point the (FOR) loop is done executing.

O/P.

And, the Event Queue looks something like:—

Exec()	Exec()	Exec()	

Event Queue.

The Event Loop at this point found out there is still global piece of code left.

④. (End) gets printed.

End.

At this point, the Event Loop found out there are no global piece of code left to be executed and also the callstack is empty.
So, Event Loop will one by one pick up callBacks from the Event Queue and push it inside the Call stack.

1st callBack gets pushed inside the callStack:

Exec

CallStack.

Once this entry has been made we will go inside (A)

① gets executed
Timer done gets printed

Timer done.

This is the end of the func. So, we will remove it's entry from callStack.

2nd callBack gets pushed inside the callStack.

✉ @ ☎

→ Same thing happen as in prev. (Timer done) gets printed

Timer done

3rd callBack gets pushed inside the callBack.

→ (Timer done) gets printed

Timer done.

SEPTEMBER 2017

W	M	T	W	T	F	S	S
36					1	2	3
37	4	5	6	7	8	9	10
38	11	12	13	14	15	16	17
39	18	19	20	21	22	23	24
40	25	26	27	28	29	30	

SATURDAY 2017

35th Week • 238 - 127

AUGUST 26

Final O/P :-

```
Hello World
End
Timer Done
Timer Done
Timer Done.
```

All the O/Ps have been justified considering the fact that console.log works synchronously.

-: Promises :-

(1) Readability Enhances :- Increases the overall readability of the code.

(2) They can solve the problem of Inversion of Control.

In JS, promises are special type of objects that get returned immediately when we call them.

Inversion of Control :-

```
function fun (x, cb) {
    for (let i=0; i<x; i++) {
        # some task;
    }
    cb();
}

fun (10, function exec () {
    console.log ("Done");
});
```

Fun() is created by Team A.
I do not know the internal implementation of the Func. fun().

I am in Team B and I am using fun() to pass my callback exec().

∵ I do not know the internal implementation of fun() ⊙ I am unware how my callback is handled.

⊙ like if my callback is called or not
⊙ Whether my callback is called 2/3... times.

∴ IOC states that I am giving the control of my func to another func. whose internal implementation I am unaware of.

SUNDAY 27

Promises also act as a placeholder for the data we hope to get back in the future.

In these promise objects we can attach the functionality we want to execute once the future task is done.

Example :- On the click of a button, it downloads a list of movies.

Lets say, downloading takes 10secs.

Post the downloading, I want to print all of these movies.

This is a deferred task. And I want to do this once I am done downloading my movies.

Mutual Agreement

Maybe we fulfil the promise — Maybe we do not fulfil the promise.

How to create a Promise ?

Promise is NATIVE to JS.

∴ JS will handle Promises in Synchronous Nature form.

① Creation of Promise Object is Synchronous in Nature.

② Three states of a Promise Object (i) Pending (ii) Reject (iii) Fulfilled.

The state can migrate from Pending to → Rejected / Fulfilled.

Pending :- When we create a new Promise Object. This is the default state. "Pending" represents work in progress.

Fulfilled :- If the operation is completed successfully. The state will migrate from Pending to Fulfilled.

09.00 **Rejected :-** If operation was not successful. The state will migrate from Pending to Rejected.

10.00 **Syntax :-**

Constructor :- Constructor is a special function that creates a new object.

11.00

12.00
```
new Promise (function (resolve, reject) {
    Inside this function we can write our time consuming task.
});
```

01.00

02.00

03.00 Resolve and Reject are functions only.

04.00

05.00 **Usecase of Resolve and Reject**

06.00 • The part where we write our time consuming task, Inside this

07.00 ⊙ If we call Resolve() anytime. The state of the promise will go from Pending to ~~Rejected~~ Fulfilled.

⊙ If we call Reject() anytime. The state of the promise will go from Pending to Reject.

⊙ If we do not call anyone, the state of the promise will fore-ever remain in the Pending State.

✉ @ ☎

2017 WEDNESDAY

30 36th Week • 242-123 AUGUST

⊙ With whatever argument we call resolve() or reject() with gets assigned to the value property.

Scenario-01 :-

```
function getRandomInt(max){
    return Math.floor(Math.random() * max);
}

function createPromiseWithLoop(){
    return new Promise(function executor(resolve, reject){
        for(let i=0; i<10000000000; i++){
            # Some Task.
        }
        let num = getRandomInt(10);
        if(num%2==0){
            resolve(num);
        }else{
            reject(num);
        }
    });
}
```

What this func. does is that this func. returns a random value from 0 to max-1.

If the random num is even we fulfil the promise. Else we do not.

In line (A) when we create a Promise Object. Inside the executor function. We have a synchronous code (FOR loop for 10s.
So, after 10s promise returned resolve/rejected. state based on the nature of the Random num.

∴ Promise Object gets returned after 10s.

Note
If we have code after (C). Those won't be executed before 10s.

(A) let x = createPromiseWithLoop();
(C) console.log(x); Promise {<fulfilled>: 8}.

Scenario-02:-

```
function createPromiseWithTimeOut() {
    return new Promise(function executor(resolve, reject) {
        setTimeOut(function() {
            let num = getRandomInt(10);
            if (num%2 == 0) {
                return resolve(num);
            } else {
                return reject(num);
            }
        }, 10000);
    }
}

let x = createPromiseWithTimeOut();
console.log(x);  → Promise {<pending>}.
```

This will immediately return us a Promise. We went inside (B) and inside which we have a feature not Native to JS.
(JS) will notify Runtime to start a timer of 10s. Post which (JS) is going to execute the callback.
Now, we immediately return a Promise Object. and its state is pending.

✉@☎ After 10s is over, if we again do console.log(x)

then we get something like Promise {<rejected> : 5}.

∴ If we have synchronous code inside a Promise, it is going to wait until and unless that synchronous code gets executed.

If we have asynchronous code inside a Promise, it is going to trigger that asynchronous code to your runtime and you immediately get your promise object.

Note

Let's say we write something like,

```
resolve (num, 5, 6, 12);
```

When the promise object gets returned and let's say, the promise gets resolved.

Then

```
Promise {<fulfilled> : 0}
```

first value

"Promise Result" i.e. the value will always take (num). No matter how many parameters you pass after that.

Scenario 03

```
function createPromise() {
    return new Promise ( function executor (resolve, reject) {
        setTimeout ( function () {
            let num = getRandomInt(10);
            if (num % 2 == 0) {
                console.log("fulfilling");
                return num;
            }
            else {
                console.log("Rejecting");
                return num;
            }
        }, 10000)
    }
}
```

```
let x = createPromise();
console.log(x);
```

① We will immediately receive a promise Object.

In the Runtime a timer of 10s started.

② Let's say 10s has been passed and the random number was (5).

We will get o/p like
Rejecting.
Promise {<Pending>}

The state of the promise is still pending because like we discussed unless and until we are calling the function resolve() or reject().

The default state (Pending) will persist throughout and not change.

Scenario-04

Note:-

Any piece of code we write after resolve(x) or reject(x).

That piece of code will also get executed.

Scenario-05

```
let num = getRandomInt(10);

if (num % 2 == 0) {
    console.log("fulfilling");
    resolve(num);
    console.log("completed Resolving");
    resolve(10);
    console.log("Resolving again");
    return num;
}
```


```
else {
    console.log("Rejecting");
    reject(num);
    console.log("Completed Rejecting");
    reject(11);
    console.log("Rejecting again");
    return number;
}
```

P Let's say num = 6;

fulfilling
Completed Resolving
Resolving again.

Promise {<fulfilled> : 6}.

Promise value here is 6 not 10.
Because, once we resolve or reject.
ie Once we change the state of our Promise object it can never be updated.