

Promise Constructor

Promises → value → default → undefined
→ state → default → pending

new Promise (function exec (res, ry) {

})

→ At the time when the constructor generates a new promise object, it also generates a pair of func, called as **resolve & reject**.

→ generally the executor callback, wraps some **async/sync** operation

→ the executor is called **sync**.

Consuming A Promise

→ assume this returns a promise.

```
let p = fetch( " " );
```



→ attach the
functionality that
we need to execute once the
promise is fulfilled or rejected

p. then (fulfillmenthandler, rejectionhandler)

p. then (— — —)
these are handler function, that we have to
implement ourselves.

value:

State:

On fulfillment: [f, g]

on rejection: [h, k]

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

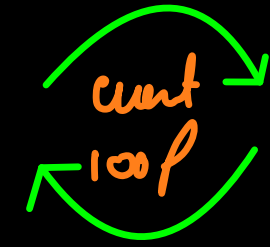
function createPromiseWithTimeout() {
  return new Promise(function executor(resolve, reject) {
    console.log("Entering the executor callback in the promise constructor");
    setTimeout(function () {
      let num = getRandomInt(10);
      if(num % 2 === 0) {
        // if the random number is even we fullfill
        resolve(num);
      } else {
        // if the random number is odd we reject
        reject(num);
      }
    }, 10000);
    console.log("Exiting the executor callback in the promise constructor");
  });
}

console.log("Starting....");
const p = createPromiseWithTimeout();
console.log("We are now waiting for the promise to complete");
console.log("Currently my promise object is like ... ", p);
p
.then(
  function fulfillHandler(value) {
    console.log("Inside fulfill handler with value", value);
    console.log("Promise after fulfillment is", p);
  },
  function rejectionHandler(value) {
    console.log("Inside rejection handler with value", value);
    console.log("Promise after rejection is", p);
  }
);
```

p →
②
③

④
⑤
⑥
⑦

~~under~~ fulfill call stack
~~under~~ 6
[] → on full ffh
on reject: [err]



Runtime
timer 1 → 10s

→ callback queue



microtask/job queue

→ At any point of time, if event-loop has a choice to pick from microtask queue or call back queue (macrotask queue) then it always gives preference to microtask queue.



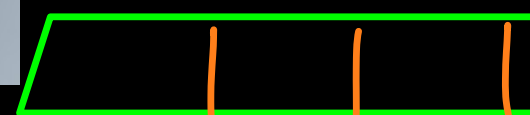
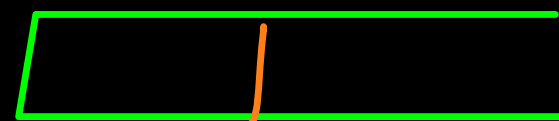
```
1 function createPromise() {  
2   return new Promise(function exec(resolve, reject) {  
3     console.log("Resolving the promise");  
4     resolve("Done");  
5   });  
6 }  
7  
8 setTimeout(function process() {  
9   console.log("Timer completed");  
10 }, 0);  
11  
12 let p = createPromise();  
13 p.then(function fulfillHandler1(value) {  
14   console.log("we fulfilled1 with a value", value);  
15 }, function rejectHandler() {});  
16 p.then(function fulfillHandler2(value) {  
17   console.log("we fulfilled2 with a value", value);  
18 }, function rejectHandler() {});  
19 p.then(function fulfillHandler3(value) {  
20   console.log("we fulfilled3 with a value", value);  
21 }, function rejectHandler() {});  
22  
23 for(let i = 0; i < 10000000000; i++) {}  
24  
25 console.log("ending");
```

fulfill
Done on fulfiller [fh1, fh2, fh3]

Event Qs



callback queue



microtask

process

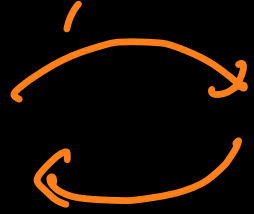


```
1 function createPromise() {
2   return new Promise(function exec(resolve, reject) {
3     setTimeout(function () {
4       console.log("rejecting the promise");
5       reject("Done");
6     }, 1000);
7   });
8 }
9
10
11 let p = createPromise();
12 p.then(function fulfillHandler1(value) {
13   console.log("we fulfilled1 with a value", value);
14 }, function rejectHandler(value) {
15   console.log("we reject1 with a value", value);
16 });
17 p.then(function fulfillHandler2(value) {
18   console.log("we fulfilled2 with a value", value);
19 }, function rejectHandler(value) {
20   console.log("we reject2 with a value", value);
21 });
22
23
24 for(let i = 0; i < 10000000000; i++) {}
25
26 console.log("ending");
```

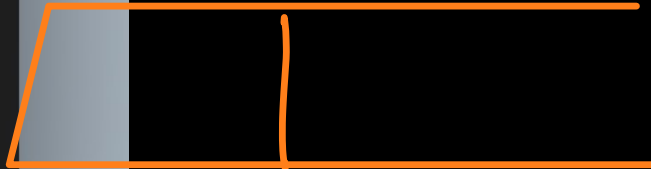
Timing IS

reject
already
undefined
Done

[fh1, fh2]

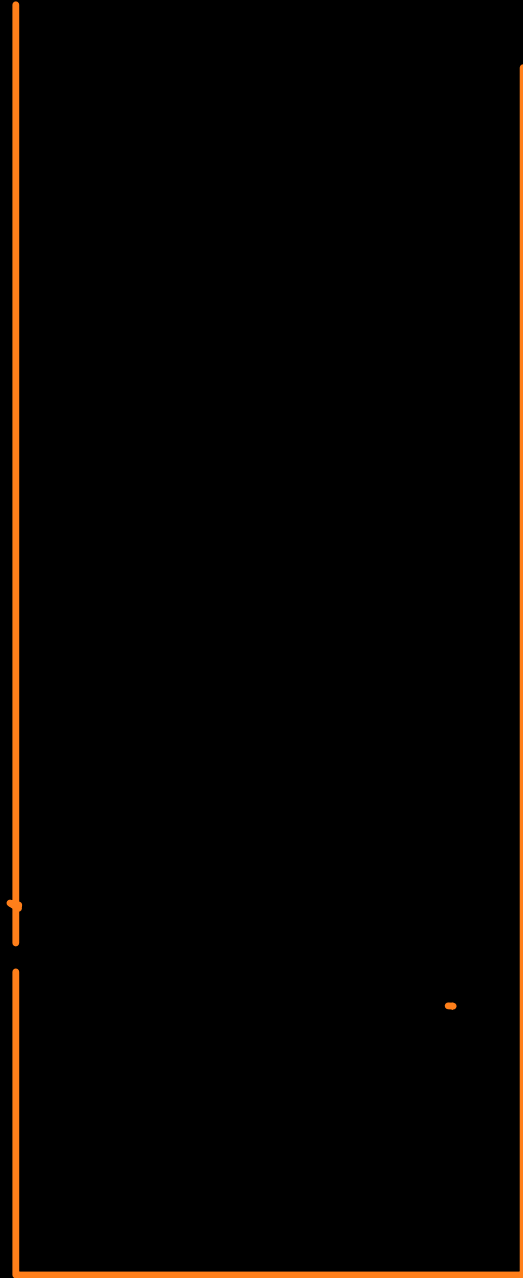


callback



[fh1 fh2]

→ milliseconds



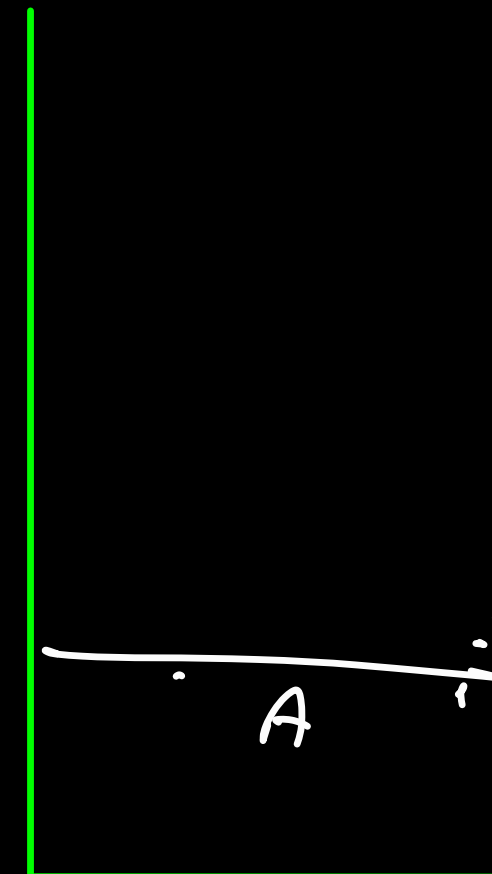


```
1 function fetchData(url) {  
2   return new Promise(function (resolve, reject) {  
3     console.log("Started downloading from", url);  
4     setTimeout(function processDownloading() {  
5       let data = "Dummy data";  
6       console.log("Download completed");  
7       resolve(data);  
8     }, 7000);  
9   });  
10 }  
11  
12 console.log("Start");  
13 let promiseObj = fetchData("skfbjdkjbv");  
14 promiseObj.then(function A(value) {  
15   console.log("value is", value);  
16 })  
17 console.log("end");
```

fulfilled



time → 7s
—



start
started downloading from —
end.
download complete
value is dummy data.

value: dummy data
state: fulfilled
on fulfill: [A]
on reject: []

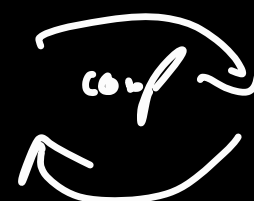


```
1 function fetchData(url) {  
2   return new Promise(function (resolve, reject) {  
3     console.log("Started downloading from", url);  
4     setTimeout(function processDownloading() {  
5       let data = "Dummy data";  
6       resolve(data);  
7       console.log("Download completed");  
8     }, 7000);  
9   });  
10 }  
11  
12 console.log("Start");  
13 let promiseObj = fetchData("skfbjkdjbv");  
14 promiseObj.then(function A(value) {  
15   console.log("value is", value);  
16 })  
17 console.log("end");
```

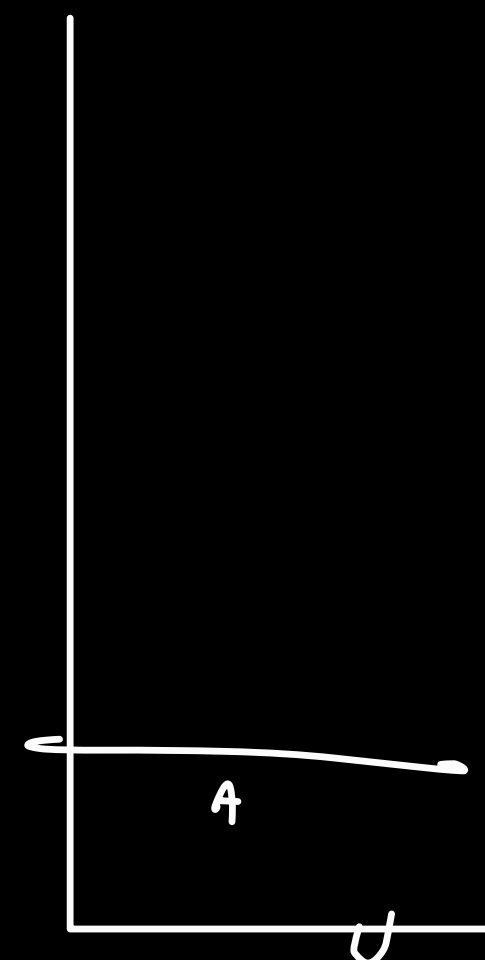
Callback



microtask



Run



Start
Should download from
end
double capture
value is dummy data

value: dummy data
State: fulfilled
onfulfill: []
onreject: []

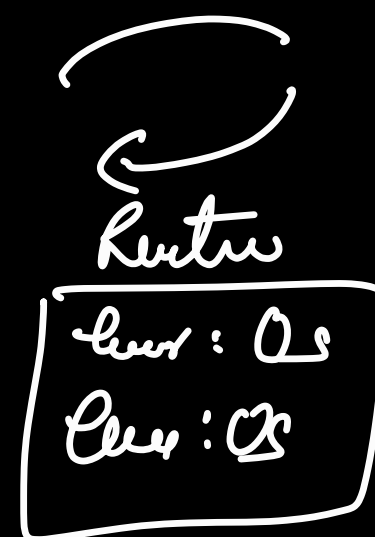
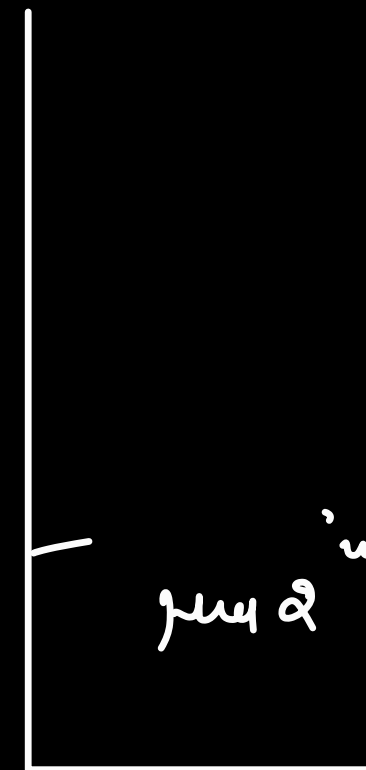
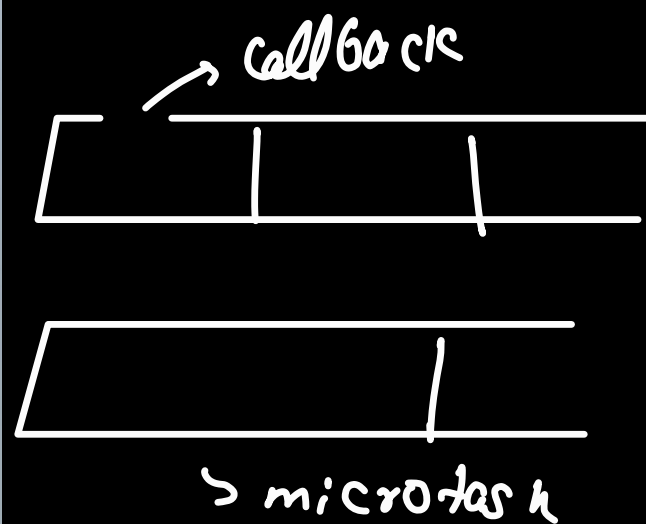
call stack > microtask > callback/
global code macrotask



```

1 console.log("Start of the file"); (1)
2
3 setTimeout(function timer1() {
4     console.log("Timer 1 done"); (4)
5 }, 0);
6
7 for(let i = 0; i < 10000000000; i++) {
8     // something 0
9 }
10
11 let x = Promise.resolve("Sanket's promise");
12 x.then(function processPromise(value) {
13     console.log("Whose promise ? ", value); (3)
14 });
15
16 setTimeout(function timer2() {
17     console.log("Timer 2 done"); (5)
18 }, 0);
19
20 console.log("End of the file"); (2)
21

```

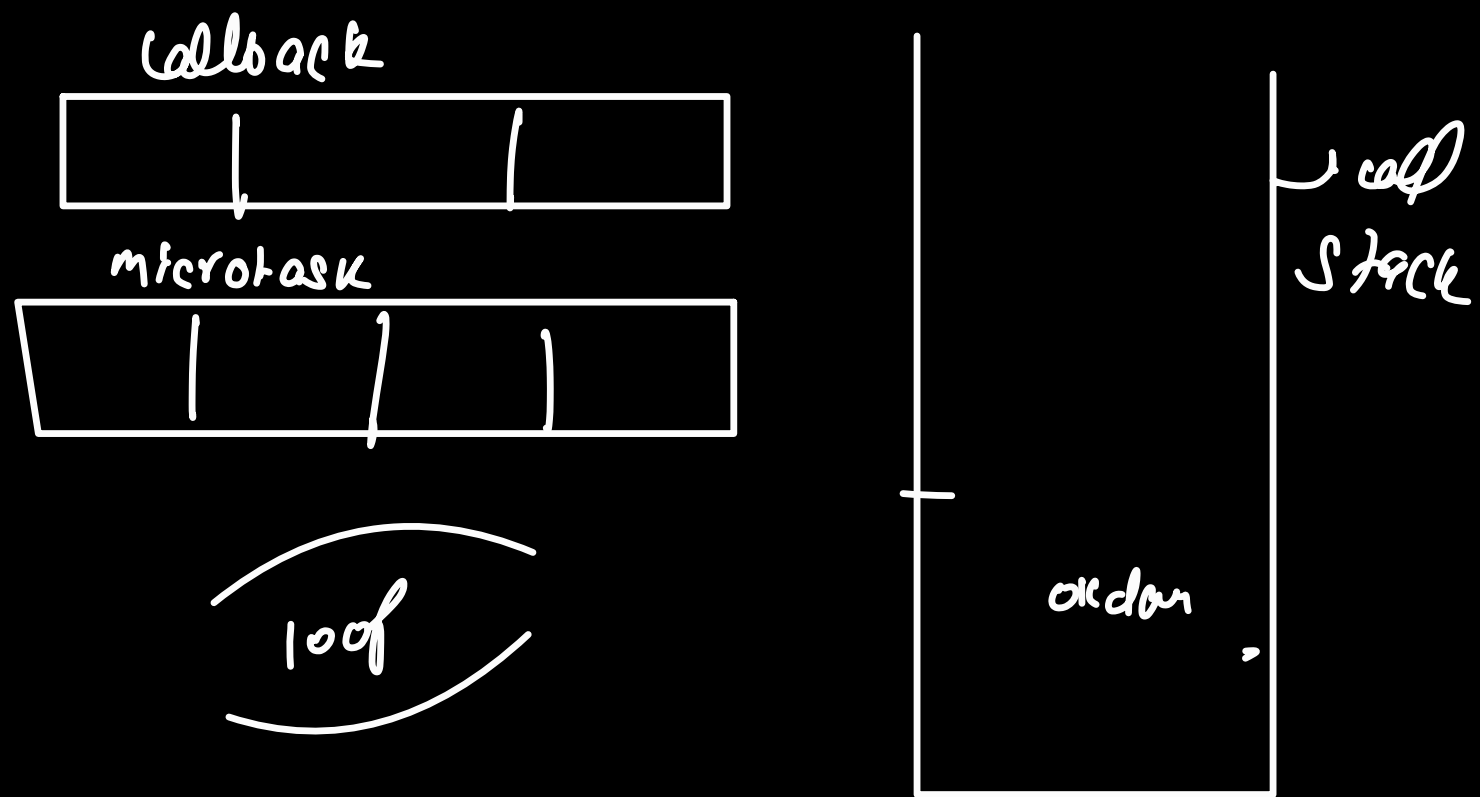


value: Sanket's promise
 state: fulfilled
 onfulfilled: [process promise]
 onrejected: []


```

1 function blocking_for_loop() {
2   for(let i = 0; i < 10000000000; i++) {
3     // something
4   }
5 }
6 console.log("Start of the file"); ①
7 → setTimeout(function timer1() {
8   console.log("Timer 1 done"); ⑥
9 }, 0);
10 → blocking_for_loop();
11 let x = Promise.resolve("Sanket's promise1");
12 x.then(function processPromise(value) {
13   console.log("Whose promise ? ", value); ③
14   blocking_for_loop();
15 });
16 let y = Promise.resolve("Sanket's promise2");
17 y.then(function processPromise(value) {
18   console.log("Whose promise ? ", value); ④ ⑤
19   setTimeout(function () {console.log("ok done")}, 0);
20 });
21 let z = Promise.resolve("Sanket's promise3");
22 z.then(function processPromise(value) {
23   console.log("Whose promise ? ", value); ⑤
24 });
25 → setTimeout(function timer2() {
26   console.log("Timer 2 done"); ⑦
27 }, 0);
28 console.log("End of the file"); ②
29

```



runtimes

Time 1: 0s

Time 2: 0s

Time 3: 0s

x → { value: Sanket's promise1
state: fulfilled
onFulfilled: [process promise]
onThen: [] }

1 → { value: Sanket's promise2
state: fulfilled
onFulfilled: [process promise]
onThen: [] }

2 → { value: Sanket's promise1
state: fulfilled
onFulfilled: [process promise]
onThen: [] }

the .then function returns a new promise object. It immediately
returns a new promise object