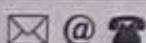


06.00

07.00

- : JS:- Advanced Promises :-

{ promise constructor :- mdn }



OCTOBER 2017

W	M	T	W	F	S	S
40	30	31		1		
41	2	3	4	5	6	7
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28
						29

TUESDAY

2017

37th Week • 248-117

SEPTEMBER

05

09.00 Properties of promise Object :- ① Value :- undefined.

② State :- Pending

new Promise (function exec (resolve, reject) {

});

When we create a new Promise Object, then the whole executor function will be executed.

i.e. once the executor callback is done executing then we get access to the Promise object.

Consuming a Promise :-

When we call a function, that returns us a promise. Inside that promise we can bind some asynchronous piece of logic. That asynchronous piece of logic might be completed in the future and the whole promise will be resolved in future. Till the time the promise is not getting resolved, we get a placeholder object with us.

Using that placeholder object we can do a lot of things. Consumption of it is one such thing.

What should happen if my promise gets fulfilled/rejected, that functionality we can attach (after sometime) → once we get fulfilled/rejected.

Unlike callbacks where we have to mention them and there

→ Eg

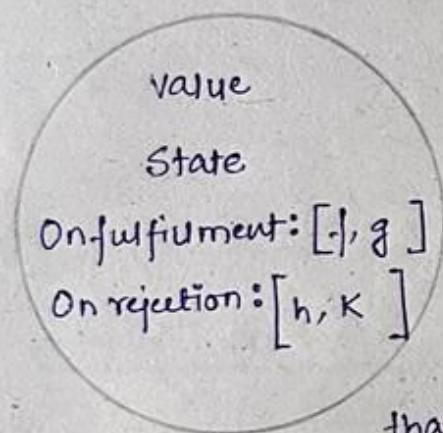
setTimeout (function exec () {
 console.log('Done');

}, 3000);

Over here we are mentioning what needs to be done after 3s.

This is to be studied before the Promise Example:-

Whenever we create a Promise Object, inside that Object we have the following properties:-



$f, g, h, K \rightarrow$ functions.

These funcs are expected to be executed ~~as~~ once the promise object state is either "resolved" or "rejected".

\therefore Onfulfillment array contains the functions that we bind with $\cdot \text{then}$.

When the value parameter is updated from undefined $\rightarrow X$.

Then, X will be served as an input to f, g, h, K .

③ return new Promise (function executor(resolve, reject) { })

017
T
F
S
1 2 3
7 8 9 10
14 15 16 17
21 22 23 24
28 29 30

How can we attach that functionality | Where is the functionality mentioned.

09.00

{ Assuming 'p' as my Promise Object }.

10.00

p.then(fulfilment Handler, rejection handler)

fulfilment handler
rejection handler } These are functions only.
those we have to implement ourselves.

11.00

With .then functionality we are registering a
fulfilment Handler:- What should happen if my promise object is fulfilled.

like what piece of code needs to be created if our promise object state is fulfilled or rejected.

01.00

rejection Handler:- What should happen if my promise object is rejected.

03.00 Example:-

```

① 04.00 function getRandomInt (max) {
    |
    05.00   return Math.floor (Math.random () * max);
    |
    06.00 }

② 07.00 function createPromiseWithTimeout () {
    |
    ③ 08.00 return new Promise (function executor (resolve, reject) {
        |
        ④ 09.00 settimeout (function () {
            |
            ⑤ 10.00 let num = getRandomInt (10);
            |
            ⑥ 11.00 if (num % 2 == 0) {
                |
                12.00   resolve (num);
                |
                ⑦ 13.00 } else {
                    |
                    14.00   reject (num);
                    |
                    15.00 });
            |
            16.00 });
        |
        17.00 });
    |
    18.00 });
  
```

OCTOBER 2017

SUN	MON	TUE	WED	THU	FRI	SAT
30	31	1				
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29						

THURSDAY

2017

37th Week • 250-115

SEPTEMBER

07

```

⑤ console.log("Starting");
09.00

⑥ const p = createPromiseWithTimeOut();
10.00

⑦ console.log("Currently Promise Object looks like", p);
11.00

⑧ console.log("We are waiting for promise to complete");
11.00

⑨ p.then(function fulfillHandler(v) {
12.00
    ⑩ console.log("Inside fulfill Handler with value", v);
01.00
    })
    , function rejectHandler(v) {
02.00
        ⑪ console.log("Inside Reject Handler", v);
03.00
    });
04.00

```

O/P:-

Starting

Currently Promise Object looks like, Promise{<pending>}

We are waiting for promise to complete.

Inside fulfill Handler with value 8.

⑤ Global piece of code:- Starting gets printed

⑥ 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.

✉ @ createPromiseWithTimeOut.

CallStack.

2017

FRIDAY

08

37th Week • 251-114

SEPTEMBER

SEPTEMBER 2017						
Wk	M	T	W	T	F	S
36					1	2
37	4	5	6	7	8	9
38	11	12	13	14	15	16
39	18	19	20	21	22	23
40	25	26	27	28	29	30

② Inside ②, we have ③.

line ③ states that to return a new Promise.

So, in our memory, a new Promise object will be created.

pending

undefined

onfulfillment : []

onreject : []

Promise Object.

During creation of the promise object, we will start calling the Executor function.

Inside the Executor func. we have ④.

④ This piece of code is NOT Native to JS. This feature is being provided to JS by Runtime.

∴ JS will notify Runtime to start a timer of 10s. Post which JS is going to execute the anonymous callback.

In the background, a timer of 10s has started. We are going to exit the executor func.

Line ③ is done executing, we will return the promise object to line ⑥.

∴ p =

pending

undefined

[]

∴ line ② is done executing, the entry corresponding to the func gets removed.

callstack is now empty.

OCTOBER 2017						
S	M	T	W	T	F	S
30	31	1	2	3	4	5
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29		

SATURDAY 2017
37th Week • 252-113
SEPTEMBER 09
10 P.M.

Q. Global piece of code:-

Currently promise object looks like

pending
undefined.

∴ We will print the promise in Pending State.

Q. Global piece of code:-

We are waiting for Promise to complete "get printed".

Q. We have •then operator.

Q. What p.then does is, it registers handlers ⁱⁿ for our promise object.

i.e. "fulfillHandler" gets registered inside our onfulfillment array.

"rejectHandler" gets registered inside our onrejection array.

Now, our promise object will look something like:-

value: ~~and~~ undefined

state: pending.

Onfulfillment: [fulfillHandler]

Onrejection: [rejectHandler]

Note:-

Registering a function ≠

Executing a function.

Note:-

Why onfulfillment & onrejection are arrays?

Because we can register multiple handlers inside them.

We will register our func. and not execute them at this time.

2017 MONDAY

SEPTEMBER 2017						
W	M	T	W	T	F	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	

38th Week • 254-111

SEPTEMBER

After line ⑨, we have no more code left to be executed. Meanwhile, the timer has eventually come to a stop.

10.00 The Runtime will push a callback to our Event Queue.

11.00 The Event Loop will check that our call stack is empty and there is no global piece of code left to be executed.

12.00 So, Event Loop will one by one push one callback at a time from the Event Queue to our call stack.

02.00

03.00

04.00 Once, this entry has been made we would go inside.

④.

05.00

CallStack.

④ 06.00 Inside ④, we have ⑤.

⑤ 07.00 We generate a random number (let say 6)

⑥ Inside ⑥, we resolve the promise.

∴ The promise object from ② Pending goes to fulfilled state.

③ undefined goes to 6.

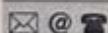
state: fulfilled

value: 6

on fulfillment: [fulfill Handler]

on rejection: [rejection Handler]

promise
object:



OCTOBER 2017						
S	M	T	W	T	F	S
					1	
40	30	31	1	2	3	4
41	1	2	3	4	5	6
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28
					29	

TUESDAY

2017

38th Week • 255 - 110

SEPTEMBER

12

After ⑥ no more code left to execute for the callback.

∴ We remove the entry corresponding to that callback from our call stack.
∴ callstack gets empty.

NOW, Since the state of the promise object is now Fulfilled.

∴ All the handlers inside onfulfillment array will now start their execution.



2017 WEDNESDAY

13

38th Week • 256-109

SEPTEMBER

-: Microtask Queue :-

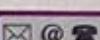
SEPTEMBER 2017						
W	M	T	W	T	F	S
36					1	2
37	4	5	6	7	8	9
38	11	12	13	14	15	16
39	18	19	20	21	22	23
40	25	26	27	28	29	30

```
① function createPromise () {  
09.00  
    ② return new Promise (function executor (resolve, reject) {  
10.00  
        ③ console.log ("Resolving the promise");  
11.00  
        ④ resolve ("Done");  
12.00  
    } );  
01.00  
⑤ setTimeout (function process () {  
02.00  
    ⑥ console.log ("Timer completed");  
03.00  
}, 0);  
04.00  
⑦ let p = createPromise ();  
05.00  
⑧ p.then (function fulfillHandler (value) {  
06.00  
    ⑨ console.log ("We fulfilled with a value", value);  
07.00  
},  
    function rejectHandler (value) {  
        ⑩ console.log ("We fulfilled with a value", value);  
    } );  
11. . console.log ("Ending");
```

Note:-

When your promise is resolved or rejected.
The Handlers present inside onfulfillment and on rejection arrays
They are not immediately executed.

Once the state of the Promise changes to:-



① Fulfilled:- All the Handlers inside onfulfillment array

gets pushed to the microtask queue. (one by one)

And they wait for Event Loop to show them a signal. i.e. Event Loop will check if the callstack is empty and there are no global piece of code left.

OCTOBER 2017						
SUN	MON	TUE	WED	THU	FRI	SAT
30	31	1	2	3	4	5
31	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12
7	8	9	10	11	12	13
8	9	10	11	12	13	14
9	10	11	12	13	14	15
10	11	12	13	14	15	16
11	12	13	14	15	16	17
12	13	14	15	16	17	18
13	14	15	16	17	18	19
14	15	16	17	18	19	20
15	16	17	18	19	20	21
16	17	18	19	20	21	22
17	18	19	20	21	22	23
18	19	20	21	22	23	24
19	20	21	22	23	24	25
20	21	22	23	24	25	26
21	22	23	24	25	26	27
22	23	24	25	26	27	28
23	24	25	26	27	28	29

⑤ NOT Native to JS.

JS notifies Runtime to start a timer of 0s.

THURSDAY

2017

Post which Runtime is going to push the callback in Event queue and JS will pick that up & execute it.

38th Week • 257 - 108

SEPTEMBER

14

- ⑦ Function call:- We will make an entry corresponding to the func. createPromise in our callstack.

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

11:00

12:00

01:00

02:00

03:00

04:00

05:00

06:00

07:00

08:00

09:00

10:00

2017

FRIDAY

15

38th Week • 258 - 107

SEPTEMBER

SEPTEMBER 2017

W	M	T	W	T	F	S	S
36						1	2
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	

Nothing more to execute inside ④.

09.00

Now, we will return the promise object.

10.00

Nothing more to execute inside ①. ∴ We will remove its entry from the callstack.

11.00

Callstack gets empty.

12.00

③ We have .then function.

01.00

• then function registers the fulfillHandler and rejectHandler inside the Promise object.

02.00

∴ After registering our promise object looks like.

03.00

value: "Done"

state: fulfilled

The moment we do this registration all of these handlers are going in the microtask queue one by one.

onfulfillment: [fulfillHandler] promise object?

04.00

onreject: [rejectHandler]

05.00

⑩ "Ending" gets printed.

After this Event Loop found out that callstack is empty and there is no global piece of code left.

We have a callback inside EventQueue

of

Handlers inside Microtask Queue.

Priority will be given to Microtask Queue.

OCTOBER 2017

SUN	MON	TUE	WED	THU	FRI	SAT
30	31	1				
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29						

SATURDAY

2017

38th Week • 259 - 106

SEPTEMBER

16

Event loop will now have to make a choice b/w

① Executing the callback from Event Queue.

② Executing the Handler from Microtask Queue.

Event Loop will prioritize the microtask queue.

fulfillHandler() goes from microtask queue to the callback.

Microtask queue gets empty.

Once this entry has been made we will go inside this handler. → Fulfill Handler

callstack.

④ "We fulfilled with a value Done" gets printed.

Nothing more to execute, we will remove its entry from the callstack. If callstack gets empty.

Event Loop will now push the callback from Event Queue to callstack. Event queue gets empty.

Once, this entry has been made we will go inside the callback. → process.

callstack.

⑥ "Timer Completed" gets printed.

Nothing more to execute, we will remove its entry from callstack. If callstack gets empty.

2017

MONDAY

18

39th Week • 261-104

SEPTEMBER

SEPTEMBER 2017						
W	M	T	W	T	F	S
36						1
37	4	5	6	7	8	2
38	11	12	13	14	15	3
39	18	19	20	21	22	10
40	25	26	27	28	29	17

-: Key Pointers :-

09.00

① IF the promise object state is "Pending".

We will register both the handlers in our Promise Object.

But we will not push any of the handler in the Microtask Queue.

01.00

② IF the promise object state is "Fulfilled" / "Rejected"

02.00

IF it is "fulfilled" we will register only the fulfill Handler in our Promise Object.

03.00

we will push fulfillHandler in the Microtask queue.

04.00

And the other way around if it is "rejected".

05.00

06.00

07.00



CAMBRIDGE

OCTOBER 2017

SUN	M	T	W	T	F	SAT
30	31	1	2	3	4	5
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29		

TUESDAY 2017

39th Week • 262-103

SEPTEMBER

19

① function createPromise () {
 ② return new Promise (function executor (resolve, reject) {
 ③ console.log ("Resolving the promise");
 ④ resolve ("Done");
 ⑤ });
 ⑥ set Timeout (function, process () {
 ⑦ console.log ("Timer Completed");
 ⑧ }, 0);
 ⑨ }
 ⑩ let p = createPromise ();

⑪ p. then (function fulfillHandler1 (value) {
 ⑫ console.log ("we fulfilled 1 with a value", value);
 ⑬ function rejectHandler1 (value) {}
 ⑭ })
 ⑮ p. then (function fulfillHandler2 (value) {
 ⑯ console.log ("we fulfilled 2 with value", value);
 ⑰ function rejectHandler2 (value) {}
 ⑱ })
 ⑲ for (let i=0; i<10000000000; i++) {}
 ⑳ console.log ("Ending") ;

2017

WEDNESDAY

20

39th Week • 263-102

SEPTEMBER

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	

- ⑤ This is not Native to JS. This feature is being provided by the Run-time.

09.00

JS notifies Runtime to start a timer of OS. Post which Run Time will push the callback inside Eventqueue & from there JS is going to take care.

11.00 ∵ In the background a timer of OS has started.

- ⑥ Function call:- By the time we reach this step, the runtime timer has struck OS.

01.00 So runtime will push the callback "process" inside the Event Queue.

02.00 Now, since this is a function call we will make this entry in our call stack.

03.00

04.00

Once this entry has been made we will go inside ①. —————— process. Create Promise

05.00

CallStack.

06.00

- ① Inside ①, we have return Promise object statement.

07.00

∴ In our memory a new Promise object gets created.

value: undefined
state: pending
onfulfilment: []
onrejection: []

During the creation of the Promise object itself, the executor func. gets called.

promise object.

OCTOBER 2017

Wk	M	T	W	T	F	S	S
40	30	31			1		
41	2	3	4	5	6	7	8
42	9	10	11	12	13	14	15
43	16	17	18	19	20	21	22
44	23	24	25	26	27	28	29

THURSDAY 2017

39th Week • 264-101
SEPTEMBER

21

Inside the executor function,

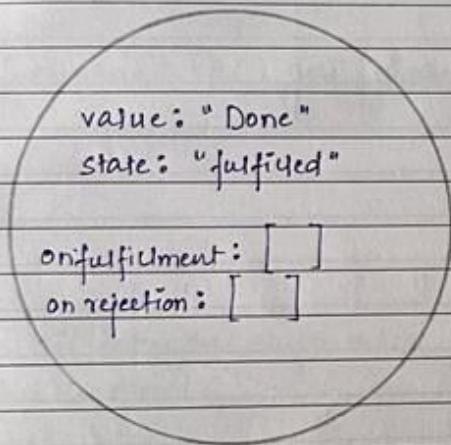
③ "Resolving the Promise" gets printed.

④ We now resolve the promise.

By doing so, our promise object state goes from pending to resolved.

Value goes from undefined to "value".

Our promise object looks like :-



Nothing more to execute inside executor function.

Inside ② nothing more to execute
So, we will return this promise
object to line ⑦.

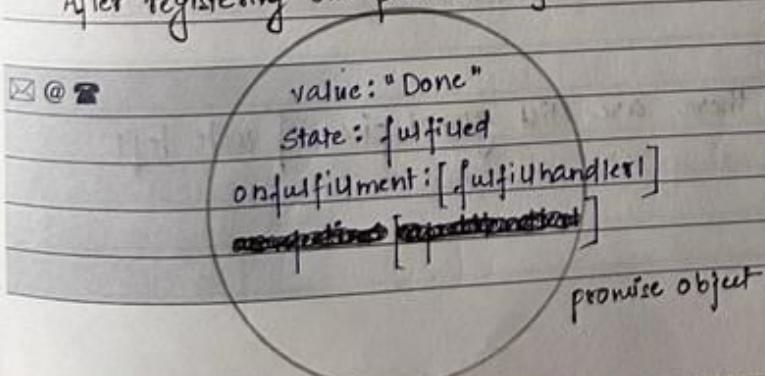
Nothing more to execute inside ①.

∴ We will remove this entry from our callstack.
So, our callstack gets empty.

⑧ We have .then operator.

.then registers the handlers in our Promise Object.

After registering our promise object looks something like:-



Since, the state is fulfilled.

We will push the Handlers inside onfulfillment in our microtask queue.

2017 FRIDAY

22

39th Week • 265-100

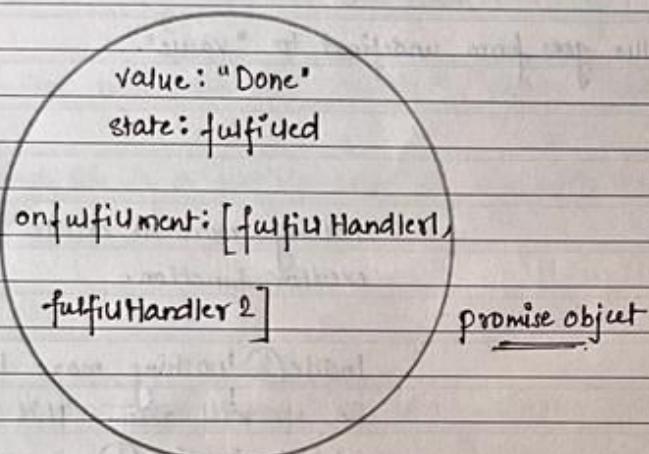
SEPTEMBER

SEPTEMBER 2017						
W	M	T	W	T	F	S
36						
37	4	5	6	7	8	9
38	11	12	13	14	15	16
39	18	19	20	21	22	23
40	25	26	27	28	29	30

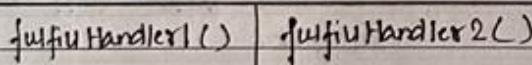
⑩. We have .then operator

• then operator registers the handlers in our promise object.

Here, since the state of the promise object is "fulfilled". We will register fulfillHandler2() inside our promise object.

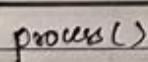


And our microtask queue looks something like:-

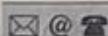


Microtask Queue.

Our Event queue looks something like:-



Event Queue.



Event loop at this point found out that there are still global piece of code left to be executed.

OCTOBER 2017

SUN	M	T	W	T	F	SAT
40	30	31			1	
41	2	3	4	5	6	7
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28
					29	

SATURDAY

2017

39th Week • 266-099

SEPTEMBER

23

- (12) This piece of code is Native to JS.

So, JS will block here and not move forward, unless and until the FOR loop is done with its execution.

- 11:00 (13) "Ending" gets printed.

12:00 At this point Event loop found out that callstack is empty and there are no global piece of code left to be executed.

01:00 So, priority to execute will be given to Microtask queue.

03:00 ∴ fulfillHandler1() gets pushed inside the callstack.

04:00

05:00

06:00 Once this entry has been made, we will go inside fulfillHandler1().

07:00 (9) "We fulfilled with value Done" gets printed.

SUNDAY 24

Nothing more to execute. We will remove its entry from our callstack.

∴ callstack gets empty.

Event Loop will now push fulfillHandler2() in our callstack.
of our microtask queue gets empty.

✉ @ 📲

Once this entry has been made we will go inside fulfillHandler2().

callstack.

2017

MONDAY

25

40th Week • 268-097

SEPTEMBER

SEPTEMBER 2017						
W	M	T	W	T	F	S
			36		1	2
			37	4	5	3
			38	11	12	10
			39	18	19	17
			40	25	26	24

- (1) "We fulfilled2 with value Done" gets printed.

09.00

Nothing more to execute. So entry corresponding to this func. gets removed.

10.00

callstack gets empty.

11.00

Now, event loop will one by one push callback from event queue to callstack.

12.00

Now, event queue gets empty.

01.00

02.00

03.00

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

process

04.00

callstack.

05.00

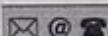
(2) "Timer Completed" gets printed.

06.00

Nothing more to execute. So entry corresponding to this func. gets removed.

07.00

Callstack gets empty.



OCTOBER 2017

SUN	MON	TUE	WED	THU	FRI	SAT
40	30	31			1	
41	2	3	4	5	6	7
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28
	29					

TUESDAY

2017

40th Week • 269-096

SEPTEMBER

26

```

① function createPromise() {
  ②   return new Promise(function executor(resolve, reject) {
    ③     setTimeout(function () {
      ④       console.log("Rejecting the promise");
      ⑤       reject("Done");
    }, 1000);
  });
  ⑥   let p = createPromise();
  ⑦   p.then(function fulfillHandler1(v) {
    ⑧     console.log("We fulfilled1 with value", v);
  }, function rejectHandler1(v) {
    ⑨     console.log("We rejected1 with value", v);
  });
  ⑩   p.then(function fulfillHandler2(v) {
    ⑪     console.log("We fulfilled2 with value", v);
  }, function rejectHandler2(v) {
    ⑫     console.log("We rejected2 with value", v);
  });
  ⑬   for (let i=0; i<10000000000; i++) {
  ⑭   console.log("Ending");
}

```

2017

WEDNESDAY

27

40th Week • 270-095

SEPTEMBER

SEPTEMBER 2017

Wk	M	T	W	T	F	S	S
36						1	2
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	

- ⑥ Function call :- We will make an entry corresponding to this func in our callstack.

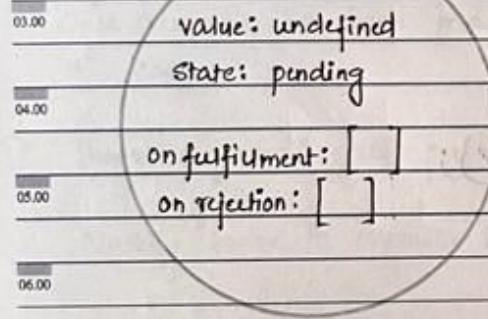
10.00

11.00 Once this entry has been made, we will go inside ①. → create Promise

12.00 Inside ①; we have ②. callstack.

- 01.00 It says to return a new Promise object. So, in our memory a promise object gets created.

02.00 During the creation of the promise object we will call the executor func.



07.00 Inside the executor func. we have ③.

- 07.00 This piece of code is NOT Native to JS. This feature is being provided to JS by Runtime.

JS will notify Runtime to start a timer of 1s. Post which Runtime will push the callback inside EventQueue and from there JS will take care of the rest.

JS comes back and in the background a timer of 1s starts.

- ③ is done and so is the executor func.

Once the executor func. is done executing, we will return a promise object to ⑥.

① is done executing. We will remove its entry from callstack.
∴ callstack gets empty.

OCTOBER 2017

W	M	T	W	F	S	S
40	30	31		1		
41	2	3	4	5	6	7
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28
						29

THURSDAY

2017

40th Week • 271-094

SEPTEMBER

28

①. We encounter .then function.

09.00

• then function registers handlers in our promise object.

10.00

∴ the current state is "Pending". We will register both the handlers in our promise object.

12.00

value: undefined

state: pending

onfulfillment: [fulfillhandler1]

onrejection: [rejectionhandler1]

Also, since the state is pending will not push any of the handlers inside the microtask queue.

promise object.

01.00

02.00

03.00

04.00

②. We encounter .then function.

05.00

• then func. registers handlers in our promise object.

06.00

Same as above point will happen.

07.00

value: undefined

state: pending

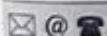
onfulfillment: [fulfillmenthandler1],

fulfillmenthandler2]

promise object.

onrejection: [rejectionhandler1],

rejectionhandler2]



2017

FRIDAY

29

40th Week • 272-093

SEPTEMBER

SEPTEMBER 2017

W	M	T	W	T	F	S	S
36						1	2
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	

- (13) This piece of code is Native to JS.

09.00

JS will block here and not move forward. Unless and until FOR loop is done executing.

10.00

While the FOR loop is executing, the Runtime has struck its.

11.00

So the callback will be pushed inside the Event Queue.

12.00

Event Queue will look something like:-

02.00

f()

03.00

Event Queue.

- (14) "Ending" gets printed.

05.00

At this point event loop encountered the callstack is empty.

No global piece of code left.

Microtask queue is empty.

06.00

∴ Event Loop will one by one pickup a callback at a time from the Event queue and push inside the callstack.

By doing so, event queue gets empty.

Once this entry has been made we will go inside the

f()

✉ @ ☎

- (3) Inside (3), we have (4).

"Rejecting the Promise" gets printed.

callstack

OCTOBER 2017

W	M	T	W	T	F	S	S
40	30	31	1	2	3	4	5
41	2	3	4	5	6	7	8
42	9	10	11	12	13	14	15
43	16	17	18	19	20	21	22
44	23	24	25	26	27	28	29

SATURDAY

2017

40th Week • 273-092

SEPTEMBER

30

- ⑤ We are ~~resolving~~ rejecting the promise with value "Done".

∴ the promise object will look something like

value: "Done"
state: rejected.

∴ The state is "rejected". We will push handlers inside on rejection ~~and~~ into the microtask queue.

on fulfillment: [fulfillmentHandler1,
fulfillmentHandler2].

on rejection: [rejectionHandler1,
rejectionHandler2].

Microtask queue will look something like:-

Rejection Handler1() Rejection Handler2()

promise object.

microtask queue.

Event Loop will one by one push handlers from microtask queue to callstack.

06.00

07.00

Once, this entry has been made we will go inside Rejection Handler1().

Rejection Handler1()

SUNDAY 01

callstack.

- ① "We rejected with value Done" gets printed.

Nothing else to execute. We will remove the entry from callstack.
callstack gets empty.

Event loop will push the next handler inside callstack.

Rejection Handler2()

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

callstack.

2017 MONDAY

OCTOBER 2017						
SUN	MON	TUE	WED	THU	FRI	SAT
	40	30	31			
	41	2	3	4	5	6
	42	9	10	11	12	13
	43	16	17	18	19	20
	44	23	24	25	26	27
						28
						29

02

41st Week • 275-090

OCTOBER

⑫. "We rejected 2 with value Done" gets printed.

09.00

Nothing more to execute, we will remove iIT entry from callstack.
Callstack gets empty.

11.00

① function fetchData(url) {

12.00

② return new Promise(function (resolve, reject) {

01.00

③ console.log("Started downloading from", url);

02.00

④ setTimeout(function processDownloading() {

03.00

⑤ let data = "Dummy data";

04.00

⑥ console.log("Download complete");

05.00

⑦ resolve(data);

06.00

}); }, 7000);

07.00

});

⑧ console.log("Start");

⑨ let promiseObj = fetchData("abcdef");

⑩ promiseObj.then(function A(value) {

⑪ console.log("Value is", value);

});

⑫ console.log("End");

NOVEMBER 2017

SUN	M	T	W	T	F	S	SAT
45	1	2	3	4	5		
46	6	7	8	9	10	11	12
47	13	14	15	16	17	18	19
48	20	21	22	23	24	25	26
49	27	28	29	30			

TUESDAY

2017

41st Week • 276-089

OCTOBER

03

⑧ Global piece of code.

09.00

"Start" gets printed.

10.00

⑨. Function call:- Since this is a func. call, we will create an entry for this func. in our callstack.

12.00

01.00

Once this entry has been made, we will go inside ①.

02.00

Inside ①, we have ②.

03.00

② It says to return a new promise. So, in the memory a new memory object is created.

04.00

value: undefined
state: pending.
on fulfillment: []
on rejection: []

promise object.

During the creation of the promise object, we will call the executor function.

③. "Started downloading from abcdef" gets printed.

④. This piece of code is not Native to JS. This is a feature which is being provided to JS by RunTime.

JS will notify the RunTime to start a timer of 7s. Post which RunTime will push the callback in the EventQueue and JS will take care of the rest.

✉ @

JS comes back and in the background a timer of 7s has started.

2017 WEDNESDAY

04

41st Week • 277-088

OCTOBER

OCTOBER 2017						
W	M	T	W	T	F	S
	40	30	31			
	41	2	3	4	5	6
	42	9	10	11	12	13
	43	16	17	18	19	20
	44	23	24	25	26	27

Nothing more to execute.
09.00

We are done executing the executor function.
10.00

We will return a promise object to line ⑨.
11.00

Nothing more to execute inside ①. So, we will remove 1G entry from the collector.
12.00

Callstack is now empty.
01.00

⑩. We have .then operation function.
02.00

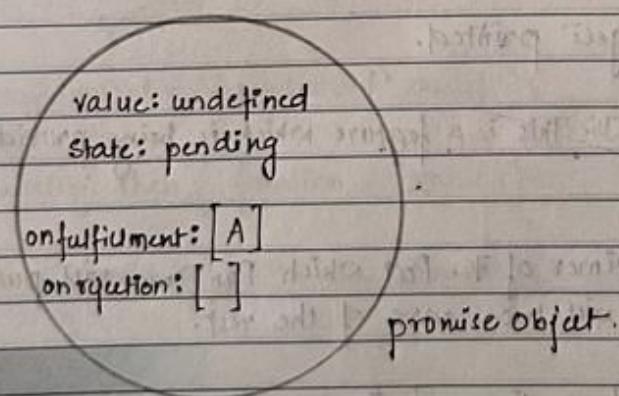
.then function registers handlers inside the Promise object.
03.00

∴ The state is "pending". We will register all the handlers.

Also, we will not push any handler inside microtask queue.
04.00

inpt
05.00: Note:- Since we are using only 1 handler inside .then
06.00 It is actually a fulfill Handler.

07.00 ∴ Our promise object will look like:-



⑪. "End" gets printed.

NOVEMBER 2017

S	M	T	W	T	F	S	S
45	1	2	3	4	5		
46	6	7	8	9	10	11	12
47	13	14	15	16	17	18	19
48	20	21	22	23	24	25	26
49	27	28	29	30			

THURSDAY

2017

41st Week • 278-087

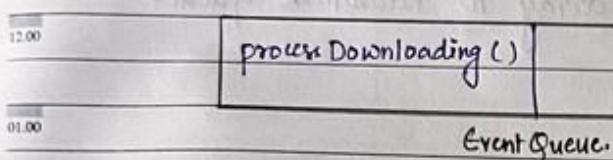
OCTOBER

05

At this point the runtime must have struck 7s.

∴ Runtime will push the callback - `processDownloading` in our Event Queue.

∴ Event Queue will look something like:-



02.00 At this point Event Loop checks that ① CallStack is empty.

- ② No global piece of code left.
- ③ microtask queue is empty.

04.00 So, event loop will push the callback from Event queue into callStack.

05.00 ∴ Event Queue gets empty.

07.00 Once this entry has been made, we will go inside the `processDownloading()` callback.

④ Variable "data" gets value "Dummy data".

⑤ "Download complete" gets printed.

⑥ We are resolving the promise. Our promise object will look like:-



value: "Dummy data"

state: fulfilled.

promise object.

onfulfillment: [A]

onrejection: [] .

2017 FRIDAY

06

41st Week • 279-086

OCTOBER

OCTOBER 2017

SUN	MON	TUE	WED	THU	FRI	SAT
40	30	31				
41	2	3	4	5	6	7
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28

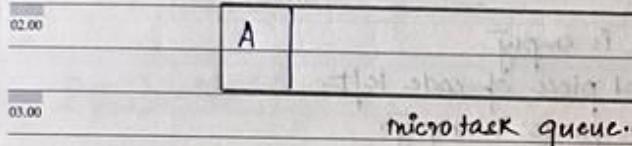
Nothing more to execute inside ⑨. We will remove the entry from the callstack.

09.00 Callstack gets empty.

10.00 Now, since the state of the promise gets changed from pending to fulfilled.

11.00 We will move the handlers from inside onfulfillment array to microtask queue.

12.00 The microtask queue will look something like:-



Now, Event Loop found that ⑩ Callstack is empty.

⑪ no global piece of code left.

⑫ But there are handlers inside microtask queue.

06.00 Event Loop will pick one handler at a time from microtask queue and push into callstack.

By doing so microtask queue gets empty.

Once this entry has been made we will go inside handler A.

A

callstack.

⑪ "Value is Dummy data" gets printed.

✉ @ 📞

Nothing more to execute. We will remove its entry from callstack.

Callstack gets empty.

NOVEMBER 2017

W	M	T	W	F	S	S
45		1	2	3	4	5
46	6	7	8	9	10	11
47	13	14	15	16	17	18
48	20	21	22	23	24	25
49	27	28	29	30		

SATURDAY

2017

41st Week • 280-085

OCTOBER

07

```

① function fetchData(url) {
    ② return new Promise(function (resolve, reject) {
        ③ console.log("Started downloading from", url);
        ④ setTimeout(function processDownloading() {
            ⑤ let data = "dummy data";
            ⑥ resolve(data);
            ⑦ console.log("Download completed");
        }, 7000);
    });
    ⑧ console.log("Start");
    ⑨ let promiseObj = fetchData("abcde");
    ⑩ promiseObj.then(function A(value) {
        ⑪ console.log("Value is", value);
    });
    ⑫ console.log("End");
}

```

SUNDAY



2017 MONDAY

09

42nd Week • 282-083

OCTOBER

OCTOBER 2017						
Mo	Tu	We	Th	Fr	Sa	Su
40	30	31				
41	2	3	4	5		
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28

⑧ "Start" gets printed.

⑨ Function call:- We will create an entry in our callstack corresponding to this func.

Once this entry has been made we will go inside ①.

⑩ A new promise object gets created in the memory.

value: undefined

state: pending.

onfulfilment: []

onrejection: []

During creation of the promise object we will be calling the executor func.

⑪ "Started downloading from abedc" gets printed.

⑫ This piece of code is not Native to JS. This feature is provided to JS by RunTime. JS notifies RunTime to start a timer of 7s, past which RunTime is going to push the callback in our Event Queue.

JS comes back, in the background a timer of 7s has started off.

No more code to execute inside the executor func. so we return promise object to line ⑨.

Nothing more to execute inside ①, the entry in the callstack gets removed.

✉ @ Callstack gets empty.

NOVEMBER 2017

SUN	M	T	W	T	F	SAT
45	1	2	3	4	5	
46	6	7	8	9	10	11
47	13	14	15	16	17	18
48	20	21	22	23	24	25
49	27	28	29	30		

TUESDAY

2017

42nd Week • 283-082

OCTOBER

10

- ⑩. We encounter .then operator.

09.00

.then operator registers handlers inside our promise object.

10.00

∴ There is only one handler inside .then. It is a fulfillment handler.

11.00

So, our promise object looks like :-

12.00

value: undefined

state: pending

onfulfilment: [A]

onrejection: []

promise object.

04.00

- ⑪. "End" gets printed.

06.00

At this point of time, its timer has struck and Runtime has pushed the callback process Downloading in the event queue.

07.00

Event Loop did the following check ① CallStack is empty

② No global piece of code left.

③ Microtask queue is empty.

So, Event loop will push the callback from Event queue to callstack.

Once this entry has been made we will go inside ④. → process Downloading()

callstack

2017 WEDNESDAY

11

42nd Week • 284-081

OCTOBER

OCTOBER 2017						
SUN	MON	TUE	WED	THU	FRI	SAT
40	30	31				
41	2	3	4	5	6	7
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28

- ⑤ Variable data gets assigned a value "dummy data" inside our callstack.
09.00
- ⑥ We are resolving / fulfilling our promise.
10.00
- ∴ The promise object will look like:-
11.00

12.00

value: "dummy data"

state: fulfilled.

onfulfillment: [A]

onrejection: []

promise object.

This step is done when the promise state changes to fulfilled.

- ⑦ 04.00 Download complete" gets printed.

05.00 Nothing else to execute, so the entry corresponding to this func gets removed and our callstack gets empty.
06.00

07.00 The state of the promise object is now "fulfilled". We will send all the handlers inside onfulfillment array inside our microtask queue one by one.

A

microtask queue.

Event Loop will check the following :- ① Callstack is empty.

② No global piece of code left.

③ Handler inside microtask queue.

✉ @ ☎

Event Loop will push the handler from microtaskqueue to callstack.

NOVEMBER 2017

W	M	T	W	F	S	S
45		1	2	3	4	5
46	6	7	8	9	10	11
47	13	14	15	16	17	18
48	20	21	22	23	24	25
49	27	28	29	30		

THURSDAY

2017

42nd Week • 285-080

OCTOBER

12

09.00

Once this entry has been made we will go inside ⑩.

A()

11.00

(11) "Value is dummy data" gets printed.

CallStack

12.00

Nothing more to execute. Entry corresponding to this handler gets removed from callstack.
01.00 Callstack gets empty

02.00

Order of priority :-

03.00

Callstack of global piece of code > Microtask queue > Event queue.

04.00

```
function f() {
```

05.00

```
    return new Promise(function executor(resolve, reject) {
```

06.00

```
        resolve("Subhadip Das");
```

07.00

```
    });
```

08.00

```
let x = f();
```

This already resolved Promise can also be written as,

```
let x = Promise.resolve("Subhadip Das");
```

✉ @ ☎

2017 FRIDAY

13

42nd Week • 286-079

OCTOBER

OCTOBER 2017						
W	M	T	W	T	F	S
40	30	31				
41	2	3	4	5	6	7
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28

① console.log ("Start of the file");

② setTimeout (function timer1 () {

 ③ console.log ("Timer 1 done");
 11.00 }, 0);

④ for (let i=0 ; i<10000000000; i++) {

 12.00
 }

⑤ let x = Promise.resolve ("Subhadip");

⑥ x.then (function processPromise (value) {

 ⑦ console.log ("Whose promise?", value);
 05.00 });

⑧ setTimeout (function timer2 () {

 ⑨ console.log ("Timer 2 done");
 07.00 }, 0);

⑩. console.log ("End of file");

① Global piece of code:- "Start of the file" gets printed.

② This piece of code is not Native to JS. This is a feature which is being provided to JS by RunTime.

@ JS notify RunTime to start a timer of 0s part of which RunTime will push the callback in our event queue and JS will take care of the rest.

JS comes back and it in the background a timer of 0s has started.

NOVEMBER 2017

W	M	T	W	T	F	S	S
45		1	2	3	4	5	
46	6	7	8	9	10	11	12
47	13	14	15	16	17	18	19
48	20	21	22	23	24	25	26
49	27	28	29	30			

SATURDAY 2017

42nd Week • 287-078
OCTOBER

14

- (4). Even before starting this synchronous piece of code.. RunTime has struck OS.
RunTime will push the callback timer1 in our Event queue.

Event queue will look something like:-

11.00 | timer1() |

12.00 | Event Queue.

01.00 ∵ FOR loop is a blocking piece of code. JS due to its synchronous nature will not move forward. unless and until a time for 10s.

- (5). We are returning an already resolved promise with value "Subhadip".

∴ X =
value: "Subhadip"
state: fulfilled
onfulfillment: []
onrejection: []

promise object.

- (6). We encounter a ".then" function.

.then func. registers handlers in our promise object.

SUNDAY 15

∴ There is only one handler inside .then. It must be a fulfill handler.

∴ Promise object will look like.

✉ @ ☎
value: "Subhadip"
state: fulfilled.
onfulfillment: [processPromise]
onrejection: []
promise object.

2017 MONDAY

16

43rd Week • 289-076

OCTOBER

OCTOBER 2017						
MON	TUE	WED	THU	FRI	SAT	SUN
40	30	31				
41	2	3	4	5	6	7
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28

Once, the registration is done.

09.00

∴ the state of the promise object is "fulfilled".

10.00

JS will push all the handlers inside the onfulfillment array inside the microtask queue.

12.00

processPromise()

01.00

microtask queue.

02.00

- ⑧. This piece of code is NOT native to JS. This feature is being provided to JS by Runtime.

JS will notify Runtime to start a timer of 0s post which Runtime is going to push the callback timer2() inside Event queue.

05.00

JS comes back and in the background a timer of 0s has started.

- ⑩. Before executing this global piece of code, Runtime has struck 0s.

07.00

Runtime will push the callback timer2() inside Event Queue.

timer1() timer2()

event queue.

"End of file" gets pointed.

Event loop found out that ① callstack is empty.

✉ @

② No global piece of code left.

Event loop found that microtask queue has 1 handler
Event queue has 2 callbacks.

NOVEMBER 2017

SUN	M	T	W	T	F	S	S
45		1	2	3	4	5	
46	6	7	8	9	10	11	12
47	13	14	15	16	17	18	19
48	20	21	22	23	24	25	26
49	27	28	29	30			

TUESDAY 2017

43rd Week • 290-075
OCTOBER 17

09:00 Event Loop prioritized handler inside microtask queue.

10:00 Event Loop pushed the handler from microtask queue to callstack.

11:00 Once this entry has been made we go inside ⑥.

processPromise()

callstack.

⑦ "Whose promise? Subhadip" gets printed.

03:00 Nothing else to execute. The entry corresponding to this handler gets removed.

04:00 Callstack gets empty.

⑧ Event Loop will one by one push callbacks from event queue to callstack.

06:00

07:00

Once this entry has been made we go inside ⑨.

timer1()

callstack.

⑩ "Timer1 done" gets printed.

Nothing else to execute. The entry gets removed from callstack. Callstack gets empty.

✉ @ ☎ Event loop pushes the 2nd callback in callstack.

Once this entry is made we go inside ⑪.

timer2()

callstack.

2017

WEDNESDAY

18

43rd Week • 291-074

OCTOBER

OCTOBER 2017						
W	M	T	W	T	F	S
40	30	31				
41	2	3	4	5	6	7
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28

⑨ "Timer 2 done" gets printed.

09.00

Nothing else to execute. The entry corresponding to callback gets removed.
Callstack gets empty.

11.00

① function blocking - for-loop()

12.00

② for(let i=0; i<10000000000; i++) {
}

01.00

02.00

03.00

04.00

05.00

06.00

07.00

08.00

09.00

10.00

11.00

12.00

13.00

14.00

15.00

16.00

17.00

18.00

19.00

20.00

21.00

22.00

23.00

24.00

25.00

26.00

27.00

28.00

29.00

30.00

31.00

32.00

33.00

34.00

35.00

36.00

37.00

38.00

39.00

40.00

41.00

42.00

43.00

44.00

45.00

46.00

47.00

48.00

49.00

50.00

51.00

52.00

53.00

54.00

55.00

56.00

57.00

58.00

59.00

60.00

61.00

62.00

63.00

64.00

65.00

66.00

67.00

68.00

69.00

70.00

71.00

72.00

73.00

74.00

75.00

76.00

77.00

78.00

79.00

80.00

81.00

82.00

83.00

84.00

85.00

86.00

87.00

88.00

89.00

90.00

91.00

92.00

93.00

94.00

95.00

96.00

97.00

98.00

99.00

100.00

101.00

102.00

103.00

104.00

105.00

106.00

107.00

108.00

109.00

110.00

111.00

112.00

113.00

114.00

115.00

116.00

117.00

118.00

119.00

120.00

121.00

122.00

123.00

124.00

125.00

126.00

127.00

128.00

129.00

130.00

131.00

132.00

133.00

134.00

135.00

136.00

137.00

138.00

139.00

140.00

141.00

142.00

143.00

144.00

145.00

146.00

147.00

148.00

149.00

150.00

151.00

152.00

153.00

154.00

155.00

156.00

157.00

158.00

159.00

160.00

161.00

162.00

163.00

164.00

165.00

166.00

167.00

168.00

169.00

170.00

171.00

172.00

173.00

174.00

175.00

176.00

177.00

178.00

179.00

180.00

181.00

182.00

183.00

184.00

185.00

186.00

187.00

188.00

189.00

190.00

191.00

192.00

193.00

194.00

195.00

196.00

197.00

198.00

199.00

200.00

201.00

202.00

203.00

204.00

205.00

206.00

207.00

208.00

209.00

210.00

211.00

212.00

213.00

214.00

215.00

216.00

217.00

218.00

219.00

220.00

221.00

222.00

223.00

224.00

225.00

226.00

227.00

228.00

229.00

230.00

231.00

232.00

233.00

234.00

235.00

236.00

237.00

238.00

239.00

240.00

241.00

242.00

243.00

244.00

245.00

246.00

247.00

248.00

249.00

250.00

251.00

252.00

253.00

254.00

255.00

256.00

257.00

258.00

259.00

260.00

261.00

262.00

263.00

264.00

265.00

266.00

267.00

268.00

269.00

270.00

271.00

272.00

273.00

274.00

275.00

276.00

277.00

278.00

279.00

280.00

281.00

282.00

283.00

284.00

285.00

NOVEMBER 2017

W	M	T	W	T	F	S	S
45		1	2	3	4	5	
46	6	7	8	9	10	11	12
47	13	14	15	16	17	18	19
48	20	21	22	23	24	25	26
49	27	28	29	30			

THURSDAY 2017

43rd Week • 292-073
OCTOBER

19

③ Global piece of code:- "Start of the file" gets printed.

④ This piece of code is not Native to JS. This feature is being provided to JS by Runtime.

JS notifies Runtime to start a timer of OS, post which Runtime will push the callback timer in the Event queue.

12.00 JS comes back. In the background a timer of OS has started off.

⑤ At this point, the runtime has already struck OS.

Runtime will push the callback timer in our ~~event loop~~ Event queue.

03.00 Event queue will look something like:-

04.00 Timer1()

05.00 event queue.

06.00 Current step is a function call. We will go inside ①.

07.00 ②. This piece of code is Native to JS. JS will block here until and unless the FOR loop is done executing.
JS blocks here for 10s.

③. We are returning a resolved promise object with value "Subhadip's promise".

X = value: "Subhadip's Promise!"

state: fulfilled.

onfulfillment: []

onrejection: []

promise object.

2017

FRIDAY

20

43rd Week • 293-072

OCTOBER

OCTOBER 2017

W	M	T	W	T	F	S
40	30	31				
41	2	3	4	5	6	7
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28

- ⑧. We are using .then functionality.

09.00

• then function registers handlers in our promise object

10.00

∴ There is a single handler inside our p.then function.

11.00

This must be a fulfill handler.

12.00

Our promise object will look something like:—

01.00

After registration, since state is fulfilled
we will push the handler inside microtask queue.

02.00

value: "Subhadip's Promise!"

03.00

state: fulfilled.

04.00

onfulfillment: [processPromise]

05.00

onrejection: []

processPromise()

microtask queue.

promise object.

- ⑪. We are returning a resolved promise object with value "Subhadip's Promise2".

06.00

value: "Subhadip's Promise2"

state: "fulfilled"

onfulfillment: []

onrejection: []

promise object.

- ⑫. We are using .then function.

✉ @ • then function registers handlers in our Promise object.

This handler will be a fulfill handler (∴ this is a single handler inside .then).

NOVEMBER 2017

W	M	T	W	F	S	S
45		1	2	3	4	5
46	6	7	8	9	10	11
47	13	14	15	16	17	18
48	20	21	22	23	24	25
49	27	28	29	30		

SATURDAY

2017

43rd Week • 294-071

OCTOBER

21

Our promise object will look something like:-

y = {
 value: "Subhadip's Promise 2"
 state: fulfilled}

onfulfillment: [processPromise]
on rejection: []

promise object

After registration, since state is fulfilled we will push the handler inside microtask queue.

processPromise()
processPromise()

microtask queue.

(15) We are returning a resolved promise object with value "Subhadip's Promise 3".

The promise object will look something like:-

z = {
 value: "Subhadip's Promise 3"
 state: fulfilled}

onfulfillment: []
on rejection: []

SUNDAY 22

value: "Subhadip's Promise 3"

state: fulfilled

onfulfillment: [processPromise]
on rejection: []

promise
object.

(16) ☐ We are using .then function.

.then registers handlers in our Promise object.

Only 1 handler inside .then. It will be fulfill Handler.

2017 MONDAY

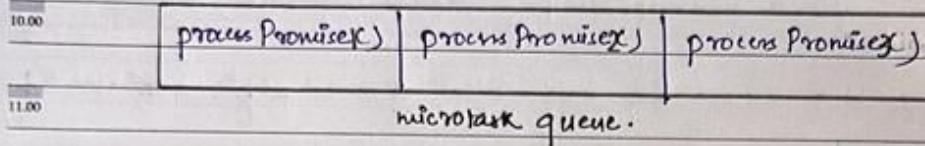
23

44th Week • 296-069

OCTOBER

OCTOBER 2017						
SUN	MON	TUE	WED	THU	FRI	SAT
30	31					
41	2	3	4	5	6	7
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28

After registration is done, since the promise object is fulfilled we will push the handlers inside on fulfillment array inside microtask queue.



(18) 12.00 This piece of code is not native to JS. This feature is being provided to JS by Runtime.

01.00 JS notifies Runtime to start a timer of Os. Post which Runtime is going to push the callback timer2() in the Event Queue.

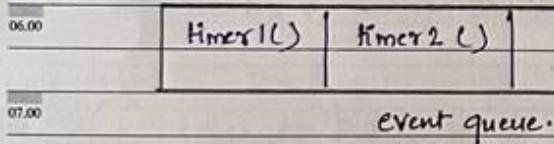
02.00

JS comes back. In the background, a timer of Os has started.

03.00

(19) At this step, the timer already struck Os. Runtime is going to push the callback timer2() in the event queue.

04.00 Event queue will look something like:-



"End of file" gets printed.

Event Loop at this point found that (1) call stack is empty.

(2) no global piece of code left.

In between microtask of event queue, event loop will prioritise microtask queue.

✉ @ 1st handler from microtask queue gets pushed to callstack.

Once this entry is done we will go inside (3).

processPromise1()

callstack.

NOVEMBER 2017						
S	M	T	W	T	F	S
	1	2	3	4	5	
45	6	7	8	9	10	11
46	13	14	15	16	17	18
47	20	21	22	23	24	25
48	27	28	29	30		

TUESDAY 2017

44th Week • 297-068

OCTOBER

24

(9) "Whose promise? Subhadip's Promise" gets printed.

(10) Function call :- The entry corresponding to this func call gets registered.

Once this entry is made we will go inside (1).

blocking-for-loop.

processPromise1()

callstack.

(11) This FOR loop is native to JS. JS will block here unless and until the FOR loop resolves.

JS will block here approximately for 10s.

Nothing more to execute. So the entry corresponding to (1) gets popped off.

Nothing more to execute inside (8). The entry corresponding to processPromise1 gets popped off.

Callstack gets empty.

Event loop will push the 2nd handler from microtask queue to callstack.

processPromise2()

callstack.

(12) "Whose promise? Subhadip's Promise2" gets printed.

This piece of code is not native to JS. This feature is provided to JS by Runtime. JS notifies Runtime to start a timer of 0s, part of which Runtime will push the callback in the event queue.

2017 WEDNESDAY

25

44th Week • 298-067

OCTOBER

OCTOBER 2017						
SUN	MON	TUE	WED	THU	FRI	SAT
40	30	31				1
41	2	3	4	5	6	7
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28

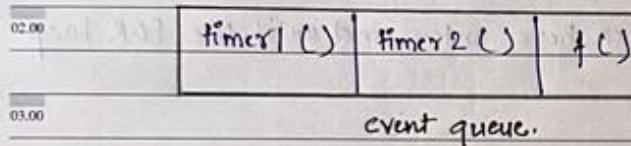
JS comes back and in the background a timer of Os has started.
09.00

Nothing more to execute inside processPromise2(). We remove its entry from callstack.

Callstack gets empty.
11.00

At this point runtime has struck Os. Runtime will push the anonymous callback in the event queue.

The 09.00 event queue will look like:-



Event loop will now push the final handler into the callstack.

05.00

06.00

Once this entry has been made, we go inside ⑯.

processPromise3()

callback.

⑯

"Those promise, Subhadip's Promise 3" gets executed.

Nothing else to execute. Entry gets removed. Callstack gets empty.

Event loop will now one by one push callbacks from event queue to callstack.

✉ @ ☎

timer1()

Once this entry is done we go inside ⑭.

call stack.

⑮ "Timer1 done" gets printed. Callstack gets empty.

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

THURSDAY 2017

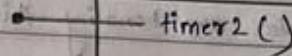
44th Week • 299-066

OCTOBER

26

Event Loop pushes the 2nd callback into callstack.

We go inside (18).



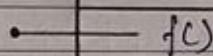
(19) "Timer2 done" gets printed.

callstack.

Nothing more to execute. Entry gets removed. Callstack gets empty.

Event Loop pushes the final anonymous callback in callstack.

We go inside (14).



"OK done" gets printed.

callstack.

Nothing more to execute. The entry gets removed. The callstack gets empty.

How promises prevent Inversion of Control!

Inversion of control:-

```
function download(url, cb){
```

I am giving control of my func.
"callback" to another func.
"download".

```
    Set Timeout(function exec(){
```

:: I do not know "download"'s
internal implementation.

```
        const data = "dummy data";
```

I have no idea how my
"callback" is being implemented
inside it.

```
        cb(data);
```

```
    }, 5000);
```

```
}
```

```
download("xyz", function callback(data){
```

```
    console.log("Data is," data);
```

```
});
```

2017 FRIDAY

27

44th Week • 300-065

OCTOBER

OCTOBER 2017						
Wk	M	T	W	T	F	S
40	30	31				1
41	2	3	4	5	6	7
42	9	10	11	12	13	14
43	16	17	18	19	20	21
44	23	24	25	26	27	28

Another issue is that, we are expecting that our "callback" will only be called once inside "download" if someone called it twice inside download

10.00
How promises are handling this

11.00
function download(url){

12.00 return new Promise(function exec(res, rej){
01.00 setTimeout(function (){
02.00 let data = "dummy data";
03.00 res(data);
04.00 res(data);
05.00 } , 5000);
});

} Even if it is called twice,
value will be displayed once once
because state transition & value
assignment already done.

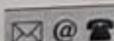
06.00 download("xyt").then(function fulfill(value){

07.00 console.log("The value is", value);
});

"download" function is not having access to any callbacks

∴ We are not giving access of my callback to "download" func.

important



NOVEMBER 2017

W	M	T	W	T	F	S	S
45		1	2	3	4	5	
46	6	7	8	9	10	11	12
47	13	14	15	16	17	18	19
48	20	21	22	23	24	25	26
49	27	28	29	30			

SATURDAY 2017

44th Week • 301-064

OCTOBER

28

Dot then chaining:-

• then function returns a promise object.

```
function download(url) {
```

```
    return new Promise(function executor(resolve, reject) {
```

```
        setTimeout(function p() {
```

```
            const content = "ABCDEF";
```

```
            resolve(content);
```

```
        }, 5000);
```

```
    });
```

```
});
```

```
x = download("www.glee.com");
```

```
x.then(function fh1(value) {
```

```
    console.log("Content downloaded is", value);
```

```
    return "new promise string";
```

```
});
```

This code will return
a new promise
object.

SUNDAY 29

```
.then(function fh2(value) {
```

```
    console.log("Value from chained then promise", value);
```

```
});
```

↓
This will be fed as an IP to fh2.

• This is same as writing

```
y = x.then(...)
```

```
y.then(...)
```

If we do not return anything,
the default return is "undefined".

2017 MONDAY

30

45th Week • 303-062

OCTOBER

- ① Promise.resolve("foo")
09.00 → A
- ② .then(function fh1(value){
10.00
③ return new Promise(function exec(resolve, reject){
11.00
④ setTimeout(function (){
12.00
⑤ value += "bar";
01.00 → B
⑥ resolve(value);
02.00
03.00
04.00
05.00
06.00
07.00
08.00
09.00
10.00
11.00
12.00
13.00
14.00
15.00
16.00
17.00
18.00
19.00
20.00
21.00
22.00
23.00
24.00
25.00
26.00
27.00
28.00
29.00
30.00
31.00
32.00
33.00
34.00
35.00
36.00
37.00
38.00
39.00
40.00
41.00
42.00
43.00
44.00
45.00
46.00
47.00
48.00
49.00
50.00
51.00
52.00
53.00
54.00
55.00
56.00
57.00
58.00
59.00
60.00
61.00
62.00
63.00
64.00
65.00
66.00
67.00
68.00
69.00
70.00
71.00
72.00
73.00
74.00
75.00
76.00
77.00
78.00
79.00
80.00
81.00
82.00
83.00
84.00
85.00
86.00
87.00
88.00
89.00
90.00
91.00
92.00
93.00
94.00
95.00
96.00
97.00
98.00
99.00
100.00
101.00
102.00
103.00
104.00
105.00
106.00
107.00
108.00
109.00
110.00
111.00
112.00
113.00
114.00
115.00
116.00
117.00
118.00
119.00
120.00
121.00
122.00
123.00
124.00
125.00
126.00
127.00
128.00
129.00
130.00
131.00
132.00
133.00
134.00
135.00
136.00
137.00
138.00
139.00
140.00
141.00
142.00
143.00
144.00
145.00
146.00
147.00
148.00
149.00
150.00
151.00
152.00
153.00
154.00
155.00
156.00
157.00
158.00
159.00
160.00
161.00
162.00
163.00
164.00
165.00
166.00
167.00
168.00
169.00
170.00
171.00
172.00
173.00
174.00
175.00
176.00
177.00
178.00
179.00
180.00
181.00
182.00
183.00
184.00
185.00
186.00
187.00
188.00
189.00
190.00
191.00
192.00
193.00
194.00
195.00
196.00
197.00
198.00
199.00
200.00
201.00
202.00
203.00
204.00
205.00
206.00
207.00
208.00
209.00
210.00
211.00
212.00
213.00
214.00
215.00
216.00
217.00
218.00
219.00
220.00
221.00
222.00
223.00
224.00
225.00
226.00
227.00
228.00
229.00
230.00
231.00
232.00
233.00
234.00
235.00
236.00
237.00
238.00
239.00
240.00
241.00
242.00
243.00
244.00
245.00
246.00
247.00
248.00
249.00
250.00
251.00
252.00
253.00
254.00
255.00
256.00
257.00
258.00
259.00
260.00
261.00
262.00
263.00
264.00
265.00
266.00
267.00
268.00
269.00
270.00
271.00
272.00
273.00
274.00
275.00
276.00
277.00
278.00
279.00
280.00
281.00
282.00
283.00
284.00
285.00
286.00
287.00
288.00
289.00
290.00
291.00
292.00
293.00
294.00
295.00
296.00
297.00
298.00
299.00
300.00
301.00
302.00
303.00
304.00
305.00
306.00
307.00
308.00
309.00
310.00
311.00
312.00
313.00
314.00
315.00
316.00
317.00
318.00
319.00
320.00
321.00
322.00
323.00
324.00
325.00
326.00
327.00
328.00
329.00
330.00
331.00
332.00
333.00
334.00
335.00
336.00
337.00
338.00
339.00
340.00
341.00
342.00
343.00
344.00
345.00
346.00
347.00
348.00
349.00
350.00
351.00
352.00
353.00
354.00
355.00
356.00
357.00
358.00
359.00
360.00
361.00
362.00
363.00
364.00
365.00
366.00
367.00
368.00
369.00
370.00
371.00
372.00
373.00
374.00
375.00
376.00
377.00
378.00
379.00
380.00
381.00
382.00
383.00
384.00
385.00
386.00
387.00
388.00
389.00
390.00
391.00
392.00
393.00
394.00
395.00
396.00
397.00
398.00
399.00
400.00
401.00
402.00
403.00
404.00
405.00
406.00
407.00
408.00
409.00
410.00
411.00
412.00
413.00
414.00
415.00
416.00
417.00
418.00
419.00
420.00
421.00
422.00
423.00
424.00
425.00
426.00
427.00
428.00
429.00
430.00
431.00
432.00
433.00
434.00
435.00
436.00
437.00
438.00
439.00
440.00
441.00
442.00
443.00
444.00
445.00
446.00
447.00
448.00
449.00
450.00
451.00
452.00
453.00
454.00
455.00
456.00
457.00
458.00
459.00
460.00
461.00
462.00
463.00
464.00
465.00
466.00
467.00
468.00
469.00
470.00
471.00
472.00
473.00
474.00
475.00
476.00
477.00
478.00
479.00
480.00
481.00
482.00
483.00
484.00
485.00
486.00
487.00
488.00
489.00
490.00
491.00
492.00
493.00
494.00
495.00
496.00
497.00
498.00
499.00
500.00
501.00
502.00
503.00
504.00
505.00
506.00
507.00
508.00
509.00
510.00
511.00
512.00
513.00
514.00
515.00
516.00
517.00
518.00
519.00
520.00
521.00
522.00
523.00
524.00
525.00
526.00
527.00
528.00
529.00
530.00
531.00
532.00
533.00
534.00
535.00
536.00
537.00
538.00
539.00
540.00
541.00
542.00
543.00
544.00
545.00
546.00
547.00
548.00
549.00
550.00
551.00
552.00
553.00
554.00
555.00
556.00
557.00
558.00
559.00
560.00
561.00
562.00
563.00
564.00
565.00
566.00
567.00
568.00
569.00
570.00
571.00
572.00
573.00
574.00
575.00
576.00
577.00
578.00
579.00
580.00
581.00
582.00
583.00
584.00
585.00
586.00
587.00
588.00
589.00
590.00
591.00
592.00
593.00
594.00
595.00
596.00
597.00
598.00
599.00
600.00
601.00
602.00
603.00
604.00
605.00
606.00
607.00
608.00
609.00
610.00
611.00
612.00
613.00
614.00
615.00
616.00
617.00
618.00
619.00
620.00
621.00
622.00
623.00
624.00
625.00
626.00
627.00
628.00
629.00
630.00
631.00
632.00
633.00
634.00
635.00
636.00
637.00
638.00
639.00
640.00
641.00
642.00
643.00
644.00
645.00
646.00
647.00
648.00
649.00
650.00
651.00
652.00
653.00
654.00
655.00
656.00
657.00
658.00
659.00
660.00
661.00
662.00
663.00
664.00
665.00
666.00
667.00
668.00
669.00
670.00
671.00
672.00
673.00
674.00
675.00
676.00
677.00
678.00
679.00
680.00
681.00
682.00
683.00
684.00
685.00
686.00
687.00
688.00
689.00
690.00
691.00
692.00
693.00
694.00
695.00
696.00
697.00
698.00
699.00
700.00
701.00
702.00
703.00
704.00
705.00
706.00
707.00
708.00
709.00
710.00
711.00
712.00
713.00
714.00
715.00
716.00
717.00
718.00
719.00
720.00
721.00
722.00
723.00
724.00
725.00
726.00
727.00
728.00
729.00
730.00
731.00
732.00
733.00
734.00
735.00
736.00
737.00
738.00
739.00
740.00
741.00
742.00
743.00
744.00
745.00
746.00
747.00
748.00
749.00
750.00
751.00
752.00
753.00
754.00
755.00
756.00
757.00
758.00
759.00
760.00
761.00
762.00
763.00
764.00
765.00
766.00
767.00
768.00
769.00
770.00
771.00
772.00
773.00
774.00
775.00
776.00
777.00
778.00
779.00
779.00
780.00
781.00
782.00
783.00
784.00
785.00
786.00
787.00
788.00
789.00
790.00
791.00
792.00
793.00
794.00
795.00
796.00
797.00
798.00
799.00
800.00
801.00
802.00
803.00
804.00
805.00
806.00
807.00
808.00
809.00
8010.00
8011.00
8012.00
8013.00
8014.00
8015.00
8016.00
8017.00
8018.00
8019.00
8020.00
8021.00
8022.00
8023.00
8024.00
8025.00
8026.00
8027.00
8028.00
8029.00
8030.00
8031.00
8032.00
8033.00
8034.00
8035.00
8036.00
8037.00
8038.00
8039.00
8040.00
8041.00
8042.00
8043.00
8044.00
8045.00
8046.00
8047.00
8048.00
8049.00
8050.00
8051.00
8052.00
8053.00
8054.00
8055.00
8056.00
8057.00
8058.00
8059.00
8060.00
8061.00
8062.00
8063.00
8064.00
8065.00
8066.00
8067.00
8068.00
8069.00
8070.00
8071.00
8072.00
8073.00
8074.00
8075.00
8076.00
8077.00
8078.00
8079.00
8080.00
8081.00
8082.00
8083.00
8084.00
8085.00
8086.00
8087.00
8088.00
8089.00
8090.00
8091.00
8092.00
8093.00
8094.00
8095.00
8096.00
8097.00
8098.00
8099.00
80100.00
80101.00
80102.00
80103.00
80104.00
80105.00
80106.00
80107.00
80108.00
80109.00
80110.00
80111.00
80112.00
80113.00
80114.00
80115.00
80116.00
80117.00
80118.00
80119.00
80120.00
80121.00
80122.00
80123.00
80124.00
80125.00
80126.00
80127.00
80128.00
80129.00
80130.00
80131.00
80132.00
80133.00
80134.00
80135.00
80136.00
80137.00
80138.00
80139.00
80140.00
80141.00
80142.00
80143.00
80144.00
80145.00
80146.00
80147.00
80148.00
80149.00
80150.00
80151.00
80152.00
80153.00
80154.00
80155.00
80156.00
80157.00
80158.00
80159.00
80160.00
80161.00
80162.00
80163.00
80164.00
80165.00
80166.00
80167.00
80168.00
80169.00
80170.00
80171.00
80172.00
80173.00
80174.00
80175.00
80176.00
80177.00
80178.00
80179.00
80180.00
80181.00
80182.00
80183.00
80184.00
80185.00
80186.00
80187.00
80188.00
80189.00
80190.00
80191.00
80192.00
80193.00
80194.00
80195.00
80196.00
80197.00
80198.00
80199.00
80200.00
80201.00
80202.00
80203.00
80204.00
80205.00
80206.00
80207.00
80208.00
80209.00
80210.00
80211.00
80212.00
80213.00
80214.00
80215.00
80216.00
80217.00
80218.00
80219.00
80220.00
80221.00
80222.00
80223.00
80224.00
80225.00
80226.00
80227.00
80228.00
80229.00
80230.00
80231.00
80232.00
80233.00
80234.00
80235.00
80236.00
80237.00
80238.00
80239.00
80240.00
80241.00
80242.00
80243.00
80244.00
80245.00
80246.00
80247.00
80248.00
80249.00
80250.00
80251.00
80252.00
80253.00
80254.00
80255.00
80256.00
80257.00
80258.00
80259.00
80260.00
80261.00
80262.00
80263.00
80264.00
80265.00
80266.00
80267.00
80268.00
80269.00
80270.00
80271.00
80272.00
80273.00
80274.00
80275.00
80276.00
80277.00
80278.00
80279.00
80280.00
80281.00
80282.00
80283.00
80284.00
80285.00
80286.00
80287.00
80288.00
80289.00
80290.00
80291.00
80292.00
80293.00
80294.00
80295.00
80296.00
80297.00
80298.00
80299.00
80300.00
80301.00
80302.00
80303.00
80304.00
80305.00
80306.00
80307.00
80308.00
80309.00
80310.00
80311.00
80312.00
80313.00
80314.00
80315.00
80316.00
80317.00
80318.00
80319.00
80320.00
80321.00
80322.00
80323.00
80324.00
80325.00
80326.00
80327.00
80328.00
80329.00
80330.00
80331.00
80332.00
80333.00
80334.00
80335.00
80336.00
80337.00
80338.00
80339.00
80340.00
80341.00
80342.00
80343.00
80344.00
80345.00
80346.00
80347.00
80348.00
80349.00
80350.00
80351.00
80352.00
80353.00
80354.00
80355.00
80356.00
80357.00
80358.00
80359.00
80360.00
80361.00
80362.00
80363.00
80364.00
80365.00
80366.00
80367.00
80368.00
80369.00
80370.00
80371.00
80372.00
80373.00
80374.00
80375.00
80376.00
80377.00
80378.00
80379.00
80380.00
80381.00
80382.00
80383.00
80384.00
80385.00
80386.00
80387.00
80388.00
80389.00
80390.00
80391.00
80392.00
80393.00
80394.00
80395.00
80396.00
80397.00
80398.00
80399.00
80400.00
80401.00
80402.00
80403.00
80404.00
80405.00
80406.00
80407.00
80408.00
80409.00
80410.00
80411.00
80412.00
80413.00
80414.00
80415.00
80416.00
80417.00
80418.00
80419.00
80420.00
80421.00
80422.00
80423.00
80424.00
80425.00
80426.00
80427.00
80428.00
80429.00
80430.00
80431.00
80432.00
80433.00
80434.00
80435.00
80436.00
80437.00
80438.00
80439.00
80440.00
80441.00
80442.00
80443.00
80444.00
80445.00
80446.00
80447.00
80448.00
80449.00
80450.00
80451.00
80452.00
80453.00
80454.00
80455.00
80456.00
80457.00
80458.00
80459.00
80460.00
80461.00
80462.00
80463.00
80464.00
80465.00
80466.00
80467.00
80468.00
80469.00
8

NOVEMBER 2017

W	M	T	W	T	F	S	S
45		1	2	3	4	5	
46	6	7	8	9	10	11	12
47	13	14	15	16	17	18	19
48	20	21	22	23	24	25	26
49	27	28	29	30			

TUESDAY 2017

45th Week • 304-061

OCTOBER

31

- ①. We are returning a new promise object with value as "foo" and state as "fulfilled".

Let's call this A.

A = value: "foo"
state: fulfilled
onfulfill: [] promise object.
onreject: []

- ②. We are doing .then operation.

• then operation registers handlers in our promise object

A = value: "foo"
state: fulfilled
onfulfill: [f1] promise object
onreject: []

We are doing basically A.then(x,y)

A.then also returns a new promise object in itself. (Let's call this B).

B = value: undefined
state: pending
onfulfill: []
onreject: [].

2017

WEDNESDAY

01

45th Week • 305-060

NOVEMBER

⑦. On **B** we are doing .then.

09.00

∴ **B** will look something like

10.00

11.00

B = Value: undefined

state: pending

onfulfill: [f₂]

onreject: []

12.00

01.00

02.00

03.00

B.then also will return us a new Promise object (lets call it c).

04.00

05.00

value: undefined

state: pending

onfulfill: []

onreject: [].

06.00

07.00

⑧. On **C** we are doing .then.

∴ **C** will look something like.

C =

value: undefined

state: pending

onfulfill: [f₃]

onreject: [].

NOVEMBER 2017						
W	M	T	W	T	F	S
45		1	2	3	4	
46	6	7	8	9	10	11
47	13	14	15	16	17	18
48	20	21	22	23	24	25
49	27	28	29	30		

DECEMBER 2017

W	M	T	W	T	F	S	S
49			1	2	3		
50	4	5	6	7	8	9	10
51	11	12	13	14	15	16	17
52	18	19	20	21	22	23	24
53	25	26	27	28	29	30	31

THURSDAY

2017

45th Week • 306-059

NOVEMBER

02

C. then will also return us a new promise object (lets call it D).

10.00 value: undefined
D = state: pending
11.00 onfulfilled: []
onreject: [] .
12.00

Now,

02.00 From step ② we came to know that A is already fulfilled promise.
03.00 if we are also done registering.

04.00 JS will push the handler fhl in microtask queue.

05.00

fh1()

06.00 microtask queue.

07.00 Event Loop found out that nothing more to execute in callstack if no global piece of code left.

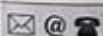
Event Loop pushes the handler from microtask queue to callstack.

Once this entry is done we go inside ②.

•

fh1()

callstack.



2017 FRIDAY

03

45th Week • 307-058

NOVEMBER

- ③. It returns a new promise object.

So in the memory a new promise object get created.

11:00 value: ~~undefined~~ → undefined
State: pending
onfulfill: []
onreject: []

promise object.

During creation of this promise object we will call the executor func.

Inside the executor func. we have

- ④. This piece of code is not native to JS. Its a runtime based feature

In the background a timer of 10s started off.

Nothing more to execute

So we return the promise object.

Important

We are not returning primitive value from f1, we are returning a promise object. So unless and until the promise object resolves/reject we are not going to execute f2, f3.

Important

If we are returning a primitive value from f1 ~~then it has been considered~~ the state of f1 get fulfilled. ~~as rejected promise.~~ ~~and~~ we would have started off with f2 or f3.

DECEMBER 2017

W	M	T	W	F	S	S
49			1	2	3	
50	4	5	6	7	8	9 10
51	11	12	13	14	15	16 17
52	18	19	20	21	22	23 24
53	25	26	27	28	29	30 31

SATURDAY

2017

45th Week • 308-057

NOVEMBER

04

∴ The promises object state is pending; function callback.

09.00

Nothing more to execute. Entry of fhi got removed.

10.00

callstack gets empty.

11.00

From line ⑦ onwards nothing will be ~~printed~~ executed as of now.

01.00

When timer struck 10s. RunTime will push the callback from runtime to

02.00

Event queue.

03.00

Event Loop will push the callback from event queue to callstack.

04.00

Once this entry is done we go inside ④ stating (5) → (6) post task

05.00

⑤ value gets updated to "foobar"

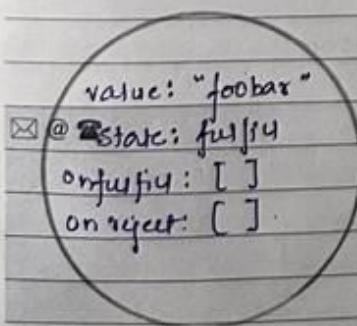
SUNDAY 05

⑥ We are resolving the promise.

Note*

This value that we are updating is not the value from promise object. This value is referring to line ②'s parameter.

∴ Promise object becomes:-



callstack gets empty.

2017 MONDAY

06

46th Week • 310-055

NOVEMBER

NOVEMBER 2017						
W	M	T	W	T	F	S
45			1	2	3	4
46	6	7	8	9	10	11
47	13	14	15	16	17	18
48	20	21	22	23	24	25
49	27	28	29	30		

④ Once the promise resolves.

⑤ The resolved promise's value "foobar" is used as a parameter in f2.

⑥ This piece of code is not native to JS. Its a runtime based feature.

In the background a timer of 1ms started off.

⑪ We are saying return value inside 'then'.

We are actually returning a resolved promise with value "foobar".

⑫ At this point runtime has struck 1s.

Run time pushes the callback inside Event queue.

⑬ "Foobar" gets printed.

Event Loop found out that ① callstack is empty

② no global piece of code left.

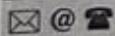
③ microtask queue is empty.

Event loop pushes the callback from event queue to callstack.

Once inside callstack we go inside ⑧.

⑨ value gets updated. to "Foobarbaz"

⑩ "Foobarbaz" gets printed.



DECEMBER 2017

SUN	M	T	W	T	F	S	S
49		1	2	3			
50	4	5	6	7	8	9	10
51	11	12	13	14	15	16	17
52	18	19	20	21	22	23	24
53	25	26	27	28	29	30	31

TUESDAY 2017

46th Week - 311-054

NOVEMBER

07

Set of Tasks:- (Using only callbacks)

- (1) Write a function to download data from url
10.00
- (2) Write a func. to save that downloaded data in a file and return file name
11.00
- (3) Write a func. to upload the file written in the prev. step to a new url.
12.00

(1) function download (url, callback1) {
07.00

(2) console.log ("Starting to download from ", url); @.

(3) setTimeout (function A() {
07.00

(4) console.log ("Download complete"); Ⓛ

(5) const data = "Dummy data";
07.00

(6) callback1 (data);
07.00 } , 4000);
07.00 }

(7) function writeFile (content, callback2) {
07.00

(8) console.log ("We are now saving your file..."); Ⓛ

(9) setTimeout (function B() {
07.00

(10) console.log ("We are done saving your file"); Ⓛ

(11) const fileName = "doc.txt";
07.00

(12) callback2 (fileName);
07.00 } , 6000);
07.00 }

2017

WEDNESDAY

08

46th Week • 312-053

NOVEMBER

NOVEMBER 2017						
WE	M	T	W	T	F	S
45		1	2	3	4	5
46	6	7	8	9	10	11
47	13	14	15	16	17	18
48	20	21	22	23	24	25
49	27	28	29	30		

⑯ function upload(url, file, callback3){
09.00}

⑰ console.log("We are now saving your file", file, "at url", url); ⑯
10.20

⑯ setTimeout(function f() {
11.00}

⑯ const message = "Success";
12.00

⑯ callback3(message);
01.00

}, 3000);
02.00

}

03.00

⑯ download("www.xyz.com", function cb1(content)){
04.00}

⑯ console.log("The download data is", content); ⑯
05.00

⑯ writeFile(content, function cb2(fileName)){
06.00}

⑯ console.log("We have saved your file", fileName); ⑯
07.00

⑯ upload("www.dropbox.com", fileName, function cb3(response)){
08.00}

⑯ console.log("The message is", response); ⑯
09.00

});
});

});
});

