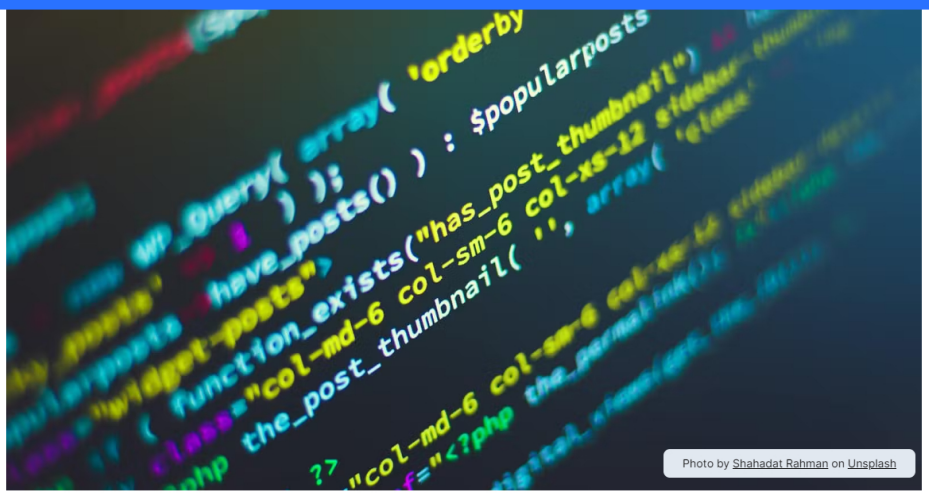


The only blog you need to learn Promises In JS

Sanket Singh · May 11, 2023 · 18 min read

Promises are one of the coolest concepts that exist in JS. This concept became so famous that you will find different technologies trying to opt-in for something similar.

Promises in simple terms are readability enhancers (which means they can improve the code readability) and can help us to solve the problem of Inversion of control. To understand the working of Promises, we need to get a hang of two foundational concepts:

- How to Create a Promise?
- How to consume a Promise?

Now before we start with these two foundational concepts, we need to understand that Promises are native JS features i.e. you can find their discussion in the official documentation in JS (Link), and they are officially shipped with JS.

27.2 Promise Objects

A Promise is an object that is used as a placeholder for the eventual results of a deferred (and possibly asynchronous) computation.

Any Promise is in one of three mutually exclusive states: *fulfilled*, *rejected*, and *pending*:

- A promise **p** is fulfilled if **p.then(f, r)** will immediately enqueue a Job to call the function **f**.
- A promise **p** is rejected if **p.then(f, r)** will immediately enqueue a Job to call the function **r**.
- A promise is pending if it is neither fulfilled nor rejected.

A promise is said to be *settled* if it is not pending, i.e. if it is either fulfilled or rejected.

A promise is *resolved* if it is settled or if it has been "locked in" to match the state of another promise. Attempting to resolve or reject a resolved promise has no effect. A promise is *unresolved* if it is not resolved. An unresolved promise is always in the pending state. A resolved promise may be pending, fulfilled or rejected.

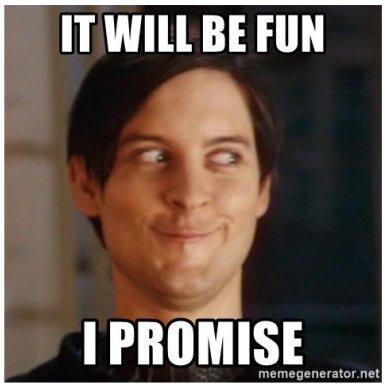
In technical terms, Promises are just another JS objects with some special capabilities. Promise objects can be used as a placeholder in situations where the answer will be a computer in future.

A Promise object has the following key properties:

- **State**: every promise object has a state value which can be either `Pending` or `Rejected` or `Fulfilled`.
  - **Pending**: Initial state of a promise, when the work is still in progress. It is also the default value.
  - **Fulfilled**: If the future task completes successfully then the state becomes fulfilled.
  - **Rejected**: If the future task doesn't complete successfully, then the state becomes Rejected.
- **Value**: every promise object has a value property associated with it. When we initially create the promise, the state of the promise is `Pending`, so till the time state of the promise remains pending, the value property remains `undefined`. When the state changes, to either rejected or fulfilled, then value can become something else.
- **onFullfilled**: This is an array that stores a bunch of callback functions that will be triggered (not run) once the state of the Promise object changes to `fulfilled`. All of these callback functions inside the array take one argument which is the `value` property of the same Promise.
- **onRejected:** This is same to same as the `onFulfilled` array it's just that the callbacks stored inside this array are executed once the state of the promise changes to `Rejected`. All of these callback functions inside the array take one argument which is the `value` property of the same Promise.

Note: Once you change the state of a promise from `Pending` to `Rejected` or `Fulfilled` then you cannot change them again.

How to create a Promise object?



To create a promise object we can use the `Promise` class constructor. We can do a new Promise and it will create a brand new Promise object for us.

Now this Promise constructor requires a mandatory argument which is a callback function also referred to as `executor callback`. This executor callback takes two arguments `resolve` and `reject` which are mainly some functions (we will discuss later).

```
let p = new Promise(function exec(resolve, reject) {
    // executor callback
});
```

```
> new Promise(function exec(resolve, reject) {
      // executor callback
  });
◂ ▶ Promise {<pending>}
>
```

If we execute just this much piece of code, we will get an output like this. You can see it has `<Pending>` written on it, which refers to the pending state.

```
> new Promise(function exec(resolve, reject) {
      // executor callback
```

```
});
Promise {<pending>}
    [[Prototype]]: Promise
    [[PromiseState]]: "pending"
    [[PromiseResult]]: undefined
```

If you will expand it in the browser there is a `PromiseResult` property which is the same `value` property we talked about.

Now if you remember, I said that once the task is done state of the promise changes. But it is still not changing. Why is that?

It is because we have not written the logic about when should the status change. And yes this is our responsibility to keep an eye on the state change. So where should we write the logic? So we write our logic inside the `executor callback`

```
let p = new Promise(function exec(resolve, reject) {
    // executor callback
   // Your logic goes here .......
});
```

Now JS will not give any special permissions to the promise object as it is a native feature, so the execution of code inside the executor callback (until async operation powered by runtime comes up) is always synchronous.

Now if you'll run the above code where `exec` is empty, you will immediately get the promise object as there is no blocking piece of code in the `exec` callback.

```
new Promise(function exec(res, rej) {
    for(let i = 0; i < 10000000000; i++) {
        // sync piece of code
    }
});
```

In the above piece of code, we have to block for loop, so the completion of the `exec` function is also halted and hence you will not immediately get the promise object, you might need to wait for it.

```
new Promise(function exec(res, rej) {
    let a = 10;
    setTimeout(function f() {
        console.log("timer done");
    }, 2000);
    a++;
});
```

In this above piece of code, as there is no blocking piece of code, every line is just a constant time operation, and even if we have a timer, this timer will be run by the runtime in the background, JS will just trigger the timer and come back.

Now you can write your future task here (in exec), which might be somehow time-consuming, and once that future task is done, you can do one of the following:

- Either signal success by making the promise go to a Fulfilled state
    - To go to a Fulfilled state we just need to call the `resolve` function with an argument. Whatever argument you pass in the `resolve` function will get allocated to the `value` property of the promise and the promise will move to the fulfilled state.
- Or signal error by making the promise go to a Rejected state.
    - To go to a Rejected state we just need to call the `reject` function with an argument. Whatever argument you pass in the `reject` function will get allocated to the `value` property of the promise and the promise will move to the rejected state.

```
let p = new Promise(function exec(resolve, reject) {
    console.log("inside exec");
```

```
    let a = 30;
    console.log("Started the timer");
    setTimeout(function f() {
        a += 10;
        resolve(a);
        console.log("Timer done");
    }, 5000);
    console.log("Timer is running");
    a += 5;
});
```

In the above piece of code, we call the promise constructor with the `exec` callback, which prints `inside exec` the moment we start the exec function execution. Then creates a variable and then goes on to start a timer in the runtime. It just triggers the timer and comes back as it will not wait for it. Once the triggering of the callback is done, JS comes back and then prints `The timer is running`. And now no other line of code is left, so `the exec` function is done, which means we got a new promise object (And the timer is still running behind the scenes).

Now the promise we got is going to be in a `Pending` State.

```
inside exec
[[PromiseResult]] the timer
Timer is running
undefined
> p
Promise {<pending>}
    [[Prototype]]: Promise
    [[PromiseState]]: "pending"
    [[PromiseResult]]: undefined
```

After 5s, when the timer will be complete, it will execute the setTimeout callback, `f` where we call the `resolve` function with value 45, and hence the state of the promise changes, and the value property is updated to 45.

```
Timer done
> p
Promise {<fulfilled>: 45}
    [[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: 45
```

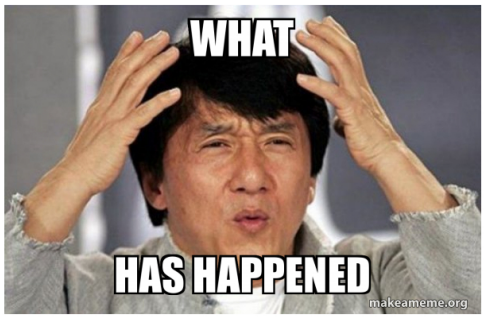
```
let p = new Promise(function exec(resolve, reject) {
    console.log("inside exec");
    let a = 30;
    console.log("Started the timer");
    setTimeout(function f() {
        a += 10;
        reject(a);
        console.log("Timer done");
    }, 5000);
    console.log("Timer is running");
    a += 5;
});
```

In this function instead of moving from Pending to fulfilled it got rejected this time, but the same thing will happen, the state will change for the promise object and the value property of the promise object is 45.

Note: As we said earlier, the state of promise changes at once and only once, so if you try to call `resolve` or `reject` multiple times, then it will take effect only once as if you just wrote one of them one time. (Value also never changes again and again).

What happens when Promise State

changes?



When a Promise state changes from `Pending` to `fulfilled`, whatever value we call the `resolve` function gets attached to the `value` property of the promise object, and it cannot be changed. Apart from that, all the callback functions which were waiting in the `onFulfilled` array, all of them are transferred to `Micro task Queue`

So we already know about Macro Task Queue, that when an async function completes, its callback has to wait inside the Macro task queue / Callback Queue. But in the case of promises, when a promise is fulfilled, then all the functions which are meant to be executed after promise fulfilment (which are stored in the onFulfilled array) go into the Micro task queue.

You might be thinking, why do these callbacks (of promises) have to go to the Microtask queue? Why they cannot be immediately executed?

Because Promise fulfilment will happen sometime in future due to some async tasks. If we have some code running on our main thread, and in between the promise fulfils, then JS will never hamper the flow of the main thread and will not execute these callback functions immediately hence they have to wait in the Micro task queue.

Now one more question might come to your mind, why Promise based callbacks have to wait in the Micro task queue and not in the Macro task queue? Because JS wants to set the priority of the callback executions, the priority of callbacks waiting in the Microtask queue is always and always higher than callbacks waiting in the Macrotask Queue.

So if at any point in time, there is a callback waiting in the Macrotask queue and another callback waiting in the Microtask queue, then the one in the Microtask queue will be executed first.

Now you might be thinking, does the execution of the Micro task queue is also maintained by the Event loop? Just like how it does for Macro task queue?

YES. The event loop keeps on checking that is the call stack empty or not and if the global code is completely executed or not. Once both are done, then it first checks if there is a callback in the Micro Task queue or not. If there is then it will be executed first as the Micro task queue has higher priority. Once the Micro task queue is empty then the Macro task queue is looked into.

In case the Promise gets `Rejected` then all of the above theory is applied to the callbacks of the `onRejected` array.

How to Consume a Promise?

Every Promise object that we get by doing `new Promise` has a `.then` function in it.

This `.then` function takes two arguments, the first argument is a `fulfillHandlerCallback` and the second argument is `rejectHandlerCallback`.

```
COPY
p.then(function fulfillHandler(){}, function rejectHandler() {});
// p is a promise object
```

The moment we execute `p.then` only one thing happens, i.e. both the handlers are registered in their corresponding arrays, i.e. `fulfillHandler` is pushed in the `onFulfilled` array and `rejectHandler` is pushed in the `onRejected` array. Keep in mind, when you do `.then` then these callbacks are not executed, they are just pushed (or you can say registered) in their corresponding arrays.

So when are these executed? When the state of the promise changes from `pending` to

So when are these executed? When the state of the promise changes from `pending` to `fulfilled` or `rejected` these callbacks are transferred to the microtask queue.

You can add multiple callbacks by writing multiple `.then`

```
p.then(function fulfillHandler1(){}, function rejectHandler1() {});
p.then(function fulfillHandler2(){}, function rejectHandler2() {});
p.then(function fulfillHandler3(){}, function rejectHandler3() {});
```

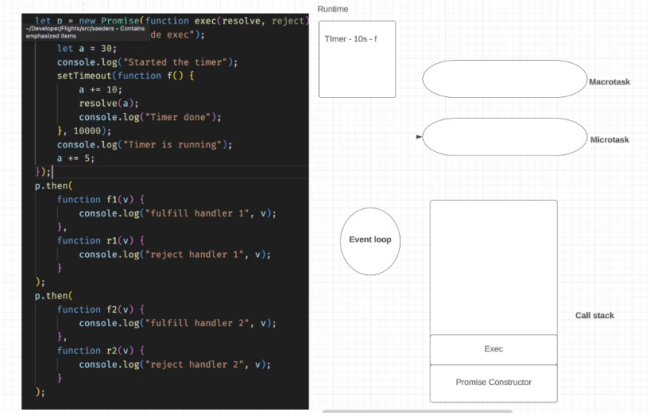
Now if you write the above code, then there will be 3 functions added in both arrays.

Note: The `rejectHandler` callback argument is not mandatory, so if you don't pass the second argument, then only the `fulfillHandler` goes to the `onFulfilled` array and nothing goes in the `onRejected`.

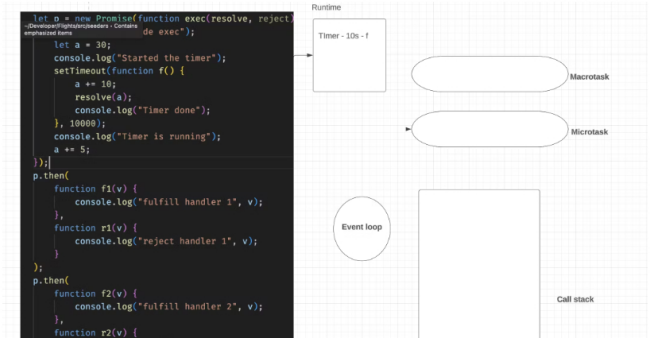
```
p.then(function fulfillHandler(){});
```

Now these callbacks (i.e. fulfil and reject callbacks) can also take one argument. This argument is the value property of the Promise object.

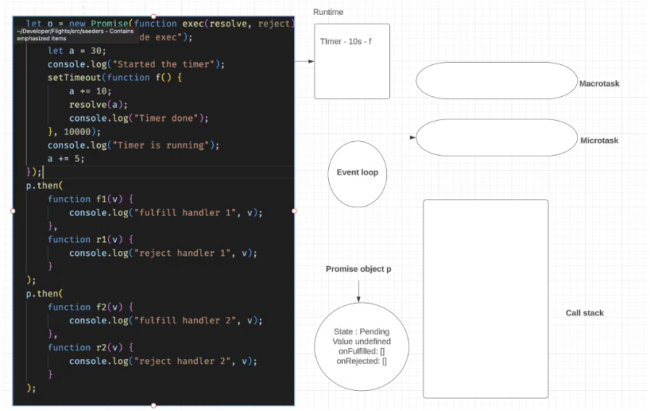
```
let p = new Promise(function exec(resolve, reject) {
    console.log("inside exec");
    let a = 30;
    console.log("Started the timer");
    setTimeout(function f() {
        a += 10;
        resolve(a);
        console.log("Timer done");
    }, 10000);
    console.log("Timer is running");
    a += 5;
});
p.then(function f1(v) { console.log("fulfill handler 1", v); }, function r1(v) {
p.then(function f2(v) { console.log("fulfill handler 2", v); }, function r2(v) {
```



So the current state of execution is that we have called the Promise constructor, and inside it, we are calling the `executor callback`. Inside the executor callback, we initiated a timer of 10s after whose completion we should execute the callback function f. JS will just trigger the timer and comes back.

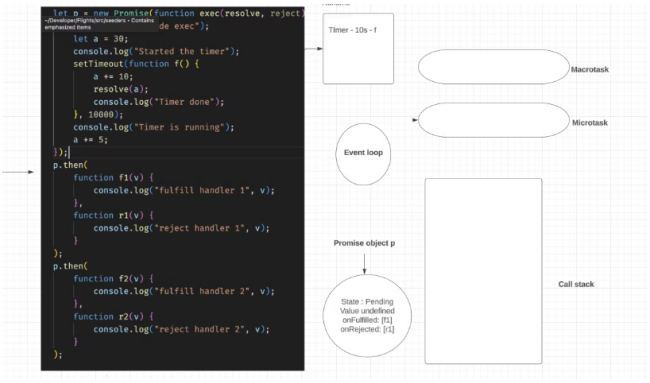


Now the executor callback is done, and we remove it from the call stack, and now The promise object will also be created, with State as `pending` and value property as `undefined` and `onFullfiled` array is also empty and `onRejected` array is also empty.

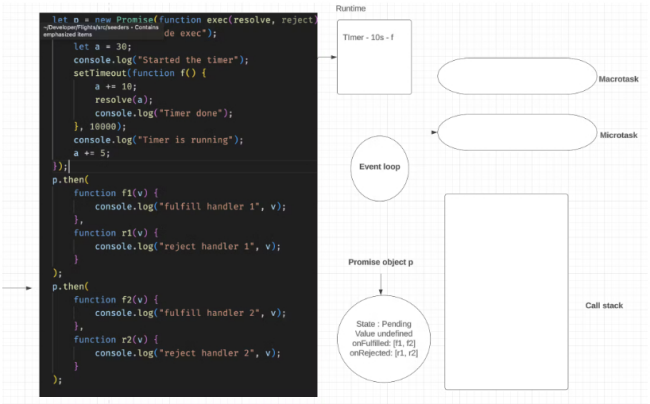


Both the arrays are empty because we have not yet registered any `.then`.

Now we encounter our first `p.then` call.



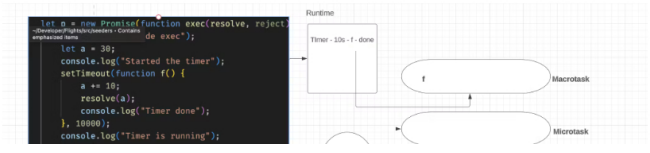
Now the moment we hit the first `p.then` we register `f1` in the `onFulfilled` array and `r1` in the `onRejected` array.

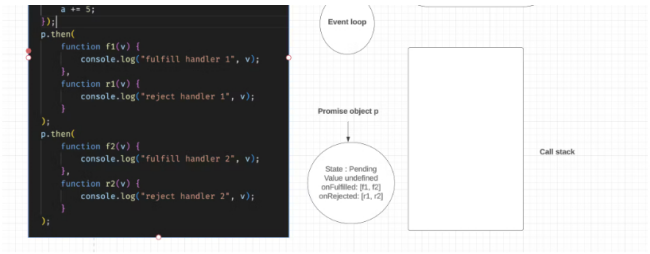


Now the moment we hit the second `p.then` we register `f2` in the `onFulfilled` array and `r2` in the `onRejected` array.

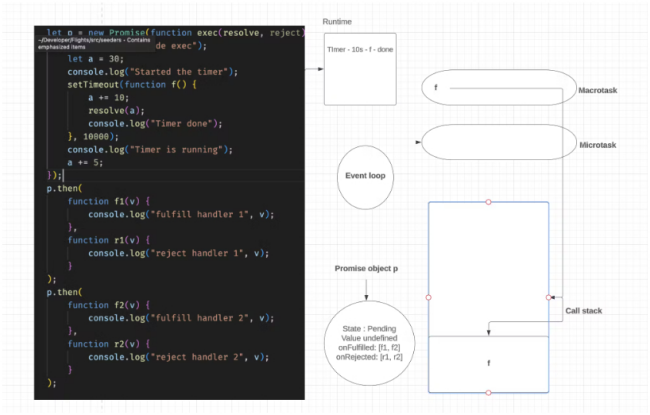
And by this timer is still going on.

Now let's say timer is completed after 10s.

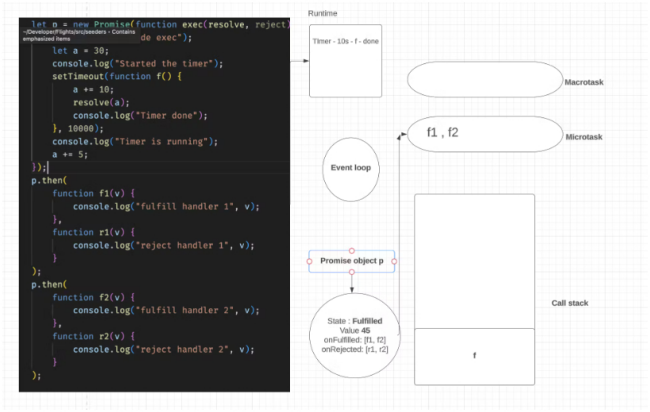




The moment timer is completed it will go to the macro task queue, meanwhile event loop is constantly in the call stack empty. YES. Is the global piece of code all done? YES. Is the Microtask queue empty? YES.

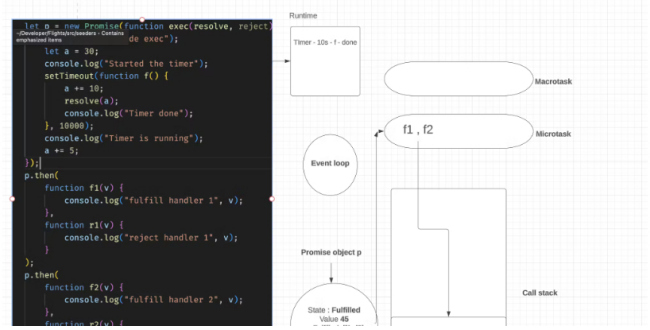


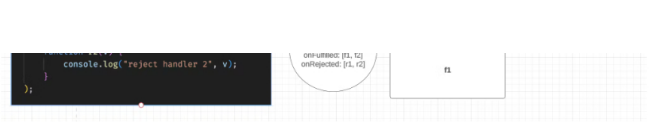
Now event loop will bring the callback function `f` from the Macro task queue to the call stack. In the function `f` we increment the value of `a` and then call the `resolve` function with the value a (which is 45)



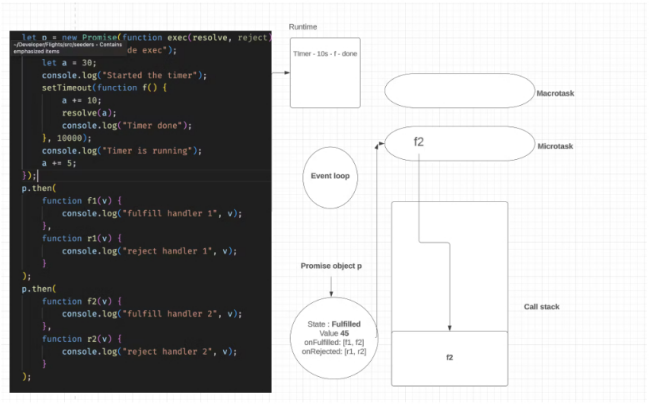
So now, the state of the promise changed to fulfilled, the value changed to 45 and it immediately `f1` and `f2` callbacks are sent from `onFulfilled` array to Microtask queue. But these callbacks cannot be immediately executed. Why? Because the event loop will check is the call stack empty? No. The function `f` is still going on as there is still one line of code left i.e. `console.log("Timer done")`.

The moment function f completes. event loop will check if the call stack is empty. yes. Is the global code done? Yes. So Event loop will check if the Microtask queue is empty? No.





First of all, `f1` will be called and executed.



Then `f2` will be called and executed.

When both of them are done, the Microtask queue will be empty, and there is nothing in the Macrotask queue, call stack or global code, so the Program ends.

What is `Promise.resolve`?

So `Promise.resolve` is a static method, that actually can take an argument and return an already resolved Promise with the value property of the Promise object set as the argument.

```
console.log("hi")
const promise1 = Promise.resolve(123);

promise1.then((value) => {
  console.log(value);
  // Expected output: 123
});
console.log("by");
```

The output of the above code will be

```
hi
by
123
```

Now you might be thinking, we got an already resolved promise, so why `123` is printed after `by`?

There is a simple reason for it, JS assumes that generally promise objects might be handling some async code. The moment you write `Promise.resolve` it has to first create a plain promise object with `Pending` state and then behind the scenes (apart from the main thread) the state will be changed to `resolved`, now this state change takes some time, maybe some microseconds, but JS will not wait for any time, because it is uncertain how much time it can take to resolve a promise. JS won't give this function any preference and the moment you call `Promise.resolve` at that moment just create an object and moves ahead, and the rest is handled in a similar way to how other promises will be handled.

`Promise.resolve` is a shorthand for the below code

```
new Promise((resolve) => resolve(value));
```

Now let's take a look at this example:

```
p = new Promise((res, rej) => {
    setTimeout(() => {res(10);}, 4000);
});
x = Promise.resolve(p);
x.then((value) => {console.log(value);})
```

Here if you will execute this code, it will not immediately print anything, it will wait for at least 3s and then print the value 10, why? Because the object x is made by `Promise.resolve` and the argument passed to it is a pending promise. So till the time argument is not resolved, object x's promise will not be resolved. When you pass a non-promise argument like a Number, String etc, then these are treated as immediately resolvable promises. Read more here

Note: Just like `promise.resolve` we have `Promise.reject` as well, read more here.

The .then function...

We know that `.then` function helps us to register the fulfillHandler and rejectHandler. It doesn't call them, it just registers them in the `onFulfilled` and `onRejected` arrays of the promise object.

Now `.then` is also a function so it must be returning something, right? It returns another Promise object (a brand new one). Every `.then` returns a new Promise object. Now the `.then` function also returns a promise, and it will be fulfilled with undefined if you will not return something manually from it.

```
x = Promise.resolve(11);
console.log(x);
y = x.then((v) => {console.log(v)})
console.log("what is .then returning ? ", y);
```

```
11
undefined
x
▶ Promise {<fulfilled>: 11}
y
▶ Promise {<fulfilled>: undefined}
```

So in the above code, we are not returning anything manually from the fulfillHandler of `.then` so we get undefined.

```
x = Promise.resolve(11);
console.log(x);
y = x.then((v) => { console.log(v); return 100; })
console.log("what is .then returning ? ", y);
```

```
what is .then returning ?  ▶ Promise {<pending>}
11
undefined
x
▶ Promise {<fulfilled>: 11}
y
▶ Promise {<fulfilled>: 100}
```

Here we returned 100 from fulfillHandler of `.then` so the promise is also resolved by 100.

Now if `y` is a promise, we can do `y.then` again and register new handlers.

```
x = Promise.resolve(11);
console.log(x);
```

```
y = x.then((v) => {console.log(v); return 100;})
console.log("what is .then returning ? ", y);
y.then((v) => {console.log("resolving y", v)})
```

```
▶ Promise {<fulfilled>: 11}
what is .then returning ?  ▶ Promise {<pending>}
11
resolving y 100
▶ Promise {<fulfilled>: undefined}
```

So here once, the promise x is resolved, we call the arrow function registered in `x.then`, once that arrow function returns 100, we call the arrow function inside `y.then`.

That means we can do a `.then` chaining as well.

```
x = Promise.resolve(11);
y = x
.then((v) => {console.log("x is", v); return 100})
.then((v) => {console.log("x.then is", v); return 123})
.then((v) => {console.log("x.then.then is", v)});
```

```
x is 11
x.then is 100
x.then.then is 123
▶ Promise {<fulfilled>: undefined}
```

When will the promise of `.then` gets rejected?

if the initial promise of the chain is rejected, then with whatever value we return the reject handler, will be the value of then's promise, and it will also reject.

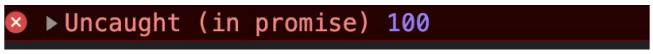
```
x = Promise.reject(11);
y = x
.then((v) => {console.log("x is", v); return 100}, (v) => {console.log("reject x
.then((v) => {console.log("x.then is", v); return 123}, (v) => {console.log("re
.then((v) => {console.log("x.then.then is", v)}, (v) => {console.log("reject x.t
```

```
reject x with 11
x.then is 100
x.then.then is 123
▶ Promise {<fulfilled>: undefined}
```

Here, Promise x got rejected with value 11, so the rejection handler of x was called, but this handler is returning a value, and return means everything went well with the function, so the fulfillHandler of `x.then` is called and so on.

```
x = Promise.reject(11);
y = x
.then((v) => {console.log("x is", v); return 100}, (v) => {console.log("reject x
.then((v) => {console.log("x.then is", v); return 123}, (v) => {console.log("re
.then((v) => {console.log("x.then.then is", v)}, (v) => {console.log("reject x.t
```

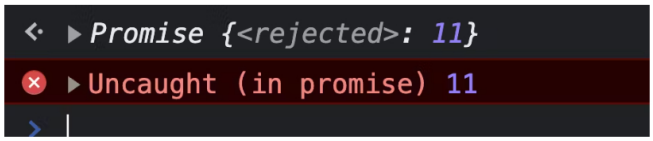
```
reject x with 11
reject x.then with 100
reject x.then.then with 100
▶ Promise {<rejected>: 100}
```



Here, Promise x got rejected with value 11, so the rejection handler of x was called, & this handler is throwing a value, and throwing means something went wrong with the function, so the rejectHdnler of `x.then` is called and so on.

Now we know that writing a reject handler is not mandatory right? So what happens then?

```
x = Promise.reject(11);
y = x
.then((v) => {console.log("x is", v); return 100})
.then((v) => {console.log("x.then is", v); return 123})
.then((v) => {console.log("x.then.then is", v)});
```



Here, Promise x got rejected with value 11, so the rejection handler of x was called, which is not present, and the whole chain is missing the reject handlers, that's why the reject handler never fulfils any chained promise hence everything is rejected.

To handle this, what we can do, we can write a `.catch` chained with the last `.then`.

```
x = Promise.reject(11);
y = x
.then((v) => {console.log("x is", v); return 100})
.then((v) => {console.log("x.then is", v); return 123})
.then((v) => {console.log("x.then.then is", v)})
.catch((err) => {console.log("handled", err);});
```

So here, we don't handle the error anywhere and directly handle it in the `.catch`

of the chain. If no error comes in the chain, `.catch` is never called.

How did promises help to resolve the inversion of control?

Now here, we are not passing our callbacks to other functions, we are not giving those functions control over us. We are keeping our callbacks at our call site, so we are damn sure they will be called once.

```
function fakeDownloader() {
    return new Promise((res, rej) => {
        setTimeout(() => {
            res("data");
        }, 4000);
    });
}
```

Now I am not passing any callback here, in a callback-based code, we need to pass a callback to execute something after downloading is over. But as we have Promises, we don't need callbacks.

```
let p = fakeDownloader();
p.then((data) => {console.log("downloaded data is", data)});
```

The fulfillHandler of `p.then` is expected to be executed once the download is done,

and we can see the control is with us

and we can see the control is with us.

If this same functionality was written with a callback it would look something like this:

```
function fakeDownloader(cb) {
  setTimeout(() => {
     cb("data");
  }, 4000);
}
fakeDownloader((data) => {console.log("downloaded data is", data)});
```

But we are not sure how they are handling the callback, as they might call it more than once, or maybe never. But with promises, even if somebody tries to call `res("data")` more than once, the state of the promise changes only once, hence callback will be called only once.

```
function fakeDownloader() {
    return new Promise((res, rej) => {
        setTimeout(() => {
            res("data");res("data");res("data");
        }, 4000);
    });
}
let p = fakeDownloader();
p.then((data) => {console.log("downloaded data is", data)});
```

The above code will call a callback only once.

```
function fakeDownloader(cb) {
  setTimeout(() => {
     cb("data");cb("data");cb("data");
  }, 4000);
}
fakeDownloader((data) => {console.log("downloaded data is", data)});
```

The above code will call a callback thrice.

So this is how inversion of control is resolved.

What about callback hell?

You can already see that the promise-based implementation is cleaner and better understood. It doesn't create a pyramid-like structure.

There is a concept of Promise hell as well, where people start writing nested Promises. Read more here. This can be easily avoided by using a `.then` chaining or async await.

.

.

.

.

Lukewarm regards

Sanket Singh

JavaScript | Javascript Promises | React | Frontend Development | promises

Written by

Sanket Singh

I am currently working as a Software engineer 2 at Google. I previously worked at LinkedIn and Interviewbit/Scaler as well. I was selected for Google Summer Of Code under the mentorship of Harvard university in 2019. I was selected as a Speaker at PyCon Italy and NDC Melbourne and provided mentorship to thousands of students and working professionals in the past.

Follow

MORE ARTICLES

Sanket Singh

Understanding Build Systems with Bazel

To fully appreciate the value of build systems in managing multi-file projects, let's explore a scen...

Sanket Singh

Getting Started With React From A Bird's Eye View

I am assuming that if you have landed on this blog then, you are an absolute beginner in React or if...

Sanket Singh

Let's Learn Linked Lists with JavaScript

Linked Lists We already know that Arrays as a data structure exist. Using arrays we can implement a ...