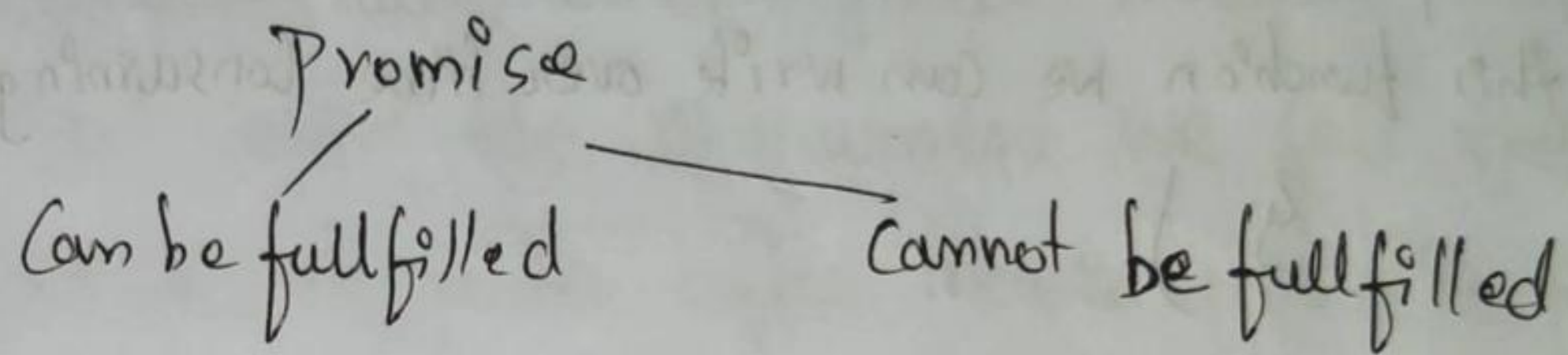


- 1) How we can create a Promise?
- 2) How can we consume a Promise?

Promise (→ generally it means suppose for a example we give the promise that we will do lot of hard work and I promise that I will make the extra effort, so the person will take the promise, but but promises can break so there might be promises can be fulfilled and promises cannot be fulfilled).



1) How to create a Promise?

→ Promises are **NATIVE** to javascript because we can see about the promise in ECMA docs.

So creation of a Promise object is **SYNCHRONOUS** in nature, ONLY creation is synchronous.

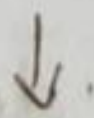
There are Three States in Promises

i) Pending



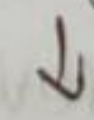
When we create a new Promise object this is a default state it represents WORK IN PROGRESS

ii) fulfilled



If the operation is completed **STATE** successfully then only this state is achieved

iii) Rejected



If the operation is not successful it will achieve the Rejected state.

new Promise(f) → this constructor expects the Callback

new Promise (function () {

→ this call back function
that we are passing takes
two parameters

resolve, reject) {

// inside this function we can write over time consuming task

resolve, reject these two are actually the functions only

* If the function resolve() is called inside then
the Promise go to the fullfilled state

* If the function reject() is called inside then
the promise will go to the rejected state.

or
* If we don't call resolve() or reject() the
Promise will forever stay in pending state

this was all about **STATE** properly

Now let's talk about **VALUE** property.

* Till the time the state is at the pending state, the value will be **UNDEFINED** (pending state \rightarrow value: Undefined)

* The moment we ^{go} to the fullfilled state or Rejected state then this value will change

Suppose if we are calling the resolve function with same value x , and that value can be string, number, boolean, null, ~~an~~ undefined; doesn't matter

what, so whatever the argument we call the resolve function with after moving to the fullfilled state the value of property gets updated with

the argument of resolve

Same applies to Reject-function too

For Example:

```
function getRandomInt(max) {
```

```
  return Math.floor(Math.random() * max)
```

```
};
```

```
function createPromiseWithLoop() {
```

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

```
    for (let i = 0; i < 1000000; i++) {
```

```
      let num = getRandomInt(10)
```

```
      if (num % 2 === 0) {
```

```
        resolve(num) // if the random number is even  
                      we fulfill
```

```
      } else {
```

```
        reject(num) // if the random number is odd  
                    we reject
```

```
      }
```

```
    }
```

```
  let x = createPromiseWithLoop()
```

```
  console.log(x)
```

Currently in above example the for loop is running
So after the running of for loop at the moment
we will get the o/p.

if the random number was even \Rightarrow Promise { 4 }

if the random number was odd \rightarrow promise { <rejected> 3 }
and here it initially won't print anything because
the for loop is taking time, once the for loop
execution is done we will get the o/p, so we can
say whenever we create a promise they are synchronous
in nature so let us better understand with one more
example

Let's try with ~~ed~~ Timeout instead of Loop

Example 2 :

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

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

}

\downarrow 10 secs

let y = create Promise With Time Out ()

2 console.log (y)

o/p → Promise {<pending>}

As we can see this time it immediately gave us pending state, as we had written return new Promise so the new Promise is expected to return, then there is call back function that function will be immediately called, and this function says setTimeout, the moment javascript sees setTimeout it will go to the runtime and say hey runtime start the timer of 10 secs, and that's it after that it immediately prints Promise {<pending>} because Promise is SYNCHRONOUS and NATIVE to javascript and ~~after the 10 secs gets completed~~ and in last example the for loop was the blocking piece of code ~~so it would~~ and as we know for loop is Native to javascript so after the completion of for loop we would get the o/p.

Here after completion of 10 secs, the setTimeout() is completed so the callback of setTimeout() will be called. and it creates a random number and then that number if it was even output would be Promise {4} if it was odd output would be Promise {<rejected> 3}

Example 3

function createPromi

suppose in previous example we had written

resolve(num) and if the num was even we would get the o/p promise { 4 }

What if we pass multiple arguments into resolve for example resolve(num, 10, 20) and o/p remains the same if we get num as even o/p will

be. promise { 8 } hence those arguments won't affect or we can say they are not considered SAME with reject too

Example 3 :

What happens if we don't write `resolve()` and `reject()`?

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

```
      setTimeout (function () {  
        console.log("fulfilling")  
        return num  
      }, 10000)  
    }  
  })  
}
```

```
setTimeout (function () {  
  let num = get Random (10)  
  if (num % 2 == 0) {
```

```
    console.log("fulfilling")  
    return num  
  }  
}
```

```
else {  
  console.log("rejecting")  
  return num  
}  
}
```

```
}, 10000)  
})  
{
```

}

→ gives the random number
have not written the function of get Random refer it for previous examples


```
let y = createPromiseWithTimeOut(),  
console.log(y)
```

%p → starting it will ~~print~~ print promise {<pending>}
after completion of 10 secs, it prints fulfilling

%p → fulfilling, and the state of the
Promise will be still in **Pending**, because
the state of Promise will only change if we call
resolve() or reject() function. if you don't
call it will never change the state.

Example 4 → what if we write something after resolve() and reject().

```
setTimeout(function() {  
  let num = getRandom(10)  
  if (num % 2 == 0) {  
    resolve(num)  
    console.log("completed")  
  }  
  else {  
    reject(num)  
    console.log("rejected")  
  }  
, 10000)
```

%p → if the num was even the %p would be
promise { 6 }
completed. > so whatever we write
below resolve that also
will be executed

Suppose if the num was odd the o/p would be
promise { checked > 3 }
rejected

Nothing but we can say it works as synchronous code.

Example 3 → What if we do resolve() or reject()
two times

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

```
  let num = get Random Int (10)
```

```
  if (num % 2 == 0) {
```

```
    console.log
```

```
    resolve(num)
```

```
    console.log("completed resolving")
```

```
    resolve(10)
```

```
    console.log("resolving again")
```

```
  }
```

```
  else {
```

```
    re reject(num)
```

```
    console.log("completed rejecting")
```

```
    reject(10)
```

```
    console.log("resolving again")
```

```
  }, 10000)
```

```
}
```



```
let y = createPromiseWithTimeout (5);  
console.log(y).
```

o/p → Initially we will be having Promise { < pending > }
after 10 secs

o/p → ~~is~~ completed resolving
resolving again.

BUT Let's see what is in the Promise.

in the Promise we have

Promise { < fulfilled > : 6 }.

which was from the random
generated num. it is not

→ 10 (which was of one more resolve)

Note → That means ~~once~~ Once we Resolve (). or

Reject (), the it can **NEVER** be updated.