06.00

07.00

—: JS:- Advanced Promises :—

{promise constructor :- mdn}.

✉ @ ☎

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 :-

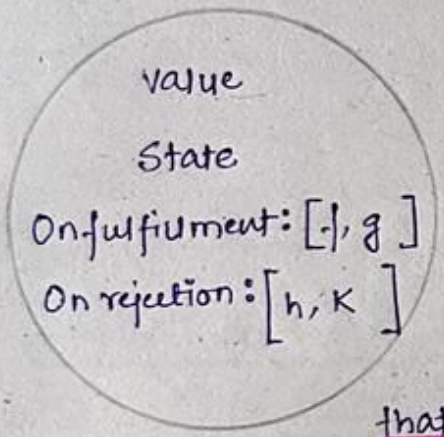
When we call a function, that returns us a promise. Inside that promise we can bind some asynchronous piece of logic. That asynchronous piece of logic might be completed in the future and the whole promise will be resolved in future.
Till the time the promise is not getting resolved, we get a placeholder 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 some time) → once we get fulfilled | rejected.
Unlike callBacks where we have to mention then 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 → functions.

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

∴ On fulfilment array contains the functions that we bind with .then.

When the value parameter is updated from undefined → X.

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

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

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

{Assuming 'p' as my Promise Object}.

p.then(fulfilment Handler, rejection handler)

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

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

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

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

Example:-

```
① function getRandomInt(max){
      return Math.floor(Math.random() * max);
  }
② function createPromiseWithTimeout(){
③     return new Promise(function executor(resolve, reject){
④         setTimeout(function(){
⑤             let num = getRandomInt(10);
⑥             if(num % 2 == 0){
                  resolve(num);
              }
⑦             else{
                  reject(num);
              }
          }, 10000);
      });
  }
```

```
(5) console.log("Starting");
09.00
(6) const p = createPromiseWithTimeout();
(7) console.log("Currently Promise Object looks like", p);
(8) console.log("We are waiting for promise to complete");
11.00
(9) p.then(function fulfilHandler(v){
12.00
        (10) console.log("Inside fulfilHandler 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.
06.00
Inside fulfilHandler with value, 8.
```

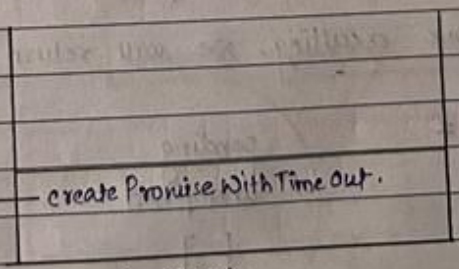
07.00

(5) Global piece of code:- (Starting) gets printed

O/P
Starting.

(5) 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).



CallStack.

2017 FRIDAY

08 37th Week • 251-114 SEPTEMBER

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	

O/P

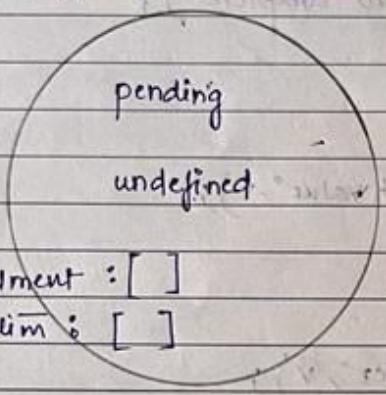
(2) Inside (2), we have (3).

09.00

Line (3) states that to return a new Promise.

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

11.00



12.00

01.00

02.00

Promise object.

03.00

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

05.00

Inside the Executor func. we have (4).

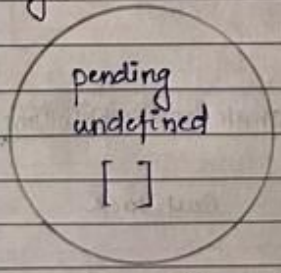
06.00

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

07.00 ∴ 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 = 

✉ @ ☎

∵ line (2) is done executing, the entry corresponding to the func. gets removed.

CallStack is now empty.

O/P.

(7) Global piece of code :- .

Currently promise object looks like pending undefined.

∴ We will print the promise in Pending State.

(8). Global piece of code :-

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

(9). We have .then operator.

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

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

"rejectHandler" gets registered inside our onrejection array.

Now, our promise object will look something like :-

value: ~~pend~~ undefined
state: pending.
Onfulfillment: [fulfilHandler]
Onrejection: [rejectionHandler]

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.

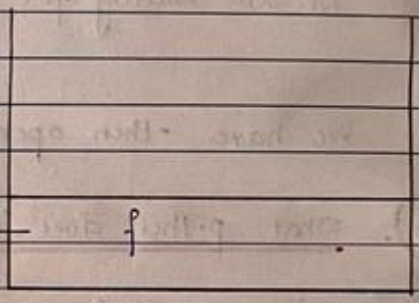
SUNDAY 10

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

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

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

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



callStack.

Once, this entry has been made we would go inside (4).

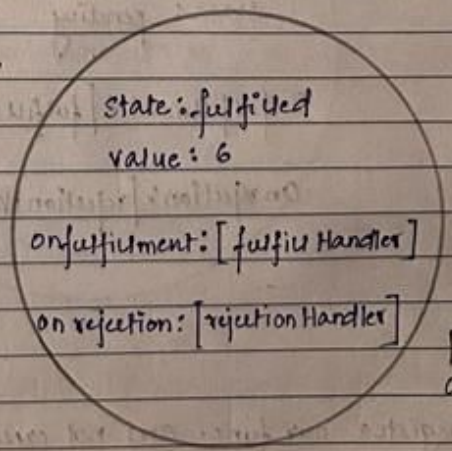
(4) Inside (4), we have (5).

(5) We generate a random number (let's say 6)

(6) Inside (6), we resolve the promise.

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

⊙ undefined goes to 6.



promise object.

After (b) no more code left to execute for the callback.

∴ We remove the entry corresponding to that callback from our callstack.

∴ callstack gets empty.

Now, since the state of the promise object is now Fulfilled.

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

2017 WEDNESDAY

13 38th Week • 256-109 SEPTEMBER

SEPTEMBER 2017

We	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	

—: Microtask Queue :—

```
09.00
(1) function createPromise () {
    (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 fulfilHandler (value) {
06.00
    (9) console.log ("We fulfiled with a value", value);
07.00
    },
    function rejectHandler (value) {
    (10) console.log ("We fulfiled with a value", value);
    }));
(11) console.log ("Ending");
```

Note:—

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

Once the state of the Promise changes to :—

① Fulfiled :— All the Handlers inside onfulfilment 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

W	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

⑤ Not Native to JS.

JS notifies Runtime to start a timer of 0s. Post which Runtime is going to push the callback in Event queue and JS will pick that up & execute it.

THURSDAY 2017

38th Week • 257-108

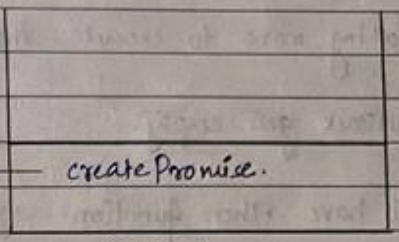
SEPTEMBER 14

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

10.00

11.00

12.00 Once, this entry has been made we will go to line ①.



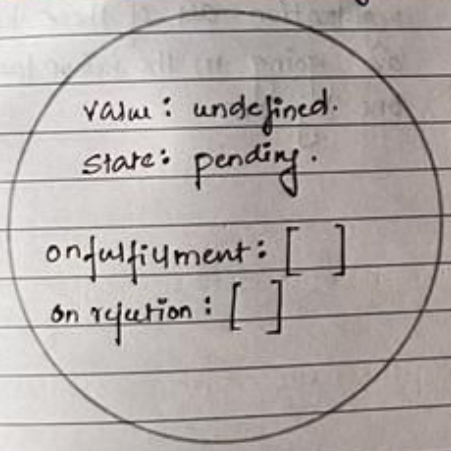
Callstack.

01.00

⑧. Inside ① we have a return Promise.

03.00 ∴ A new Promise object gets created in the memory.

04.00



promise object.

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

05.00

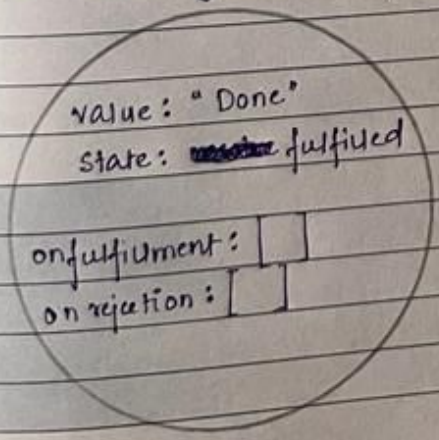
06.00

07.00

③. "Resolving the Promise" gets printed.

④ We are resolving the promise. ∴ Promise state goes from pending to resolve.

value goes from undefined to "Done".



promise object.

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

⑧ We have .then function.

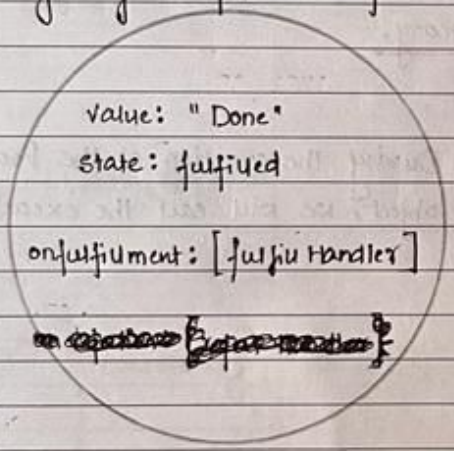
01.00

.then function registers the fulfilHandler and rejectHandler inside the Promise object.

02.00

∴ After registering our promise object looks like.

03.00



promise object.

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

04.00

05.00

06.00

07.00

⑨ ⑪ "Ending" gets printed.

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

imp

We have a callback inside EventQueue
&
Handlers inside Microtask Queue.

Priority will be given to Microtask Queue.

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

① Executing the callback from Event Queue.

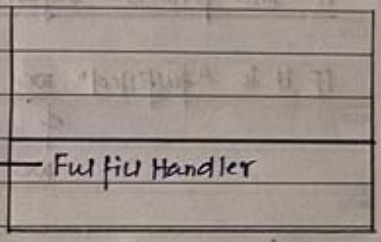
② Executing the Handler from Microtask Queue.

Event Loop will prioritise the microtask queue.

Fulfil Handler() goes from microtask queue to the callback.

Microtask queue gets empty.

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

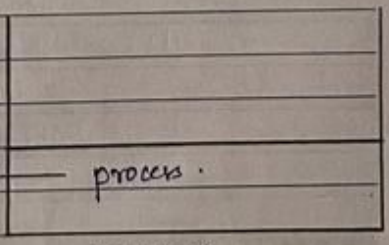


⑨. "We fulfilled with a value Done" gets printed.

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

Event Loop will now push the callback from Event Queue to callstack. SUNDAY 17
Event queue gets empty.

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



⑩. "Timer Completed" gets printed.

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

2017 MONDAY

18 39th Week • 261-104
SEPTEMBER

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	

09.00 -: Key Pointers :-

10.00 ① IF the promise object state is "Pending".

11.00 We will register both the handlers in our Promise object.

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

01.00

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

03.00 IF it is "fulfilled" we will register only the fulfilHandler in our Promise object.
&
we will push fulfilHandler in the Microtask queue.

04.00

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

06.00

07.00

✉ @ ☎

```
(1) function createPromise() {
    (2) return new Promise(function executor(resolve, reject) {
        (3) console.log("Resolving the promise");
        (4) resolve("Done");
    });
}

(5) setTimeout(function process() {
    (6) console.log("Timer Completed");
}, 0);

(7) let p = createPromise();

(8) p.then(function fulfillHandler1(value) {
    (9) console.log("We fulfilled 1 with a value", value);
},
function rejectHandler1(value) { });

(10) p.then(function fulfillHandler2(value) {
    (11) console.log("We fulfilled 2 with value," value);
},
function rejectHandler2(value) { });

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

(13) console.log("Ending");
```

(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 0s. Post which Run Time will push the

10.00 callback inside Eventqueue & from there JS is going to take care.

11.00 ∴ In the background a timer of 0s has started.

(7) 12.00 Function call :— By the time we reach this step, the Runtime timer has struck 0s.

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
onfulfillment: [ ]
on rejection: [ ]

promise object.

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

Inside the executor function,

09.00

③. "Resolving the Promise" gets printed.

10.00

④. We now resolve the promise.

11.00

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

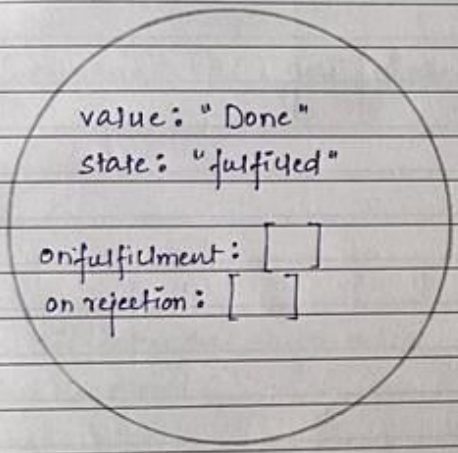
12.00

value goes from undefined to "value".

01.00

Our promise object looks like :—.

02.00



promise object.

03.00

Nothing more to execute inside executor function.

04.00

Inside ② nothing more to execute So, we will return this promise object to line ④.

05.00

Nothing more to execute inside ①.

06.00

∴ We will remove this entry from our callstack.

So, our callstack gets empty.

07.00

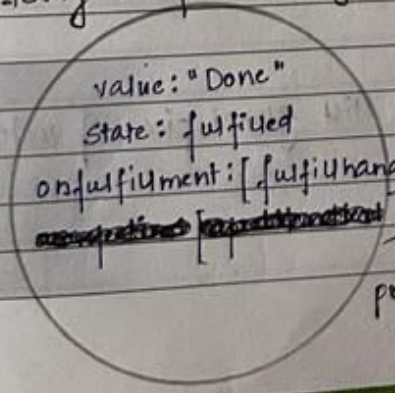
⑧. We have .then operator.

.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.



promise object

(10). We have .then operator

09.00

.then operator registers the handlers in our promise object.

10.00

Here, since the state of the promise object is "fulfilled". We will register

11.00

fulfilHandler2() inside our promise object.

12.00

01.00

value: "Done"
state: fulfilled
onfulfilment: [fulfilHandler1, fulfilHandler2]

promise object

∵ The state of the Promise object is fulfilled.
We will push "fulfilHandler2" inside our microtask queue.

02.00

03.00

04.00

05.00

06.00 And our microtask queue looks something like:—

07.00

fulfilHandler1()	fulfilHandler2()	

Microtask Queue.

Our Event queue looks something like:—

process()	

Event Queue.

✉ @ ☎

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

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

09.00

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

10.00

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.

02.00

03.00 ∴ fulfilHandler1() gets pushed inside the callstack.

04.00

05.00

06.00 Once this entry has been made, we will go inside fulfilHandler1().

07.00

fulfilHandler1

callstack.

(9). "We fulfilled1 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 fulfilHandler2() in our callstack.
& our microtask queue gets empty.

✉ @ ☎

Once this entry has been made we will go inside fulfilHandler2().

fulfilHandler2

callstack.

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	

09.00 (11) "We fulfilled 2 with value Done" gets printed.

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

11.00 callstack gets empty.

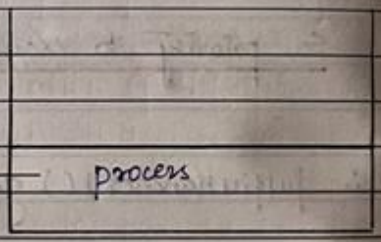
12.00 Now, event loop will one by one push callback from event queue to callStack.

01.00 Now, event queue gets empty.

02.00

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

04.00



callstack.

05.00 (6) "Timer Completed" gets printed.

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

07.00 Callstack gets empty.

```
(1) function createPromise() {
09.00
    (2) return new Promise(function executor(resolve, reject) {
10.00
        (3) setTimeout(function () {
11.00
            (4) console.log("Rejecting the promise");
            (5) reject("Done");
12.00
        }, 1000);
01.00
    });
}
02.00
(6) let p = createPromise();
03.00
(7) p.then(function fulfilHandler1(v) {
04.00
        (8) console.log("We fulfiled 1 with value", v);
05.00
    },
06.00
    function rejectHandler1(v){
07.00
        (9) console.log("We rejected 1 with value", v);
    });
(10) p.then(function fulfilHandler2(v) {
        (11) console.log("We ~~rejected~~ fulfiled 2 with value", v);
    },
    function rejectHandler2(v) {
        (12) console.log("We rejected 2 with value", v);
    });
(13) for(let i=0; i<10000000000; i++) { };
(14) console.log("Ending");
```

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

09.00

10.00

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

12.00 Inside ①; we have ②.

② 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.

03.00 value: undefined
State: pending

04.00 onfulfillment: [ ]
on rejection: [ ]

promise object

05.00 Inside the executor func. we have ③.

06.00

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.

(7). 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.

11.00

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

12.00

value: undefined
state: pending
onfulfillment: [fulfillhandler1]
on rejection: [rejectionhandler1]

01.00

promise object.

02.00

03.00

04.00

(10). 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
onfulfilment: [fulfilmenthandler1, fulfilmenthandler2]
on rejection: [rejectionhandler1, rejectionhandler2]

promise object.

✉ @ ☎

(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.

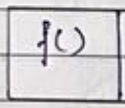
10.00

11.00 While the FOR loop is executing, the Runtime has struck 1s.

12.00 So the callback will be pushed inside the Event Queue.

01.00 Event Queue will look something like:—

02.00



03.00 Event Queue.

(14). "Ending" gets printed.

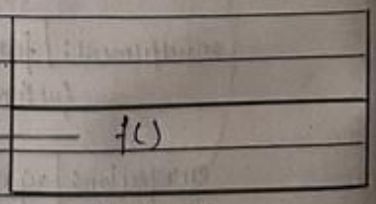
05.00 At this point event loop encountered the callstack is empty.
No global piece of code left.
Microtask queue is empty.

06.00

07.00 ∴ Event Loop will one by one pick up 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 callback (3).



Callstack.

✉ @ ☎

(3). Inside (3), we have (4).
"Rejecting the Promise" gets printed.

OCTOBER 2017

WI	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

SATURDAY 2017

40th Week • 273-092

SEPTEMBER 30

09.00 (5). We are ~~resolving~~ rejecting the promise. with value "Done".

10.00 ∴ The promise object will look something like

11.00 value: "Done"
State: rejected.

12.00 onfulfilment: [fulfilmenthandler1, fulfilmentHandler 2].

01.00 on rejection: [rejection handler1, rejection handler2].

02.00 promise object.

03.00 ∵ The state is "rejected". We will push handlers inside on rejection into the microtask queue.

Microtask queue will look something like:-

Rejection Handler1()	Rejection Handler2()	

04.00 microtask queue.

05.00 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()

callstack.

(6). "We rejected1 with value Done" get printed.

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

✉ @ ☎ Event loop will push the next handler inside callstack.

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

Rejection Handler2()

callstack.

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

09.00

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

10.00 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");
```

(8) Global piece of code:

09.00

"Start" gets printed.

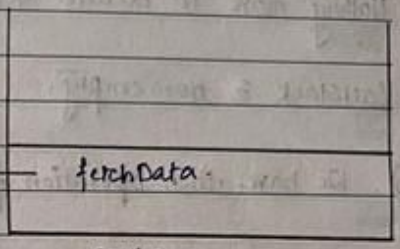
10.00

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

11.00

12.00

01.00



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

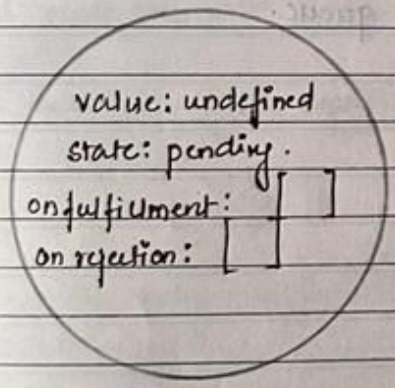
02.00

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



promise object.

05.00

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

06.00

07.00

(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.

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 it entry from the callstack.

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

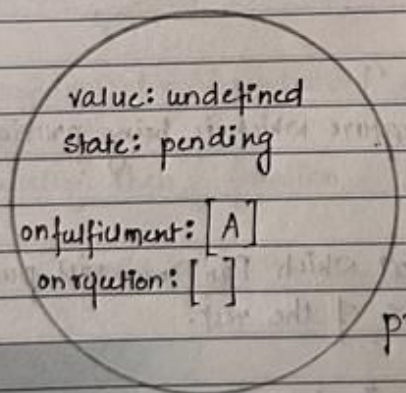
∵ The state is "pending". We will register all the handlers.

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

IMP

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

07.00 ∴ Our promise object will look like:-



promise object.

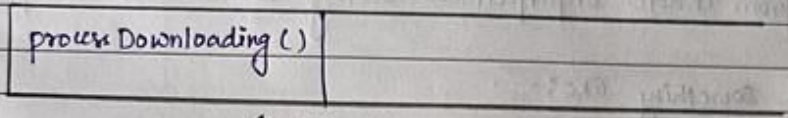
✉ @ ☎

(12). "End" gets printed.

At this point the runtime must have struck 7s.

∴ Runtime will push the callback. process Downloading in our Event Queue.

∴ Event Queue will look something like:-



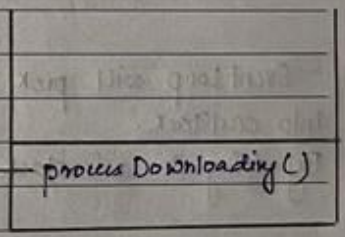
Event Queue.

At this point Event Loop checks that ⊙ Callstack is empty.
⊙ No global piece of code left.
⊙ microtask queue is empty.

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

∴ Event Queue gets empty.

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

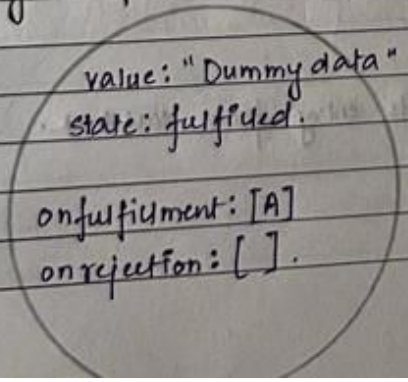


callstack.

5. Variable "data" gets value "Dummy data".

6. "Download Complete" gets printed.

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



promise object.

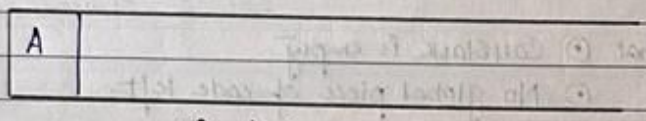
Nothing more to execute inside (9). We will remove the entry from the callstack.

Callstack gets empty.

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

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

The microtask queue will look something like:- .

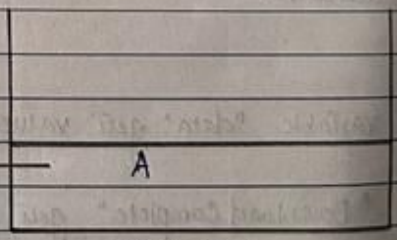


microtask queue.

Now, Event Loop found that ⊙ Callstack is empty
⊙ no global piece of code left.
⊙ But there are handlers inside microtask queue.

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

By doing so microtask queue gets empty.



callstack.

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

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

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

Callstack gets empty.

NOVEMBER 2017

Wk	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

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

(8). "Start" gets printed.

09.00

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

10.00

11.00

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

fetch Data

01.00

callstack.

(2). 02.00 A new promise object gets created in the memory.

03.00

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

promise object.

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

04.00

05.00

06.00

(3). "Started downloading from abcdc" gets printed.

07.00

(4). This piece of code is not Native to JS. This feature is provided to JS by Run Time. JS notifies Run Time to start a timer of 7s, post which Run Time 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

WK	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

42nd Week • 283-082

OCTOBER 10

09.00

(10). We encounter .then operator.

10.00

.then operator registers handlers inside our promise object.

11.00

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

12.00

So, our promise object looks like :- .

01.00

value: undefined
state: pending
onfulfillment : [A]
onrejection : [ ]

promise object.

02.00

03.00

04.00

(11). "End" gets printed.

06.00

At this point of time, 7s 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.

process Downloading ( )

callstack.

✉ @ ☎ Once this entry has been made we will go inside (9).

09.00 ⑤. Variable data gets assigned a value "dummy data" inside our callstack.

10.00 ⑥ We are resolving / fulfilling our promise.

11.00 ∴ The promise object will look like:-

12.00 value: "dummy data"
state: fulfilled.
onfulfilment: [A]
on rejection: [ ]

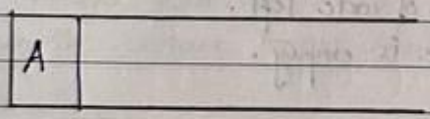
01.00 promise object.

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

03.00 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 on fulfilment array inside our microtask queue one by one.



microtask queue.

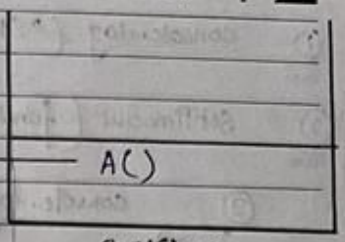
Event Loop will check the following :- ① . Callstack is empty.
② No global piece of code left.
③ Handler inside microtask queue.

✉ @ ☎

Event Loop will push the handler from microtaskqueue to callstack.

09.00

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



11.00

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

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 > Eventqueue.

04.00

```
function f() {
```

05.00

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

06.00

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

07.00

```
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

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

09.00 ① console.log("Start of the file");

10.00 ② setTimeout(function timer1(){

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

12.00

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

02.00 }

03.00 ⑤ let X = Promise.resolve("Subhadip");

04.00 ⑥ X.then(function processPromise(value){

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

06.00 ⑧ setTimeout(function timer2(){

07.00 ⑨ console.log("Timer 2 done");
},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 notifies RunTime to start a timer of 0s post which RunTime will push the callback in our Eventqueue and JS will take care of the rest.

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

(4). Even before starting this synchronous piece of code. RunTime has struck 0s. RunTime will push the callback timer1 in our Eventqueue.

Eventqueue will look something like:—.

timer1( )	

Event Queue.

∵ 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 fulfil handler.

∴ Promise object will look like.

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

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 onfulfilment 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 03.00 Runtime.

JS will notify Runtime to start a timer of 0s post which Runtime is going to push 04.00 the callback timer2() inside Event queue.

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

(10). 06.00 Before executing this global piece of code, Runtime has struck 0s.

07.00 Runtime will push the callback timer2() inside EventQueue.

timer1()	timer2()	

event queue.

"End of file" gets printed.

Event loop found out that ⊙ callstack is empty.

✉ @ ☎ ⊙ No global piece of code left.

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

Event Loop prioritised handler inside microtask queue.

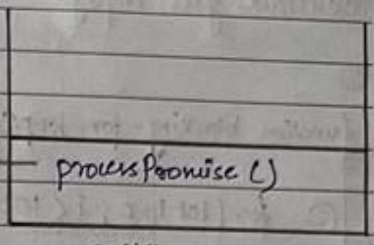
09.00 Event Loop pushed the handler from microtask queue to callstack.

10.00

11.00

12.00 Once this entry has been made we go inside (6).

01.00



(7). "Whose promise ? Subhadip" gets printed.

03.00 Nothing else to execute. The entry correspondig to this handler gets removed.

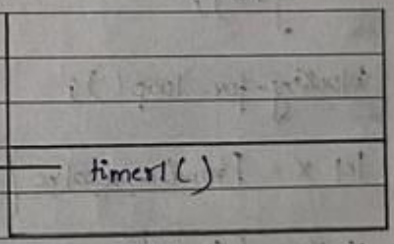
04.00 Callstack gets empty.

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 (2).

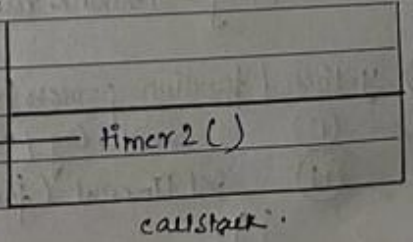


(3) "Timer 1 done" gets printed.

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 (8).



09.00

⑨. "Timer 2 done" gets printed.

10.00

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
};
③ console.log("Start of the file");
④ setTimeout(function timer1(){
05.00
    ⑤ console.log("Timer1 done");
},0);
06.00
⑥ blocking-for-loop();
07.00
⑦ let x = Promise.resolve("Subhadip's promise1");
⑧ x.then(function procesPromise1(v){
    ⑨ console.log("Whose promise?", v);
    ⑩ blocking-for-loop();
});
⑪ ✉@☎ let y = Promise.resolve("Subhadip's promise2");
⑫ y.then(function procesPromise2(v){
    ⑬ console.log("Whose promise?", v);
    ⑭ setTimeout(function(){console.log("ok done")},0);
});
⑮ let z = Promise.resolve("Subhadip's promise3");
⑯ z.then(function procesPromise3(v){
    ⑰ console.log("Whose promise", v);
});
⑱ setTimeout(function timer2(){
    ⑲ console.log("Timer2 done");
},0);
⑳ console.log("End of file");
```

09.00 (3) Global piece of code :— "Start of the file" gets printed.

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

11.00 JS notifies Runtime to start a timer of 0s, post which Runtime will push the callback timer1 in the Event queue.

12.00 JS comes back. In the background a timer of 0s has started off.

(5). At this point, the runtime has already struck 0s.

02.00 Runtime will push the callback timer1 in our ~~callback~~ Event queue.

Event queue will look something like :—

03.00

timer1 ( )	

04.00 event queue.

05.00 Current step is a function call. We will go inside (1).

06.00 (2). This piece of code is Native to JS. JS will block here until and unless the FOR loop

07.00 is done executing.
JS blocks here for 10s.

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

X = value: "Subhadip's Promise"
state: fulfilled.
onfulfillment: [ ]
onrejection: [ ].

promise object.

✉ @ ☎

(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 fulfil 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.

x =

value: "Subhadip's Promise1"
state: fulfilled.
onfulfillment: [processPromise1]
onrejection: [ ]

promise object.

processPromise()

microtask queue

03.00

04.00

05.00

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

06.00

y =

07.00

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 fulfil handler (∵ this is a single handler inside .then).

Our promise object will look something like:—

y = value: "Subhadip's Promise 2"
state: fulfilled
onfulfillment: [process Promise2]
on rejection: [ ]
promise object

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

process Promise( )	process Promise( )

microtask queue.

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

The promise object will look something like:—

SUNDAY 22

Z = value: "Subhadip's Promise 3"
state: fulfilled
onfulfillment: [ ]
on rejection: [ ].
promise object.

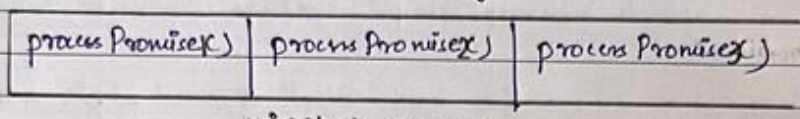
(16). ✉ @ ☎ We are using .then function.

.then registers handlers in our Promise object.

∵ Only 1 handler inside .then. It will be fulfil Handler.

value: "Subhadip's Promise 3"
state: fulfilled
onfulfillment: [process Promise3]
on rejection: [ ]
promise object.

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



microtask queue.

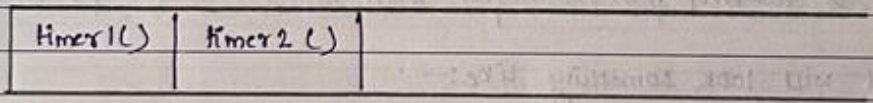
(18) This piece of code is not native to JS. This feature is being provided to JS by Runtime.

JS notifies Runtime to start a timer of 0s. Post which Run time is going to push the callback timer2() in the Event Queue.

JS comes back. In the background, a timer of 0s has started.

(20). At this step, the timer already struck 0s. Runtime is going to push the callback timer2 in the event queue.

Event queue will look something like:-



event queue.

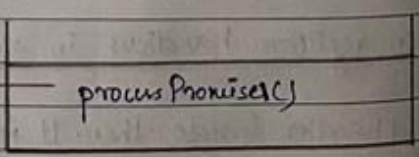
"End of file" gets printed.

Event Loop at this point found that ① callstack is empty.
② no global piece of code left.

In between microtask & 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 (8).

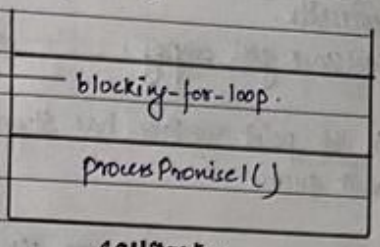


callstack.

(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).



(2). 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.

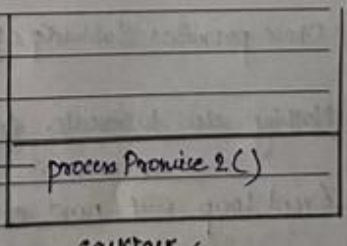
Nothing more to execute. So the entry corresponding to (1) gets popped off.

Nothing more to execute inside (2). The entry corresponding to processpromise1 gets popped off.

Callstack gets empty.

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

Once this entry has been made we will go inside (12).



(3) ✉ @ ☎ "Whose promise ? Subhadip's Promise2" gets printed.

(4). 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, past which Runtime will push the callback in the event queue.

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

Nothing more to execute inside. process promise 2 ( ). We remove it's entry from callstack.

Callstack gets empty.

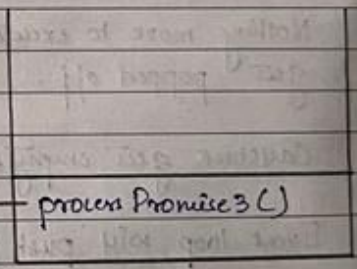
At this point runtime has struck 0s. Runtime will push the anonymous callback in the event queue.

The event queue will look like:-

timer1 ( )	timer2 ( )	f ( )

event queue.

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

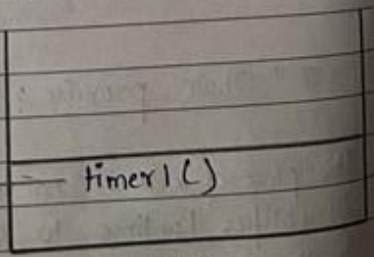


callback.

(17) "Whose 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.



Once this entry is done we go inside (4).

callstack.

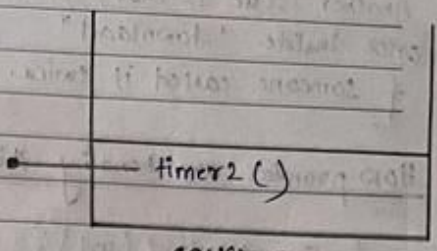
(5) "Timer1 done" gets printed. Callstack gets empty.

THURSDAY 2017

44th Week • 299-066

OCTOBER 26

Event Loop pushes the 2nd callback into callstack.



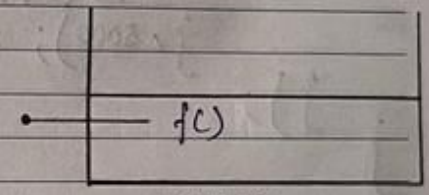
callstack.

We go inside (18).

(19). "Timer 2 done" gets printed.

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

Event Loop pushes the final anonymous callback in callstack.



callstack.

We go inside (14).

"OK done" gets printed.

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){
    Set Timeout (function exec(){
        const data = "dummy data";
        cb(data);
    },5000);
}
download ("xyz", function callback (data){
        console.log ("Data is," data);
});
```

I am giving control of my func. "callback" to another func. "download".
∵ I do not know "download"'s internal implementation.
I have no idea how my "callback" is being implemented inside it.

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

How promises are handling this

```
function download(url){
    return new Promise(function exec(res, rej){
        setTimeout(function(){
            let data = "dummy data";
            res(data);
            res(data);
        }, 5000);
    });
}

download("xyz").then(function fulfil(value){
    console.log("The value is", value);
});
```

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

"download" function is not having access to any callbacks

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

✉ @ ☎

NOVEMBER 2017

Wk	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";
            res(content);
        }, 5000);
    });
};
x = download("www.glee.com");
x.then(function fh1(value){
    console.log("Content downloaded is", value);
    return "new promise string";
}).then(function fh2(value){
    console.log("Value from chained then promise", value);
});
```

This code will return a new promise object.

This will be fed as an I/P to fh2.

- This is same as writing

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

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

SUNDAY 29

2017 MONDAY

30 45th Week • 303-062

OCTOBER

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

09.00
```
Promise.resolve("foo")
```
→ A

10.00
```
.then(function fn1(value){
```

11.00
```
    return new Promise(function exec(resolve, reject){
```

12.00
```
        setTimeout(function(){
```

01.00
```
            value += "bar";
```
→ B

02.00
```
            resolve(value);
```

03.00
```
        }, 10000);
```

04.00
```
    });
```

```
})
```

06.00
```
.then(function fn2(value){
```

07.00
```
    setTimeout(function(){
```

```
        value += "baz";
```
→ C

```
        console.log(value);
```

```
    }, 1);
```

```
    return value;
```

```
})
```

✉ @ ☎

```
.then(function fn3(value){
```
→ D.

```
    console.log(value);
```

```
});
```

NOVEMBER 2017

Wk	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

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

10.00 Let's call this (A).

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

01.00 ② . We are doing .then operation.

02.00 • then operation registers handlers in our promise object.

03.00

04.00

A = value: "foo"
state: fulfilled
onfulfil: [fn1]
onreject: [ ]
promise object

07.00 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
onfulfil: [ ]
on reject: [ ].

✉ @ ☎

NOVEMBER 2017

WK	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		

(7). On (B) we are doing .then.

∴ (B) will look something like

B =

value: undefined
state: pending
onfulfil: [fh2]
on reject: [ ]

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

It will look like:-

value: undefined
state: pending
onfulfil: [ ]
on reject: [ ].

(12). On (C) we are doing .then.

∴ (C) will look something like.

C =

value: undefined
state: pending
onfulfil: [fh3]
on reject: [ ]

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

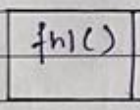
D = value: undefined
state: pending
onfulfil: [ ]
on reject: [ ].

Now,

From step (2) we came to know that (A) is already fulfilled promise.

& we are also done registering.

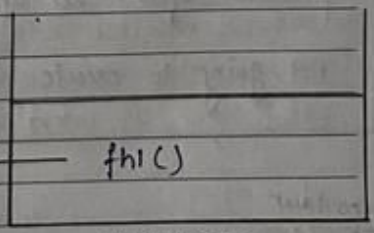
∴ JS will push the handler fh1 in microtask queue.



microtask queue.

Event Loop found out that nothing more to execute in call stack & no global piece of code left.

Event Loop pushes the handler from microtask queue to callstack.



callstack.

Once this entry is done we go inside (2).

③. It returns a new promise object.

So in the memory a new promise object gets created.

value: ~~undefined~~ undefined
state: pending
onfulfu: [ ]
on reject: [ ]

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

promise object.

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 fh1, we are returning a promise object. So unless and until the promise object resolves/reject we are not going to execute fh2, fh3.

Important

If we are returning a primitive value from fh1 the state of fh1 gets fulfilled. we would have started off with fh2 & fh3.

∵ [redacted]

[redacted]

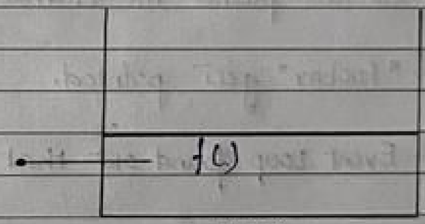
Nothing more to execute. Entry of thi gets removed.

callstack gets empty.

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

When timer struck 10s. Run Time will push the callback from runtime to Event queue.

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


callstack.

Once this entry is done we go inside (4).

(5) value gets updated to "foobar" → Note *
This value that we are updating is not the value from promise object. This value is referring to line (2).'s parameter.

(6) We are resolving the promise.

∴ Promise Object becomes :- .

value: "foobar"
state: fulfill
onfulfill: [ ]
on reject: [ ]

promise object.

Callstack gets empty.

NOVEMBER 2017

Wk	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			

(6). Once the promise resolves.

09.00

(7). The resolved promise's value "foobar" is used as a parameter in .then.

10.00

(8). This piece of code is not native to JS. Its a runtime based feature.

11.00

In the background a timer of 1ms started off.

12.00

(11). 01.00 We are saying return value inside .then.

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

(12). 03.00 At this point runtime has struck 1s.

04.00 Run time pushes the callback inside Event queue.

(13). 05.00 "Foobar" gets printed.

06.00 Event Loop found out that ⊙ callstack is empty
⊙ no global piece of code left.

07.00 ⊙ 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

Wk	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)

09.00

(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) {
(2)      console.log ("Starting to download from ", url); (a)
(3)      setTimeout ( function A ( ) {
(4)          console.log ("Download complete"); (b)
(5)          const data = "Dummy data";
(6)          callback1 (data);
         }, 4000);
07.00 }
(7)  function writeFile (content, callback2) {
(8)      console.log ("We are now saving your file..."); (d)
(9)      setTimeout ( function B ( ) {
(10)         console.log ("We are done saving your file"); (e)
(11)         const fileName = "doc.txt";
(12)         callback2 (fileName);
         }, 6000);
     }
```

NOVEMBER 2017

Wk	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			

```
(13) function upload (url, file, callback3) {
09.00
(14)     console.log("We are now saving your file", file, "at url", url); (g)
10.00
(15)     setTimeout( function() {
11.00
(16)         const message = "Success";
12.00
(17)         callback3(message);
01.00
     }, 3000);
02.00
}
03.00
(18) download("www.xyz.com", function cb1(content){
04.00
(19)     console.log("The download data is", content); (e)
05.00
(20)     writeFile(content, function cb2(fileName){
06.00
(21)         console.log("We have saved your file", fileName); (f)
07.00
(22)         upload("www.dropbox.com", fileName, function cb3(response){
(23)             console.log("The message is", response); (h)
     });
});
});
```

@ ☎