Sixth Pillar of JS

Promises:

- Readability Enhances
- Solve the problem of Inversion of control
- In JS, promises are special types of object that get returned immediately when we call them.

Inversion of control

```
function fun(x, cb){
    for(i=0, i<x, i++)
    {
    }
    cb();  // you don't this function is even called or
}          // it's called multiple times this is IOC.

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

- Promises acts as a placeholders for the data we hope to get back sometime in future.

→ runtime (JS will not wait for it to complete)

```
x = fetch("http://www.xyz.com")
```

↳ time consuming task

Now, assume fetch is written using promise then, it will immediately returns a promise obj which will act as a placeholder.

↳ for the result.

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

- Once the future task is done promise will automatically execute the attached functionality.

setTimeout(exec, 2000);

is it same as callbacks? Yes it's same but the difference is IOC. and attachment of the algo can be done later also.

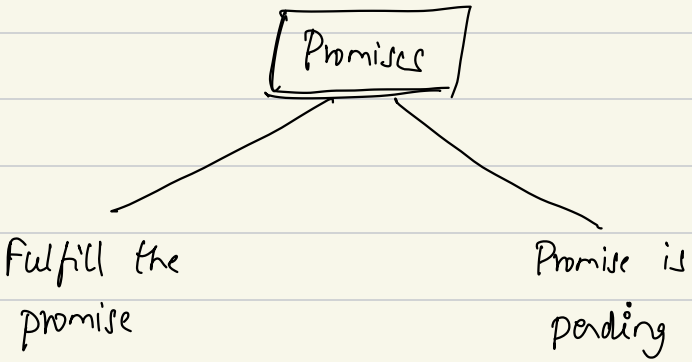
x = fetch("http://www.xyz.com")
↳ if return 'x'

fetch("http://www.xyz.com", function exec(){

});
↳ doesn't return anything and attachment of f() done immediately.

Important parts of learning promises:

1) How we can create a promise?
2) How can we consume a promise?

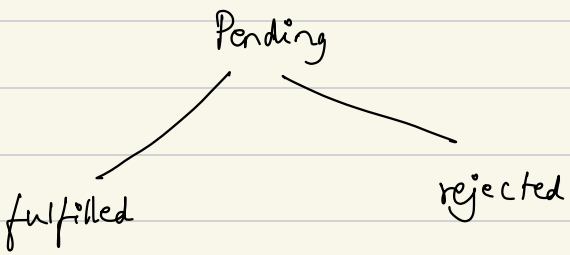


How to create a promise ?

— Creation of promise is sync in nature, but if inside a promise there is fetch() then that is async.

Three state of promise

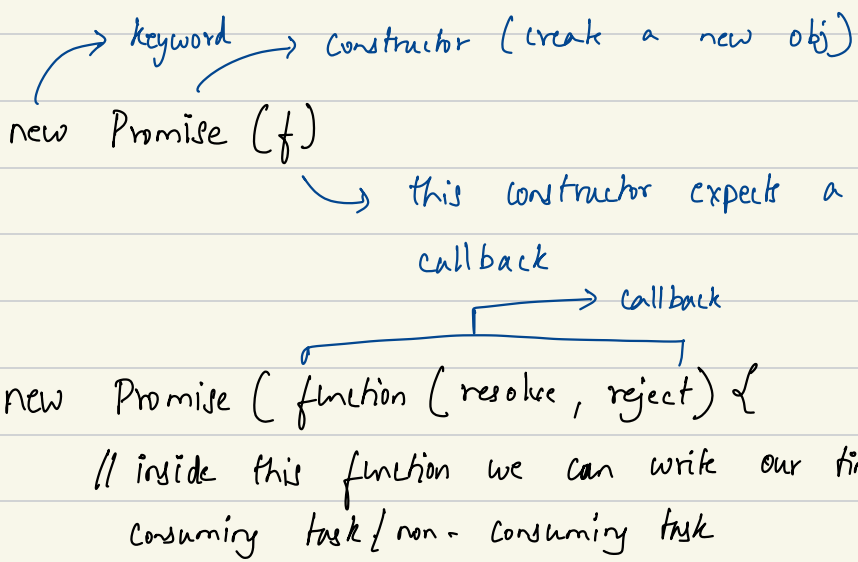
i) Pending : When we create a new promise object this is the default state.
It represent work in progress.



2) Fulfilled : If the operation is completed then state move from pending → fulfilled.

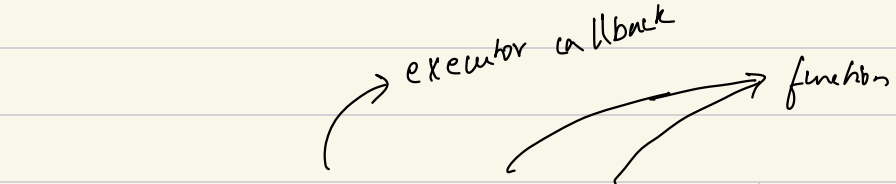
3) Rejected : If the operation is not successfully then state move from pending → rejected.

Syntax for new promise :



3)

Resolve & Reject call backs:



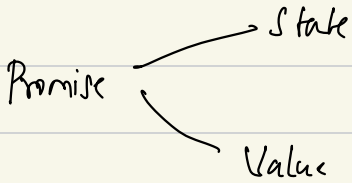
resolve(); // if you call this then the promise will go from pending → resolve state (fulfilled)

reject (); // if you call this then the promise will go from pending → rejected state

});

If you don't call anything then it remain in pending state.

Promise has two property state and value.



- Till the time state is pending, the value is undefined.

- if state is pending/rejected, the value can change.

resolve(x)

↳ whatever you pass in argument of resolve, the value will get assigned to the value property of the promise object

Same for the reject(x)

Coding:

```
> function  getRandomInt(max) {
    return Math.floor(Math.random() * max);
  }
< undefined
> getRandomInt(5);
< 1
> getRandomInt(5);
< 2
```

```
> function createPromise() {
    return new Promise(function executor(resolve, reject) {
        for(let i = 0;  i < 10000000000; i++){}
        let num = getRandomInt(10);
        if(num % 2 == 0){
            // if the random num is even then fulfill
            resolve(num);
        } else {
            // if the random num is odd then reject
            reject(num);
        }

    });
```

```
> x = createPromise();
  ▼ Promise {<fulfilled>: 2}
    ▶ [[Prototype]]: Promise
      [[PromiseState]]: "fulfilled"
      [[PromiseResult]]: 2
> x
```

→ even so fulfilled

```
> let x = createPromise();
  console.log(x)
  ▼ Promise {<fulfilled>: 6}
    ▶ [[Prototype]]: Promise
      [[PromiseState]]: "fulfilled"
      [[PromiseResult]]: 6
undefined
```

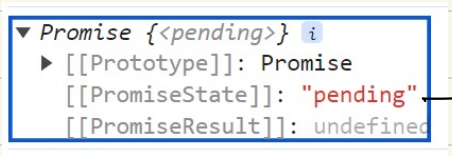
// Once you click enter it will take some time to print, because creation of promise is Sync in nature

Promise with setTimeOut

```
> function createPromisewithTimeout() {
    return new Promise(function executor(resolve, reject) {
      setTimeout(function () {
        let num = getRandomInt(10);
        if (num % 2 == 0) {
          // if the random num is even then fulfill
          resolve(num);
        } else {
          // if the random num is odd then reject
          reject(num);
        }
      }, 10000);
    });
  }
  let y = createPromisewithTimeout();
  console.log(y)
```

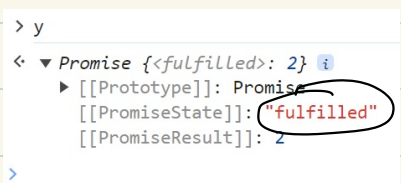
→ Once you click enter, it immediately create a promise with pending state because there is a setTimeout callback starts a timer of 10s in

a runtime as there is no blocking pieces of code so it immediately returns a promise.



→ The promise is pending

Now after 10 sec, the promise is fulfilled because it's a even (2).



If you pass multiple params in resolve / reject

for eg. resolve(num, 10, 20)

It will only take 1st argument.

Can we return from resolve/ reject ?

with return

```
> function createPromisewithTimeout() {
    return new Promise(function executor(resolve, reject) {
      setTimeout(function () {
        let num = getRandomInt(10);
        if (num % 2 == 0) {
          // if the random num is even then fulfill
          console.log("Fulfilling");
          return num;
        } else {
          // if the random num is odd then reject
          console.log("Rejecting");
          return num;
        }
      }, 5000);
    });
  }
  let y = createPromisewithTimeout();
  console.log(y);
```

```
▼ Promise {<pending>} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "pending"   → Pending
    [[PromiseResult]]: undefined
undefined
Fulfilling   → Fullfilling
```

Now after 5 sec you call y.

```
> y
▼ Promise {<pending>} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "pending"
    [[PromiseResult]]: undefined
>
```

The state of promise is still pending, because the state of promise will only change if you call resolve/ reject function

Now if you resolve and then return.

```
> function createPromisewithTimeout() {
    return new Promise(function executor(resolve, reject) {
      setTimeout(function () {
        let num = getRandomInt(10);
        if (num % 2 == 0) {
          console.log("Fulfilling");
          resolve(num);
          console.log("Completed Resolving");
          return num;
        } else {
          console.log("Rejecting");
          reject(num);
          console.log("Completed Rejecting");
          return num;
        }
      }, 5000);
    });
  }

  let y = createPromisewithTimeout();
  console.log(y);
```

```
▼ Promise {<pending>} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "pending"
    [[PromiseResult]]: undefined
undefined
Rejecting
Completed Rejecting
▶ Uncaught (in promise) 9
```

even after reject it goes to next line

after 5 sec

```
> y
▼ Promise {<rejected>: 9} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "rejected"
    [[PromiseResult]]: 9
```

→ Promise is rejected.

What if you resolve, reject two times?

```
function createPromisewithTimeout() {
  return new Promise(function executor(resolve, reject) {
    setTimeout(function () {
      let num = getRandomInt(10);
      if (num % 2 == 0) {
        console.log("Fulfilling");
        resolve(num);
        console.log("Completed Resolving");
        resolve(10);
        console.log("Resolving again");
        return num;
      } else {
        console.log("Rejecting");
        reject(num);
        console.log("Completed Rejecting");
        reject(11);
        console.log("Rejecting again");
        return num;
      }
    }, 5000);
  });
}

let y = createPromisewithTimeout();
console.log(y);
```

```
▼ Promise {<pending>} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "pending"
    [[PromiseResult]]: undefined
undefined
Rejecting
Completed Rejecting
Rejecting again
```

```
> y
▼ Promise {<rejected>: 3} i
  ▶ [[Prototype]]: Promise
    [[PromiseState]]: "rejected"
    [[PromiseResult]]: 3
>
```

// It's not 10,

that means once you resolve/reject your promise state you can't change/update the promise.

Follow mdn promise constructor

Consuming a promise :

assume this returns a promise.

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

In promise what to do when the promise is either fulfilled / reject that "to do" you can mention after some time. (OR) attach the functionality that we need to execute once the promise is fulfilled / rejected.

that was not the case with setTimeout after the timer completes it should executes the function.

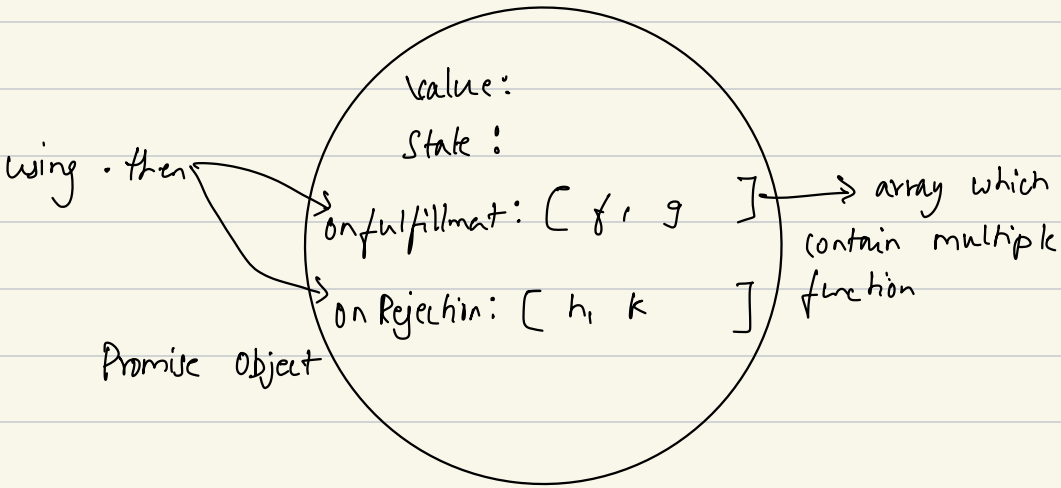
How can you attach the functionality ?

- We have .then( )

```
p.then(fulfillmentHandler, rejectionHandler)
```

this are handler function

- Inside the f(), that we have to implement ourselves.



Code

```
10-Sixth Pillar JS > 05.consumingPromise.js > rejectionHandler
1  function getRandomInt(max) {
2    return Math.floor(Math.random() * max);
3  }
4
5  function createPromisewithTimeout() {
6    return new Promise(function executor(resolve, reject) {
7      console.log("Entering the executor callback in the promise constructor");
8      setTimeout(function () {
9        let num = getRandomInt(10);
10       if (num % 2 == 0) {
11         resolve(num);
12       } else {
13         reject(num);
14       }
15     }, 10000);
16     console.log("Existing the executor callback in the promise constructor");
17   });
18 }
20 console.log("Starting....");
21 const p = createPromisewithTimeout();
22 console.log("We are now waiting for the promise to complete");
23 console.log("Currently my promise object is like ...", p);
24
25 p.then(
26   function fulfillHandler(value) {
27     console.log("Inside the fulfill handler with value", value);
28     console.log("Promise after the fullfillment is", p);
29   },
30   function rejectionHandler(value) {
31     console.log("Inside the rejection handler with value", value);
32     console.log("Promise after the rejection is", p);
33   }
34 );
```

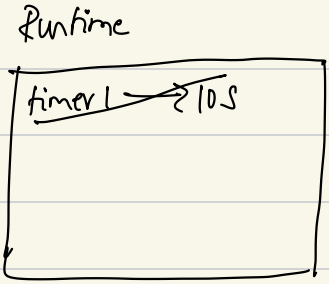
Output

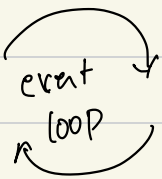
```
Starting....
Entering the executor callback in the promise constructor
Existing the executor callback in the promise constructor
We are now waiting for the promise to complete
Currently my promise object is like ... Promise { <pending> }
Inside the fulfill handler with value 4
Promise after the fullfillment is Promise { 4 }
```

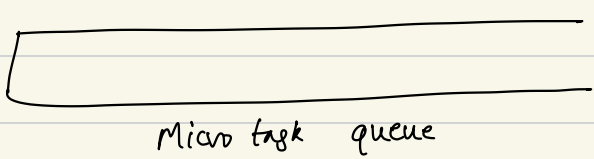
Output

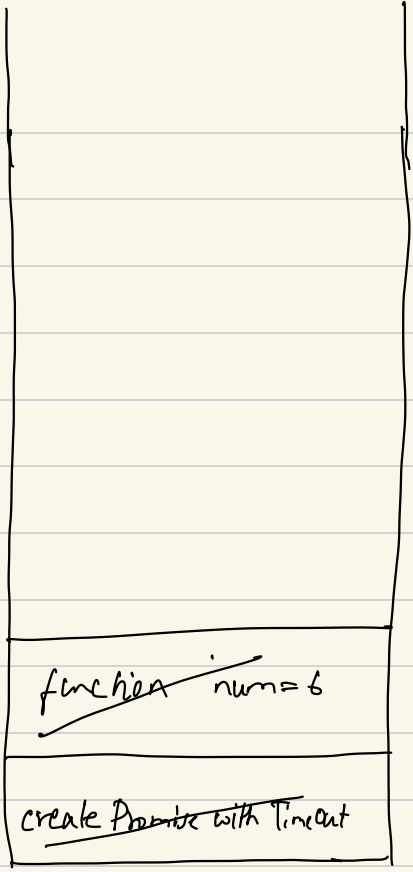
```
Starting....
Entering the executor callback in the promise constructor
Existing the executor callback in the promise constructor
We are now waiting for the promise to complete
Currently my promise object is like ... Promise { <pending> }
Inside the rejection handler with value 5
Promise after the rejection is Promise { <rejected> 5 }
```

Dry Run









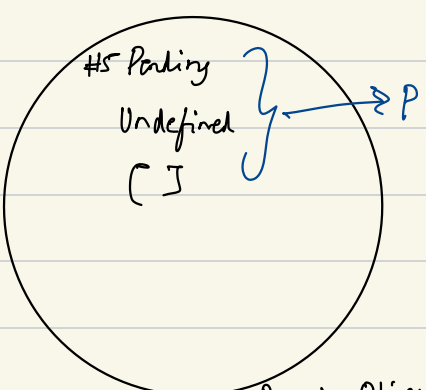
Output

#20 Starting

#21 → #5

#7 Entering the executor callback in the promise constructor

#8 SetTimeout → runtime



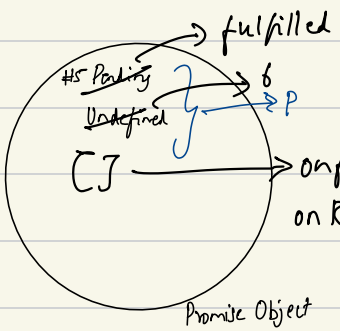
#6 Existing the executor callback in the promise constructor

#22 We are now waiting for the promise to complete

#23 Currently my promise object is like Promise <Pending>

Now p.then register handler for the promise, you are just registering NOT executing, it will execute only when the promise state changes

#25



when you do .then it got register in array.

-ẋ- Registering of promises happen in execution phase.

End of line and the final is done, so it send the callback to callback queue meanwhile event loop check is there anything in call stack answer is NO, is there any global pieces of code NO, so now so the function #1 go to call stack

Now the function does is generate Random number let's say the num is 8.

The num is even, so you resolve it #11

Promise object changes from pending → fulfillment and the value undefined → 6 (check the diagram)

The moment you resolve it, the onFullmat array has some fCl register that is going to execute. It will start executing with fulfillment Handler (6) #26

#27 Inside fulfill handler with value : 6

#28 Promise after fullfilment is {6}

You can have multiple fulfilment and rejection handler

```
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) {
        resolve(num);
      } else {
        reject(num);
      }
    }, 10000);
    console.log("Existing 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);
console.log("Going to register my 1st set of handlers");

p.then(
  function fulfillHandler1(value) {
    console.log("Inside the fulfill handler 1 with value", value);
    console.log("Promise after the fullfillment is", p);
  },
  function rejectionHandler1(value) {
    console.log("Inside the rejection handler 1 with value", value);
    console.log("Promise after the rejection is", p);
  }
);

console.log("Going to register my 2nd set of handlers");
p.then(
  function fulfillHandler2(value) {
    console.log("Inside the fulfill handler 2 with value", value);
    console.log("Promise after the fullfillment is", p);
  },
  function rejectionHandler2(value) {
    console.log("Inside the rejection handler 2 with value", value);
    console.log("Promise after the rejection is", p);
  }
);

console.log("Ending");

for (let i = 0; i < 100000000; i++) {}
console.log("Ending the loop also");
```

```
Starting....
Entering the executor callback in the promise constructor
Existing the executor callback in the promise constructor
We are now waiting for the promise to complete
Currently my promise object is like ... Promise { <pending> }
Going to register my 1st set of handlers
Going to register my 2nd set of handlers
Ending
Ending the loop also
Inside the rejection handler 1 with value 7
Promise after the rejection is Promise { <rejected> 7 }
Inside the rejection handler 2 with value 7
Promise after the rejection is Promise { <rejected> 7 }
```

Who executes this callbacks in onfullment and onRejection

→ When the promise is actually completed this function (fH, rH) goes inside microtask queue.

- Microtask queue is given more piority than callback queue.

```
function createPromise() {
  return new Promise(function exec(resolve, reject) {
    console.log("Resolving the promise");
    resolve("Done");
  });
}

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

let p = createPromise();
p.then(
  function fulfillHandler(value) {
    console.log("We fulfilled with a value", value);
  },
  function rejectHandler() {}
);
console.log("Ending");
```

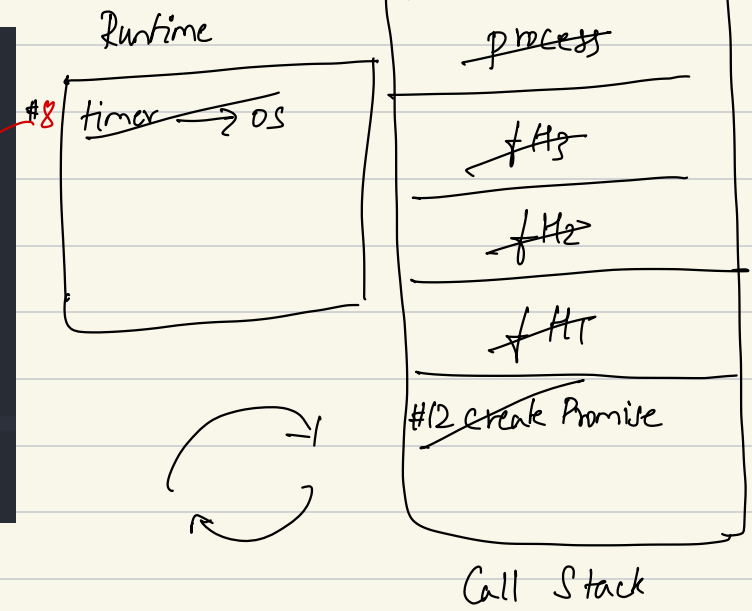
```
Resolving the promise
Ending
We fulfilled with a value Done
Timer Completed
```

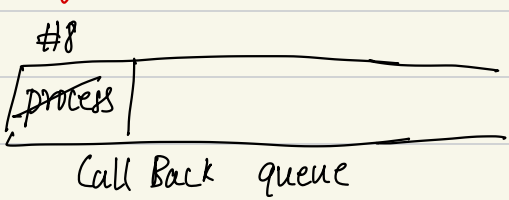
→ microtask

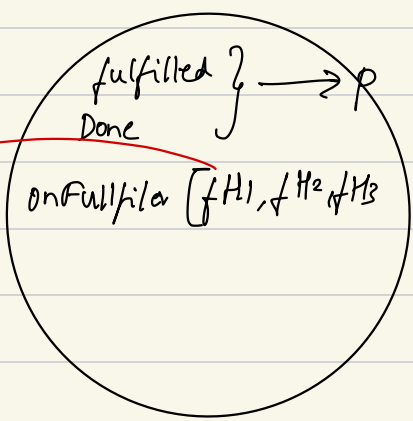
→ call back queue

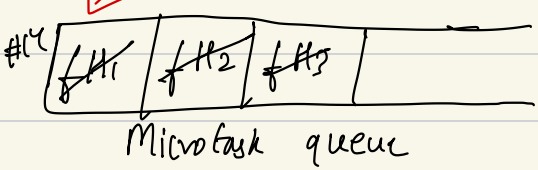
```
function createPromise() {
  return new Promise(function exec(resolve, reject) {
    console.log("Resolving the promise");
    resolve("Done");
  });
}

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

let p = createPromise();
p.then(
  function fulfillHandler1(value) {
    console.log("We fulfilled 1 with a value", value);
  },
  function rejectHandler1() {}
);
p.then(
  function fulfillHandler2(value) {
    console.log("We fulfilled 2 with a value", value);
  },
  function rejectHandler2() {}
);
p.then(
  function fulfillHandler3(value) {
    console.log("We fulfilled 3 with a value", value);
  },
  function rejectHandler3() {}
);

for (let i = 0; i < 10000000000; i++) {}
console.log("Ending");
```

fH1
fH2
fH3









#8 → Settimeout of timer os in Runtime

#12 → Creak Promise

#1 → Promise is fulfilled with a value Done

#13 → Register the fulfill Handler 1

as the promise is fulfilled so #14 goes to the microtask queue & wait

Same for fn2, fn3

Event loop check if it can execute microtask queue or not, ans is NO because the global piece of code is still left.

#32

#33 Ending.

Now event loop give preferance to microtask queue so it move fn1 to call stack

#15 we fulfilled with a value done

#21, 27 Same for fn2, fn3

Now event loop check is microtask queue is empty or not, it's empty now it move to the Callback queue.

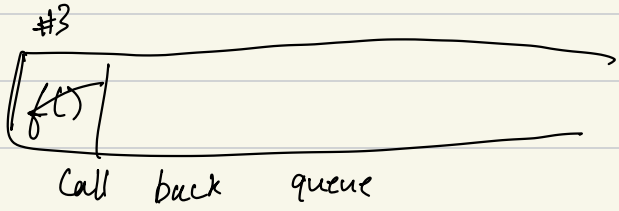
#8

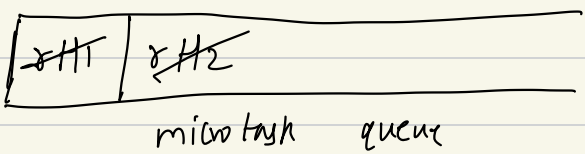
#9 timer completed

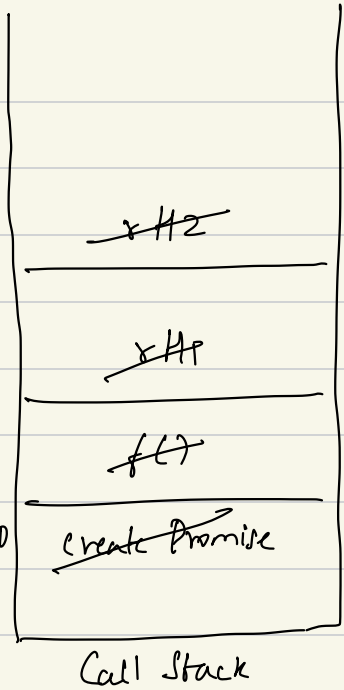
```
Resolving the promise
Ending
We fulfilled 1 with a value Done
We fulfilled 2 with a value Done
We fulfilled 3 with a value Done
Timer Completed
```

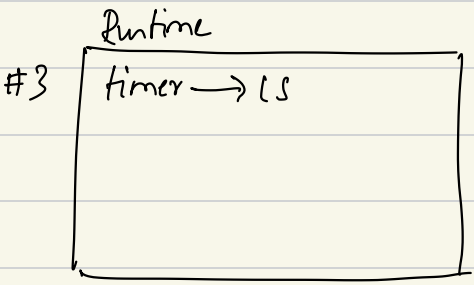
```
function createPromise() {
  return new Promise(function exec(resolve, reject) {
    setTimeout(function () {
      console.log("Rejecting the promise");
      reject("Done");
    }, 1000);
  });
}

let p = createPromise();
p.then(
  function fulfillHandler1(value) {
    console.log("We fulfilled 1 with a value", value);
  },
  function rejectHandler1(value) {
    console.log("We rejected 1 with a value", value);
  }
);
p.then(
  function fulfillHandler2(value) {
    console.log("We fulfilled 2 with a value", value);
  },
  function rejectHandler2(value) {
    console.log("We rejected 2 with a value", value);
  }
);

for (let i = 0; i < 10000000000; i++) {}
console.log("Ending");
```

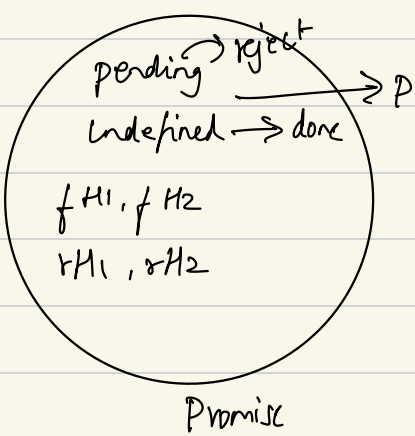
```
Ending
Rejecting the promise
We rejected 1 with a value Done
We rejected 2 with a value Done
```











Output

#10 → callstack

#3 → exec() timer and return a new Promise object

#11 → Pending promise register two fulfill Handler and two reject

#28 while for loop complete meanline timer completes foo, so f() goes to call back #3

#29 Ending

Nothing in microtask queue so f() get execute from call back

#4 Rejecting the promise

#5 Promise → reject value → done

as it reject, so rH1, rH2 goes to the micro task queue

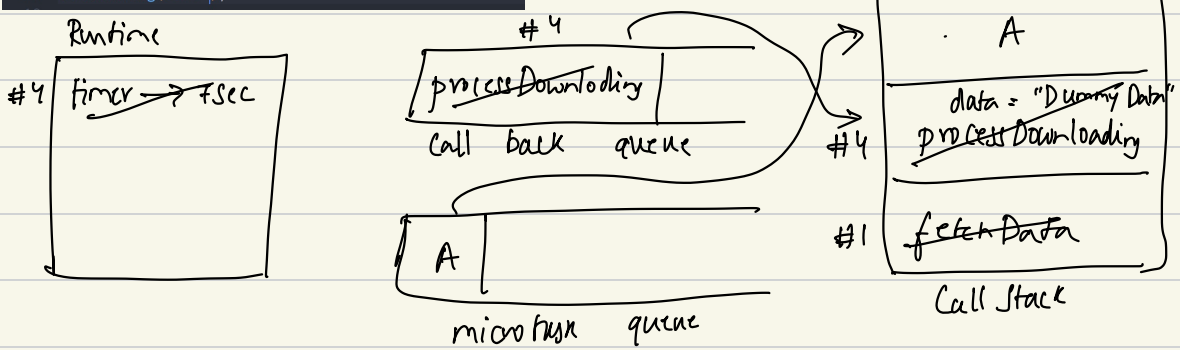
so nothing in global so it get execute in callstack

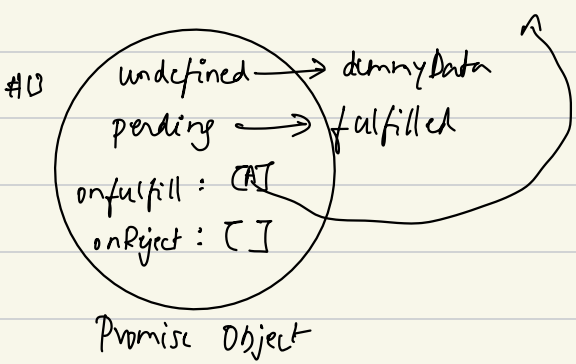
#16 we reject 1 with a value done

#24 we rejected 2 with a value done.

```
10-Sixth Pillar JS > 12.microtaskQueue04.js > ...
1  function fetchData(url) {
2    return new Promise(function (resolve, reject) {
3      console.log("Started downloading", 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("hello");
14 promiseObj.then(function A(value) {
15   console.log("value is ", value);
16 });
17 console.log("End");
```

fulfillment






```
[Running] node "e:\Backend-Dev
Start
Started downloading hello
End
Download completed
value is  Dummy data
```

#12 Start

#13 → #1

#2 New Promise Object

#4 Settimeout → runtime

#13 created promise object

#14 • then register, you see only one callback then it's only fulfilment. (A)

#17 End

Timer get over and moved to callback queue and to call stack

#6 Download complete

#7 resolve is called so now value is dummy data and it's fulfilled

once it's fulfilled the function A which is in onfulfillment array is moved to microtask queue then moved to call stack

#14 ⟶ #15 value is dummy Data

One more e.g.

```
10-Sixth Pillar JS > 13.microtaskQueue05.js > fetchData > <function> > processDownloading
1  function fetchData(url) {
2    return new Promise(function (resolve, reject) {
3      console.log("Started downloading", 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("hello");
14 promiseObj.then(function A(value) {
15   console.log("value is ", value);
16 });
17 console.log("End");
```

```
Start
Started downloading hello
End
Download completed
value is  Dummy data
```

Priority Order :

call stack > micro task > callback

↓

global code

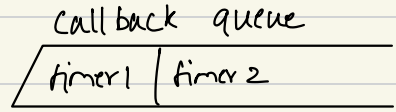
Using promise dot resolve :

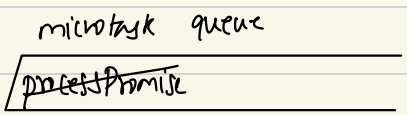
```
1  console.log("Start of the file");
2
3  setTimeout(function timer1() {
4    console.log("Timer 1 done");
5  }, 0);
6
7  for (let i = 0; i < 1000000000; i++) {
8    // something
9  }
10
11 let x = Promise.resolve("Rizon Promise");
12 x.then(function processPromise(value) {
13   console.log("Whose promise ? ", value);
14 });
15
16 setTimeout(function timer2() {
17   console.log("Timer 2 done ");
18 }, 0);
19
20 console.log("End of the file");
```

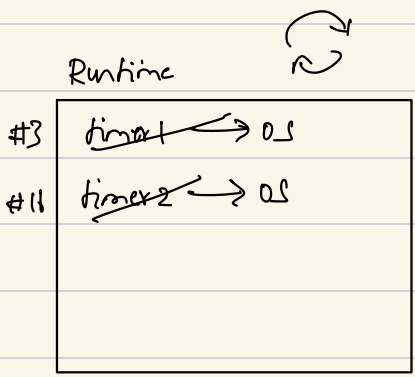
→ already resolved promise

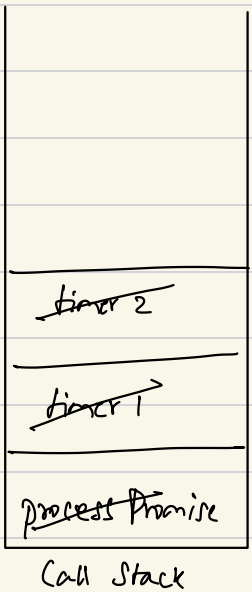
```
function f() {
  return new Promise(function exe(res, rej) {
    res("Rizon Promise")
  });
}
```

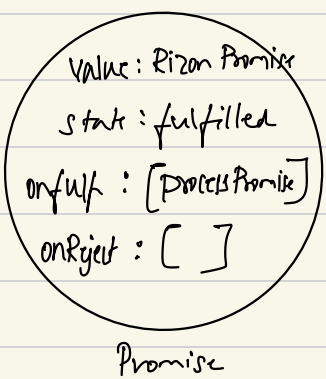
```
1  console.log("Start of the file");
2
3  setTimeout(function timer1() {
4    console.log("Timer 1 done");
5  }, 0);
6
7  for (let i = 0; i < 1000000000; i++) {
8    // something
9  }
10
11 let x = Promise.resolve("Rizon Promise");
12 x.then(function processPromise(value) {
13   console.log("Whose promise ? ", value);
14 });
15
16 setTimeout(function timer2() {
17   console.log("Timer 2 done ");
18 }, 0);
19
20 console.log("End of the file");
```












```
[Running] node "e:\Backend-Deve
Start of the file
End of the file
Whose promise ?  Rizon Promise
Timer 1 done
Timer 2 done
```

How promise resolves issues of callbacks?

```
function download(url, cb) {
  console.log("Started downloading from url", url);
  setTimeout(function exec() {
    console.log("Completed downloading after 5sec", url);
    const content = "ABCDEF";
    cb(content);
    // cb(content); // two times cb is called IOC issue
  }, 5000);
}

download("www.zyx.com", function processDownload(data) {
  console.log("download data is", data);
});
```

Callback code

Promise Code

```
function download(url) {
  console.log("started downloadin from ", url);
  return new Promise(function exec(res, rej) {
    setTimeout(function p() {
      console.log("Completed downloading data in 5 sec");
      const content = "ABCDF";
      res(content);
    }, 5000);
  });
}

download("www.xyx.com").then(function fulFillHandler(value) {
  console.log("Content download is", value);
});
```

res(content) // it won't execute 2nd time

- You have access to your callback, not giving access to 3rd party function.

- You yourself responsible for calling the callback
- Whole control is with you.

IOC is handled here.

.then chaining:

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

```
function download(url) {
  console.log("started downloadin from ", url);
  return new Promise(function exec(res, rej) {
    setTimeout(function p() {
      console.log("Completed downloading data in 5 sec");
      const content = "ABCDF";
      res(content);
    }, 5000);
  });
}

x = download("www.xyx.com");
x.then(
  function fulFillHandler(value) {
    console.log("Content download is ", value);
    return "New Promise String";
  },
  function rejectHandler(value) {
    console.log("rejected with ", value);
  }
).then(function newFullFillHandler(value) {
  console.log("value from chained then promise", value);
});
```

You did return.

```
[Running] node "e:\Backend-Development\10-Sixth Pillar JS\17.dotthenChaining.js"
started downloadin from  www.xyx.com
Completed downloading data in 5 sec
Content download is  ABCDF
value from chained then promise New Promise String
```

- If you return then you can do chaining .then with a value.
- If you don't return then the value is undefined.