

Chapter 20: Callback

There are 2 Parts of Callback:

1. Good Part of Callback - Callbacks are super important while writing asynchronous code in JS
2. Bad Part of Callback - Callbacks inside Callback leads to issues like -
 - Callback Hell
 - Inversion of control

- Understanding Bad part of Callback is super important to learn Promise in next chapter.

> 💡 JavaScript is synchronous, single threaded language. It can Just do one thing at a time. JS Engine has just one call-stack where it executes code line by line, it does not wait.

```
console.log("Namaste");  
console.log("JavaScript");  
console.log("Season 2");
```

```
// Namaste  
// JavaScript  
// Season 2
```

```
// 💡 It is quickly printing because `Time, tide & JavaScript waits for none.`
```

But what if we have to delay code execution of any line. We could utilize Callback.

```
console.log("Namaste");  
setTimeout(function () {  
  console.log("JavaScript");  
}, 5000);  
console.log("Season 2");
```

```
// Namaste  
// Season 2  
// JavaScript
```

```
// 💡 Here we are delaying the execution using Callback approach of setTimeout.
```

CALLBACK HELL -

🛒 e-Commerce web app situation

Assume a scenario of e-Commerce web, where one user is placing order, he has added items like, shoes, pants and kurta in cart and now he is placing order. So, in backend the situation could look something like this.

```
const cart = ["shoes", "pants", "kurta"];  
// Two steps to place an order  
// 1. Create an Order  
// 2. Proceed to Payment
```

// It could look something like this:

```
api.createOrder();  
api.proceedToPayment();
```

Assume, once order is created then only we can proceed to payment, so there is a dependency. So How to manage this dependency.

Callback can come to the rescue, How?

```
api.createOrder(cart, function () {  
  api.proceedToPayment();  
});  
// 💡 Over here `createOrder` api is first creating a order then it is responsible to call  
`api.proceedToPayment()` as part of callback approach.
```

To make it a bit complicated, what if after the payment, you have to show Order summary by calling `api.showOrderSummary()` and now it has dependency on `api.proceedToPayment()`

Now my code should look something like this:

```
api.createOrder(cart, function () {  
  api.proceedToPayment(function () {  
    api.showOrderSummary();  
  });  
});
```

Now what if we have to update the wallet, now this will have a dependency over ``showOrderSummary``

```
api.createOrder(cart, function () {  
  api.proceedToPayment(function () {  
    api.showOrderSummary(function () {  
      api.updateWallet();  
    });  
  });  
});
```

// 💡 Callback Hell - Callbacks inside Callback creates a call back hell structure.

When we have a large codebase having numbers of APIs with internal dependencies to each other, then we fall into Callback hell where the code will grow horizontally and it is very difficult to read and maintain. This Callback hell structure is also known as **Pyramid of Doom**.

Till this point we are comfortable with concept of Callback hell but now let's discuss about **`Inversion of Control`**. It is very important to understand in order to get comfortable around the concept of promise.

INVERSION OF CONTROL –

> 💡 Inversion of control is like, you lose the control of code when we are using Callbacks.

Let's understand with the help of example code and comments:

```
api.createOrder(cart, function () {  
  api.proceedToPayment();  
});
```

// 💡 So over here, we are creating an order and then we are blindly trusting ``createOrder`` API to call ``proceedToPayment``.

// 💡 It is risky, as ``proceedToPayment`` is important part of code and we are blindly trusting ``createOrder`` to call it and handle it.

// 💡 When we pass a function as a Callback, basically we are dependent on our parent function that it is his responsibility to run that function. This is called ``inversion of control`` because we are dependent on that function. What if parent function stopped working, what if it was developed by another programmer or Callback runs two times or it never runs at all.

// 💡 in next chapter, we will see how we can fix such problems.

> 💡 Async programming in JavaScript exists because Callback exists. (more at ``http://callbackhell.com/``)

Chapter 21: Promises

> Promises are used to handle async operations in JavaScript.

We will discuss with code example that how things used to work before ``Promises`` and then how it works after ``Promises``

Suppose, taking an example of E-Commerce

```
const cart = ["shoes", "pants", "kurta"];
```

// Below two functions are asynchronous and dependent on each other

```
const orderId = createOrder(cart);  
proceedToPayment(orderId);
```

// In code below, it is the responsibility of createOrder function to create the order first and call the Callback function proceedToPayment later. But with this approach we have a problem called ``Inversion of Control (discussed on previous chapter)``

```
createOrder(cart, function () {  
  proceedToPayment(orderId);  
});
```

Q: How to fix the above issue?

Ans: By using Promise.

Now, we will make ``createOrder`` function returning a promise object and we will capture that ``promise`` into a ``variable``.

Promise is nothing but a plain javascript object having properties such as Prototype, PromiseState and PromiseResult . PromiseResult will hold data whatever this ``createOrder`` function will return.

A typical promise object looks like:

{Prototype:<Prototype>,PromiseState:<promiseState>,PromiseResult:<data>} where data is a variable factor and it depends on the promise data itself like when the data is available <data> is replaced by the actual data . Initially while making network calls <data> is undefined.

Since ``createOrder`` function is an async function, we don't know how much time will it take to finish execution.

So, the moment ``createOrder`` API gets executed, it will return an ``undefined`` value. Let's say after 5 seconds, execution got finished and ``orderId`` value is available. Now this undefined value will be replaced by the actual data i.e. ``orderId``.

In short, when `createOrder` got executed, it immediately returned a `promise object` with `undefined` value. Then JavaScript continued to execute next lines of code. After sometime when `createOrder` finished its execution, `orderId` data was available and the earlier value undefined got replaced with actual order Id data.

Q: How will we get to know `response` is ready?

Ans: we usually attach a `Callback` function to the `promise object` using `then` that get triggered automatically when `result` is ready.

```
const cart = ["shoes", "pants", "kurta"];
```

```
const promiseRef = createOrder(cart);
```

```
// promiseRef is a promise which has access to `then ()`
```

```
// Initially promise data will be undefined, so function inside then() won't trigger.
```

```
// After a while, when execution has finished and when promiseRef gets the data then automatically  
the function inside then() gets triggered.
```

```
promiseRef.then(function () {  
  proceedToPayment(orderId);  
});
```

Q: How is it better than Callback approach?

In Earlier solution we used to pass the function and then used to trust the function to execute the Callback.

But with promise, we are attaching a Callback function to a promise Object.

There is difference between these words `passing a function` and `attaching a function`.

Promise guarantees that it will call the `then` attached Callback function only once when the data is available. We call it as `Promise is fulfilled or resolved`. And if the data is not available then Promise will call the `catch` attached Callback function and we call it as `promise is rejected`.

Earlier we talked about promise is an object with empty data but that's not entirely true, `Promise` is much more than that.

Now let's understand and see a real promise object.

`fetch` is a web-API which is utilized to make API call and it returns a promise.

We will be calling public github api (<https://api.github.com/users/alok722>) to fetch user data.

```
const URL = "https://api.github.com/users/alok722";
const user = fetch(URL); // user is a promise object
console.log(user); // Promise {<Pending>}
```

/ OBSERVATIONS:

```
* As we have discussed before `promise` object has 3 properties.
* `prototype`, `promiseState` & `promiseResult`
* `promiseResult` is the same data which we talked earlier as the returned data from APIS & initially
`promiseResult` is `undefined`
* `promiseResult` will store data returned from API call
* `promiseState` will tell in which state the promise is currently in, initially it will be in `pending`
state and later it will become `fulfilled` or rejected depending on the data availability.
*/

/
* When above line is executed, `fetch` makes API call and return a `promise` instantly which is in
`Pending` state and JavaScript doesn't wait to get it `fulfilled`
* And in next line it consoles out the `pending promise`.
* NOTE: chrome browser has some in-consistency, the moment console happens it shows in
pending state but if you will expand that it will show fulfilled because chrome updated the log when
promise get fulfilled.
* Once fulfilled, data is available in promiseResult and this data is not directly accessible to the
external world. This data is in readable stream format and there is a way to extract it.
*/
```

How we can attach Callback to above response?

Using `.then``

```
const URL = "https://api.github.com/users/alok722";
const user = fetch(URL); // you can't edit this user data as promise is immutable
```

```
user.then(function (data) {
  console.log(data);
});
```

```
// And this is how Promise is used. It guarantees that it could be resolved only once, either it could
be `success` or `failure`
```

```
/
```

```
*Promise State-
```

A Promise is in one of these states:

pending: initial state, neither fulfilled nor rejected.

fulfilled: meaning that the operation was completed successfully.

rejected: meaning that the operation failed.

```
*/
```

💡 Promise Objects are immutable.

-> Once promise is fulfilled, we get the data. We don't have to worry that someone can mutate/change that data because by nature promise is immutable.

Mutable - we can modify an existing object.

Immutable - If we try to modify an object, a new object will be created. The original object will not be changed.

So over above we can't directly mutate ``user`` promise object.

Interview Guide

💡 What is Promise?

-> Promise object is a placeholder for certain period of time until we receive value from asynchronous operation.

-> A container for a future value.

-> A Promise is an object representing the eventual completion or failure of an asynchronous operation.

We are now done solving one issue of Callback i.e. Inversion of Control. But there is one more issue, Callback hell.

// Callback Hell Example

```
createOrder(cart, function (orderId) {  
  proceedToPayment(orderId, function (paymentInf) {  
    showOrderSummary(paymentInf, function (balance) {  
      updateWalletBalance(balance);  
    });  
  });  
});
```

// And now above code is expanding horizontally and this is called pyramid of doom. Callback hell is ugly and hard to maintain. Promise fixes this issue too using ``Promise Chaining``.

// Example Below is a Promise Chaining

```
createOrder(cart)
  .then(function (orderId) {
    proceedToPayment(orderId);
  })
  .then(function (paymentInf) {
    showOrderSummary(paymentInf);
  })
  .then(function (balance) {
    updateWalletBalance(balance);
  });
```

// ⚠ Common Pitfall - We forget to return promise in Promise Chaining.

// The idea is promise data returned can be used in the next promise object in the promise chain and so on. We are passing the promise data down to the promise chain and we need return keyword for that to happen.

```
createOrder(cart)
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInf) {
    return showOrderSummary(paymentInf);
  })
  .then(function (balance) {
    return updateWalletBalance(balance);
  });
```

// To improve readability, you can use arrow function instead of regular function

Promise chain Example -

```
const promiseObject = createOrder(cart)
  .then(function (orderId) {
    return proceedToPayment(orderId);
  })
  .then(function (paymentInf) {
    return showOrderSummary(paymentInf);
  })
  .then(function (balance) {
    return updateWalletBalance(balance);
  });
```


Splitting promise chain into individual promises -

promise 1

```
const createOrderPromise = createOrder(cart) // we get the order ID after the order is created
```

promise 2

```
const proceedToPaymentPromise = createOrderPromise.then(  
  function (orderId) {  
    return proceedToPayment(orderId);  
  }) // we passed the object ID from the promise object into proceedToPayment() which will return  
another promise
```

promise 3

```
const ShowOrderSummaryPromise = proceedToPaymentPromise.then(function (paymentInf) {  
  return showOrderSummary(paymentInf);  
}) // The process continues
```

promise 4

```
const UpdateWalletBalancePromise = ShowOrderSummaryPromise.then(function (balance) {  
  return updateWalletBalance(balance);  
}); // The process continues ...
```

Chapter 22: Creating a Promise, Chaining & Error Handling

promise - consumer part - How a promise is consumed -

```
const cart = ["shoes", "pants", "kurta"];

const promise = createOrder(cart); // orderId
// What will be printed in below line?
// It prints Promise {<pending>}, but why?
// Because above createOrder is going to take some time to get resolved, so pending state. But once
the promise is resolved, promise attached ` . then ` calls the Callback to get executed.
console.log(promise);

promise.then(function (orderId) {
  proceedToPayment(orderId);
})

// Now we will see, how createOrder is implemented so that it is returning a promise. In short, we
will see, "How we can create Promise" and then return it.
```

promise - producer part - how a promise is produced/created -

```
function createOrder(cart) {
  // JS provides a Promise constructor through which we can create promise. It accepts a Callback
  function with two parameters `resolve` & `reject`.
  const promise = new Promise(function (resolve, reject) {
    // What is this `resolve` and `reject`?
    // These are function which are passed by JavaScript to us in order to handle success and failure of
    function call.
    // pseudocode to `createOrder`
    // 1. Validate Cart
    // 2. Insert in DB and get an orderId
    // We are assuming in real world scenario, validateCart would be defined
    if (!validateCart(cart)) { // If cart is not valid, reject the promise
      const err = new Error("Cart is not Valid");
      reject(err);
    }
    const orderId = "12345"; // We got this id by calling to database (Assumption)
    if (orderId) {
      // If cart is valid, resolve the promise
      resolve(orderId);
    }
  });
  return promise;
}
```

```
function validateCart(cart){
  return true // cart is validated
}
```

In code above, if the cart is validated successfully, the promise will be resolved (success case),

Now let's see if there was some error and we are rejecting the promise, **how we could catch that?**

-> Using `.catch``

```
const cart = ["shoes", "pants", "kurta"];
```

```
const promise = createOrder(cart); // orderId
```

```
// Here we are consuming Promise and will try to catch promise error
```

```
promise
```

```
.then(function (orderId) {
  // ✅ success- promise resolved case
  proceedToPayment(orderId);
})
.catch(function (err) {
  // ⚠️ failure - promise reject case
  console.log(err);
});
```

```
// Here we are creating Promise
```

```
function createOrder(cart) {
  const promise = new Promise(function (resolve, reject) {
    // Assume below `validateCart` return false then the promise will be rejected
    // And then our browser is going to throw the error.
    if (!validateCart(cart)) {
      const err = new Error("Cart is not Valid");
      reject(err);
    }
    const orderId = "12345";
    if (orderId) {
      resolve(orderId);
    }
  });
  return promise;
}
```

Now, Let's understand the concept of Promise Chaining

-> In promise chaining, whatever is returned from the first promise becomes data for the next promise and so on.

-> At any point of promise chaining, if one of the promises is rejected, the execution will fall back to `.catch` provided there is only one `.catch` exist at the end of promise chain. When this happens, others promise in the promise chain won't run. Let's take an example.

```
const cart = ["shoes", "pants", "kurta"];
```

```
createOrder(cart)
  .then(function (orderId) {
    // ✓ success - promise is resolved
    console.log(orderId);
    return orderId;
  })
  .then(function (orderId) {
    // ✓ success - promise is resolved
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    // ✓ success - promise is resolved
    console.log(paymentInfo);
  })
  // Handling promise error. Below code will be executed in case either one of the above promises is
  // rejected. But in our case intentionally we are resolving all the promises. Thus, below code will not be
  // executed.
```

```
.catch(function (err) {
  console.log(err);
});
```

```
function createOrder(cart) {
  const promise = new Promise(function (resolve, reject) {
    // If `validateCart` returns false then the promise will be rejected and then our browser will throw
    // the error if it is not handled properly. For now, assume validateCart returns true.
    if (!validateCart(cart)) {
      const err = new Error("Cart is not Valid");
      reject(err);
    }
    const orderId = "12345";
    if (orderId) {
      resolve(orderId);
    }
  });
  return promise;
}
```

```
function proceedToPayment(cart) {
  return new Promise(function (resolve, reject) {
    // For the time being, we are simply `resolving` this promise
    resolve("Payment Successful");
  });
}
```

Q: What if we want to continue execution even if any of the promise is failing, how to achieve this?

-> By placing the `.catch` block next to the promise which has a chance of failure.

-> once `.catch` handles the promise JS will keep executing the next promise in the promise chain.

-> There could be multiple `.catch` block in a promise chain.

Example-

```
createOrder(cart)
  .then(function (orderId) {
    console.log(orderId);
    return orderId; // ⚠ lets say the promise is rejected here.
  })
  .catch(function (err) {
    // This catch block is only responsible for handling errors in above promise or promises along the
    chain. In our case we have one promise and we encountered error and our catch block here is
    handling that error.

    console.log(err);
  });
  .then(function (orderId) {
    console.log('This block is definitely executed')
    return proceedToPayment(orderId);
  })
  .then(function (paymentInfo) {
    console.log(paymentInfo);
  })

  .catch(function (err) { // generic catch block for promise error handling
    console.log(err);
  });
```

Note- We can only resolve a promise once. not more than that.

Chapter 23 - Async & Await

Topics Covered:

- What is async?
- What is await?
- How async await works behind the scenes?
- Example of using async/await
- Error Handling
- Interviews
- Async await vs Promise.then/.catch

Q: What is async?

A: Async is a keyword that is used before a function to create an async function.

Q: What is Async function and how it is different from Normal function?

💡 Async function always returns a promise, even if I return a simple string from async function, async keyword will wrap it under Promise object and then returns whereas Normal functions return anything depending on what the context is . It even returns a promise out of it.

Async function returning a Non-promise value -

```
async function getData() {  
  return "Namaste JavaScript";  
}  
const dataPromise = getData();  
console.log(dataPromise); // Promise {<fulfilled>: 'Namaste JavaScript'}  
  
//💡 How to extract data from above promise? One way is using promise.then()  
dataPromise.then(res => console.log(res)); // Namaste JavaScript
```

Async function returning a promise -

```
const p = new Promise((resolve, reject) => {  
  resolve('Promise resolved value!!');  
})  
  
async function getData() {  
  return p;  
}  
  
// In above case, since we are already returning a promise async function would simply return that  
// instead of wrapping with a new Promise.  
  
const dataPromise = getData();
```

```
console.log(dataPromise); // Promise {<fulfilled>: 'Promise resolved value!!'}
dataPromise.then(res => console.log(res)); // Promise resolved value!!
```

Q: How we can use `await` along with async function?

A: `async` and `await` combo is used to handle promises.

Q: How we used to handle promises earlier and why do we even need async/await?

Handling Promise before async and await were introduced -

```
const p = new Promise((resolve, reject) => {
  resolve('Promise resolved value!!');
})

function getData() {
  p.then(res => console.log(res));
}

getData(); // Promise resolved value!!
```

// ✂ Till now we have been using Promise.then/.catch to handle promise.
// Now let's see how async await can help us and how it is different
// The rule is we have to use keyword await in front of promise.

Handling Promise after async and await are introduced –

```
const p = new Promise((resolve, reject) => {
  resolve('Promise resolved value!!');
})

async function getData() {
  const val = await p;
  console.log(val);
}

getData(); // Promise resolved value!!
```

✂ `await` is a keyword that can only be used inside a `async` function.

```
await function() {} // Syntax error: await is only valid under async function.
```

Q: What makes `async` - `await` special?

A: Let's understand with one example where we will compare async-await way of resolving promise with older `.then/.catch` fashion. For that we will modify our promise `p` with some delay factor added.

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Promise resolved value!!');
  }, 3000);
})
```

Handling Promise using then and catch

```
function getData() {
  // JS engine will not wait for promise to be resolved
  p.then(res => console.log(res));
  console.log('Hello There!');
}
getData();
```

```
// Output :
// Hello There!
// Promise resolved value!!
```

Code Explanation:

As we know JavaScript waits for none, so JS engine will register this promise Callback in a separate space and attaches a Timer of 3 sec to it and then it moves to the next line and prints 'Hello There' in the console. Once 3 sec is elapsed promise data is available, after that then () is invoked which in turns logs the promise response in the console.

Handling Promise using async and await

```
async function handlePromise() {
  // JS Engine will wait the for promise to be resolved
  const val = await p;
  console.log('Hello There!');
  console.log(val);
}
handlePromise();
```

```
// This time 'Hello There!' won't be printed immediately instead after 3 secs 'Hello There!' will be
// printed followed by 'Promise resolved value!!'
// 💡 So basically code was waiting at 'await' line to get the promise resolve before moving on to
// next line.
```

```
// Above is the major difference between Promise.then/.catch vs async-await
```


🤖 Let's brainstorm more around async-await

```
async function handlePromise() {  
  console.log('Hi');  
  const val = await p;  
  console.log('Hello There!');  
  console.log(val);  
  
  const val2 = await p;  
  console.log('Hello There! 2');  
  console.log(val2);  
}  
handlePromise();
```

// In above code example, will our program wait for 2 time or will it execute parallelly.
// 🏹 `Hi` printed instantly -> now code will wait for 3 secs -> After 3 secs both promises will be resolved so ('Hello There!' 'Promise resolved value!!' 'Hello There! 2' 'Promise resolved value!!') will get printed immediately.

// Let's create one promise and then resolve two different promise.

```
const p2 = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve('Promise resolved value by p2!!');  
  }, 2000);  
})
```

```
async function handlePromise() {  
  console.log('Hi');  
  const val = await p;  
  console.log('Hello There!');  
  console.log(val);  
  
  const val2 = await p2;  
  console.log('Hello There! 2');  
  console.log(val2);  
}  
handlePromise();
```

// 🏹 `Hi` printed instantly -> now code will wait for 3 secs -> After 3 secs both promises will be resolved so ('Hello There!' 'Promise resolved value!!' 'Hello There! 2' 'Promise resolved value by p2!!') will get printed immediately. So even though `p2` was resolved after 2 secs it had to wait for `p` to get resolved

// Now let's reverse the order execution of promise and observe response.

```
async function handlePromise() {  
  console.log('Hi');  
  const val = await p2;  
  console.log('Hello There!');  
  console.log(val);  
  
  const val2 = await p;  
  console.log('Hello There! 2');  
  console.log(val2);  
}  
handlePromise();
```

// ⚡ `Hi` printed instantly -> now code will wait for 2 secs -> After 2 secs ('Hello There!' 'Promise resolved value by p2!!') will get printed and in the subsequent second i.e. after 3 secs ('Hello There! 2' 'Promise resolved value!!') will get printed

Q: Is JavaScript program actually waiting or what is happening behind the scene?

A: As we know, Time, Tide and JS wait for none. And it's true. Over here it appears that JS engine is waiting but JS engine is not waiting over here. It has not occupied the call stack if that would have been the case our page may have got frozen. So, JS engine is not waiting. So, if it is not waiting then what it is doing behind the scene? Let's understand with below code snippet.

```
const p1 = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve('Promise resolved value by p1!!');  
  }, 5000);  
})
```

```
const p2 = new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve('Promise resolved value by p2!!');  
  }, 10000);  
})
```

```
async function handlePromise() {  
  console.log('Hi');  
  debugger;  
  const val = await p;  
  console.log('Hello There!');  
  debugger;  
  console.log(val);  
  
  const val2 = await p2;  
  console.log('Hello There! 2');
```

```

debugger;
console.log(val2);
}
handlePromise();

```

// When this function is executed, it will go line by line as JS is synchronous single threaded language. Lets observe what is happening under call-stack. Above you can see we have set the break-points.


Code Flow:

* handlePromise () is pushed and It will log `Hi` to console first.

*In the next line as soon as JavaScript engine sees the await keyword, where promise is supposed to be resolved, will it wait for promise to resolve and block call stack? No, it does not but it suspends the execution of handlePromise () till the promise is resolved and moved out of call stack.

*when `p` gets resolved after 5 secs, handlePromise () will be pushed to call-stack. But this time JS Engine will start executing code from where it was left off. Now it will log 'Hello There!' and 'Promise resolved value!!'

*In the next line it will check whether `p2` is resolved or not. but P2 will take 10 secs to be resolved. The point JavaScript engine reaches to this line 5 seconds had already elapsed so it will take 5 more seconds to resolve promise for P2. same process will repeat. Execution will be suspended until promise is resolved.

*  the important point is JS is not waiting, call stack is not getting blocked. But it appears to be waiting. (Imaginative)

* Moreover, in above scenario what if p1 would be taking 10 secs and p2 5 secs -> even though p2 got resolved earlier but JS is synchronous single threaded language so it will first wait for p1 to be resolved, once p1 is resolved it prints Hello There and promise p1 object value and then immediately execute p2 along with the subsequent lines of code.

Real World example of async/await

```

async function handlePromise() {
  // How fetch works? fetch(https://api.github.com/users/alok722) returns a Response Object as
  // body as Readable stream and this is also a promise and we are waiting for this promise to get
  // resolved. Thats why the await keyword is used before the promise object.
  const data = await fetch('https://api.github.com/users/alok722');
  // Response or data.json() returns a promise with promise data in json format but again the return
  // value is also a promise which is why the second await keyword is used before the promise object to
  // get it resolved.
  const res = await data.json();
  console.log(res);
};
handlePromise()

```

Error Handling

While we were using normal Promise, we were using `.catch` to handle error, now in `'async-await'` we would be using `'try-catch'` block to handle error.

```
async function handlePromise() {  
  try {  
    const data = await fetch('https://api.github.com/users/alok722');  
    const res = await data.json();  
    console.log(res);  
  } catch (err) {  
    console.log(err)  
  }  
};  
handlePromise()
```

// In above whenever any error will occur the execution will move to catch block. One could try above with bad URL which will result in error.

// Other way of handling error:

```
handlePromise().catch(err => console.log(err)); // this will work as handlePromise will return error  
promise in case of failure.
```

Async await vs Promise.then/.catch

What one should use? `'async-await'` is just a syntactic sugar around promise. Behind the scene `'async-await'` is just promise. So, both are same, it's just `'async-await'` is new way of writing code. `'async-await'` solves few of the short-coming of Promise like `'Promise Chaining'`. `'async-await'` also increases the readability. So, sort of it is always advisable to use `'async-await'`.

Chapter 24: Promise APIs + Interview Questions

Promise APIs

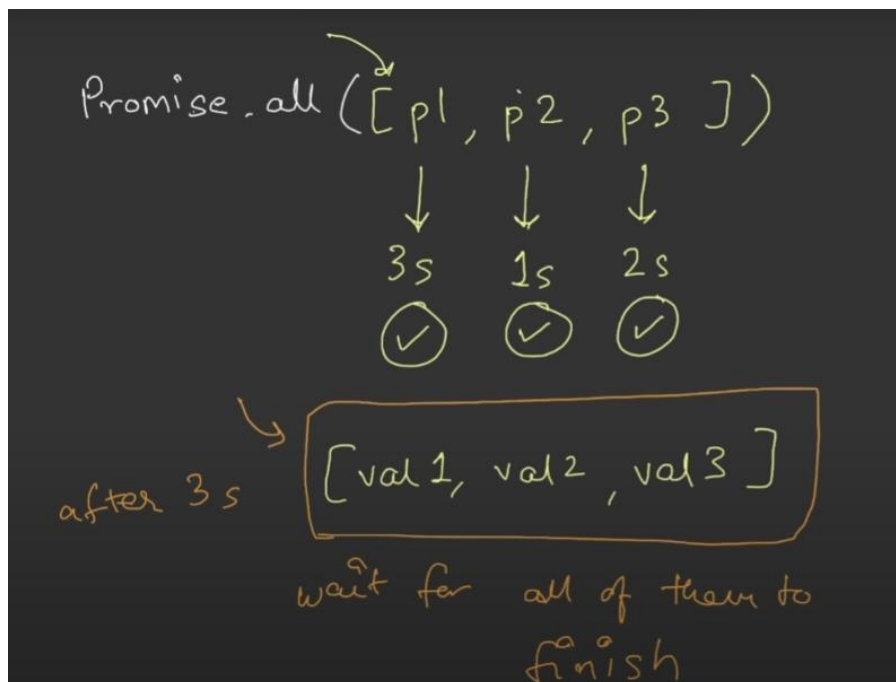
- * Promise.all
- * Promise.allSettled
- * Promise.race
- * Promise.any

Promise.all(<Iterable>) – (Fail Fast API)

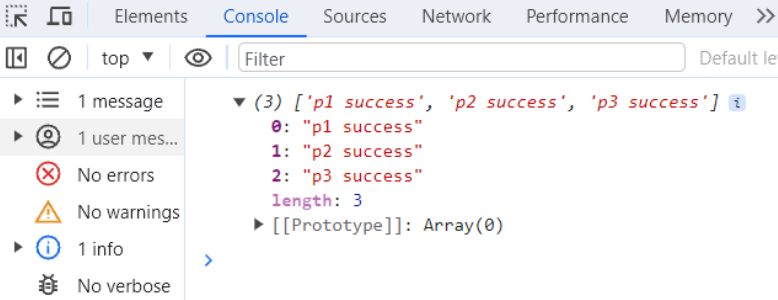
Iterable is usually an array of promises. Let's consider Three promises within the Iterable array. [P1, P2, P3]

Case 1: If all of the promises are resolved.

In our case, Promise.all() makes three API calls for three promises in parallel and wait for all the promises to get resolved. So, this API will take time for the promise which takes maximum time to get resolved. Promise.all() returns an array of promise results.

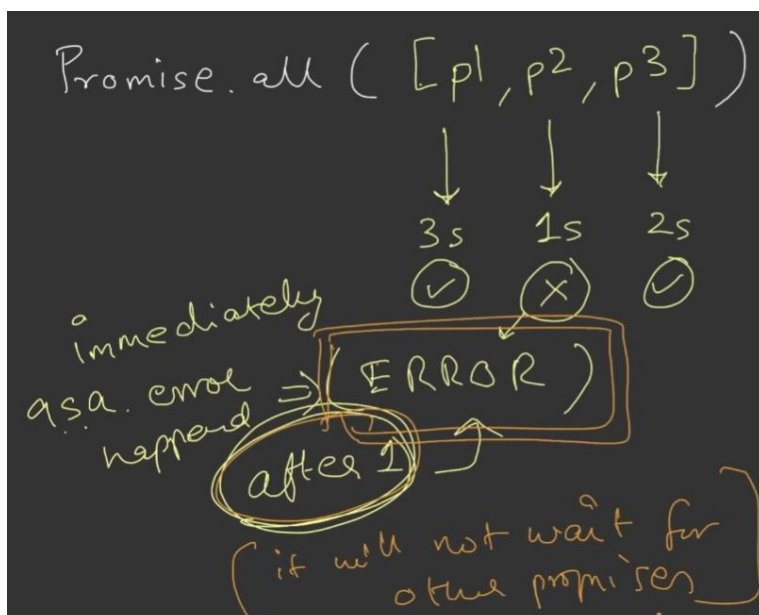


```
index.js x
index.js > ...
1  const p1 = new Promise((resolve, reject) => {
2    |   setTimeout(() => {
3    |     | resolve("p1 success");
4    |   }, 3000);
5  });
6
7  const p2 = new Promise((resolve, reject) => {
8    |   setTimeout(() => {
9    |     | resolve("p2 success");
10   |   }, 1000);
11 });
12
13 const p3 = new Promise((resolve, reject) => {
14   |   setTimeout(() => {
15   |     | resolve("p3 success");
16   |   }, 2000);
17 });
18
19 Promise.all([p1, p2, p3]).then((res) => console.log(res));
```



Case 2: If any/some/all of the promises are rejected.

If any one of the promises is failed, then this API will throw an error and it will not wait for the other promises to get settled (resolved/rejected). Whatever error it will get from the rejected promise, the API will throw the same error as a result.



```
index.js x
index.js > ...
1  const p1 = new Promise((resolve, reject) => {
2    |   setTimeout(() => {
3    |     |   resolve("p1 success");
4    |   }, 3000);
5    | });
6
7  const p2 = new Promise((resolve, reject) => {
8    |   setTimeout(() => {
9    |     |   reject("p2 fail");
10   |   }, 1000);
11   | });
12
13  const p3 = new Promise((resolve, reject) => {
14    |   setTimeout(() => {
15    |     |   resolve("p3 success");
16    |   }, 2000);
17   | });
18
19  Promise.all([p1, p2, p3])
20    .then((res) => console.log(res))
21    .catch((err) => console.error(err));
22
```

Elements Console Sources

top Filter

1 message p2 fail

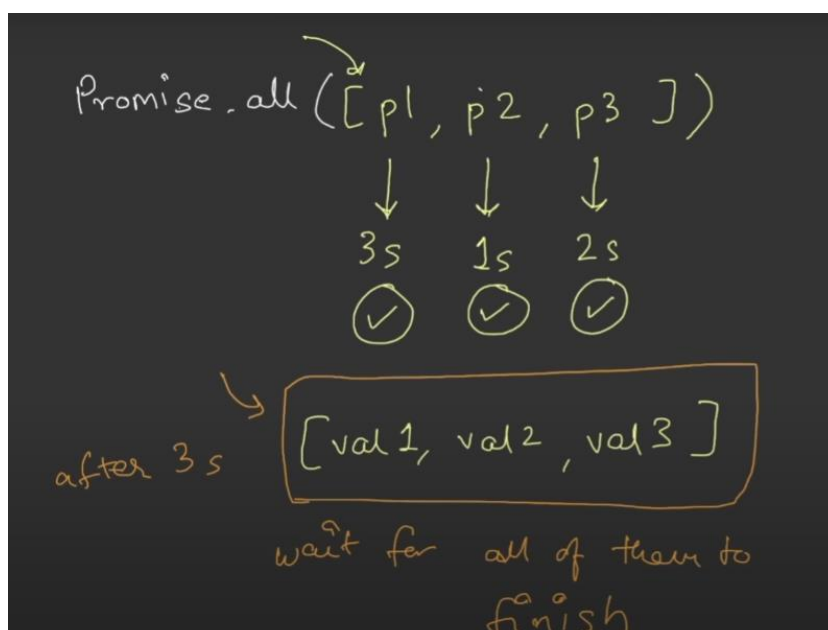
1 user mes...

1 error

Promise.allSettled(<Iterable>) –

Case 1: If all of the promises are resolved.

In our case, Promise.allSettled() makes three API calls for three promises in parallel and wait for all the promises to get resolved. So, it will take time for the promise which takes maximum time to get resolved. Promise.allSettled() returns an array of promise results.



```
index.js  X
index.js > ...
1  const p1 = new Promise((resolve, reject) => {
2    setTimeout(() => {
3      resolve("p1 success");
4    }, 3000);
5  });
6
7  const p2 = new Promise((resolve, reject) => {
8    setTimeout(() => {
9      resolve("p2 fail");
10   }, 1000);
11  });
12
13 const p3 = new Promise((resolve, reject) => {
14   setTimeout(() => {
15     resolve("p3 success");
16   }, 2000);
17  });
18
19 Promise.allSettled([p1, p2, p3])
20   .then((res) => console.log(res))
21   .catch((err) => console.error(err));
```

Elements Console Sources Network Performance Memory

top Filter

1 message

1 user mes...

No errors

No warnings

1 info

No verbose

(3) [{...}, {...}, {...}]

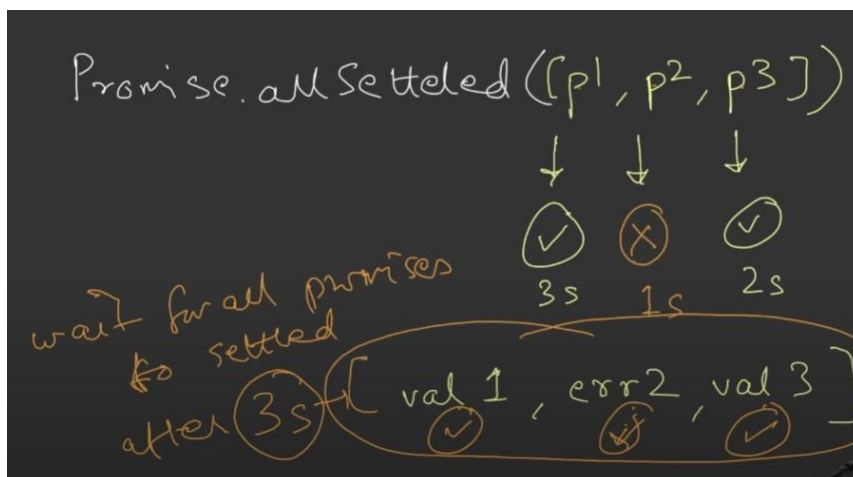
- 0: {status: 'fulfilled', value: 'p1 success'}
- 1: {status: 'fulfilled', value: 'p2 fail'}
- 2: {status: 'fulfilled', value: 'p3 success'}

length: 3

[[Prototype]]: Array(0)

Case 2: If one/some of the promises are rejected.

If one/some of the promises is/are rejected, then this API will wait for other promises to get settled(resolved/rejected), then returns the array of promise results.




```

index.js  X
index.js > ...
1  const p1 = new Promise((resolve, reject) => {
2    |   setTimeout(() => {
3    |     |   resolve("p1 success");
4    |   }, 3000);
5  });
6
7  const p2 = new Promise((resolve, reject) => {
8    |   setTimeout(() => {
9    |     |   reject("p2 fail");
10   |   }, 1000);
11  });
12
13  const p3 = new Promise((resolve, reject) => {
14    |   setTimeout(() => {
15    |     |   resolve("p3 success");
16    |   }, 2000);
17  });
18  ⚡
19  Promise.allSettled([p1, p2, p3])
20    .then((res) => console.log(res))
21    .catch((err) => console.error(err));

```

Elements Console Sources Network Performance Memory >>

top Filter Default lev

1 message
1 user mes...
No errors
No warnings
1 info
No verbose

(3) [{...}, {...}, {...}] i

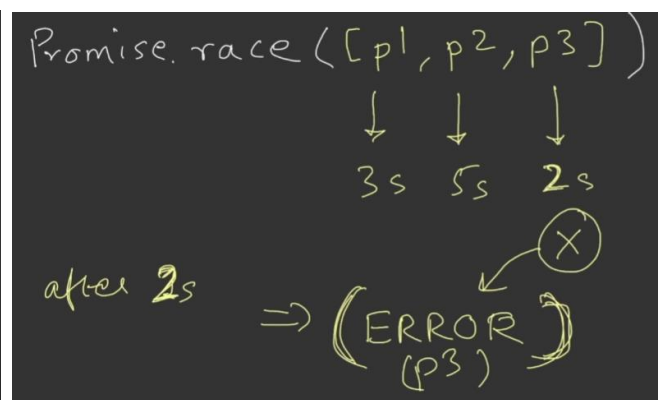
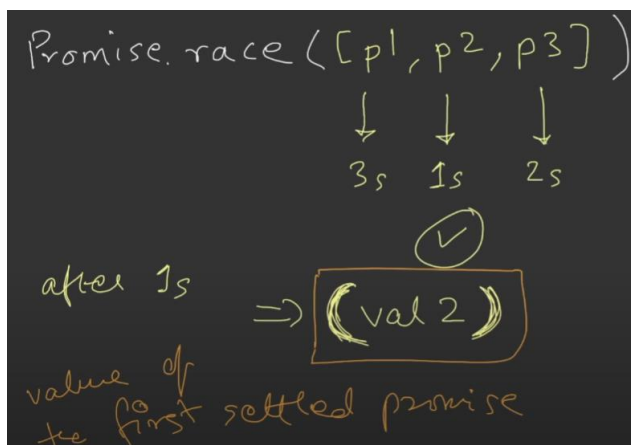
- 0: {status: 'fulfilled', value: 'p1 success'}
- 1: {status: 'rejected', reason: 'p2 fail'}
- 2: {status: 'fulfilled', value: 'p3 success'}

length: 3
[[Prototype]]: Array(0)

Promise.race(<Iterable>) –

The promise who will finish first or get settled first will be the winner.

The API will return first settled promise result irrespective of whether the promise is resolved or rejected.



```

index.js  X
index.js > ...
1  const p1 = new Promise((resolve, reject) => {
2    |   setTimeout(() => {
3    |     |   resolve("p1 success");
4    |   }, 3000);
5  });
6
7  const p2 = new Promise((resolve, reject) => {
8    |   setTimeout(() => {
9    |     |   resolve("p2 success");
10   |   }, 1000);
11 });
12
13 const p3 = new Promise((resolve, reject) => {
14   |   setTimeout(() => {
15   |     |   resolve("p3 success");
16   |   }, 2000);
17 });
18
19 Promise.race([p1, p2, p3])
20   .then((res) => console.log(res))
21   .catch((err) => console.error(err));

```

Elements Console Sources Network Performance

top Filter

- 1 message p2 success
- 1 user mes... >
- No errors
- No warnings

```

index.js  X
index.js > ...
1  const p1 = new Promise((resolve, reject) => {
2    |   setTimeout(() => {
3    |     |   resolve("p1 success");
4    |   }, 3000);
5  });
6
7  const p2 = new Promise((resolve, reject) => {
8    |   setTimeout(() => {
9    |     |   reject("p2 fail");
10   |   }, 1000);
11 });
12
13 const p3 = new Promise((resolve, reject) => {
14   |   setTimeout(() => {
15   |     |   resolve("p3 success");
16   |   }, 2000);
17 });
18
19 Promise.race([p1, p2, p3])
20   .then((res) => console.log(res))
21   .catch((err) => console.error(err));
22

```

Elements Console Sources Network

top Filter

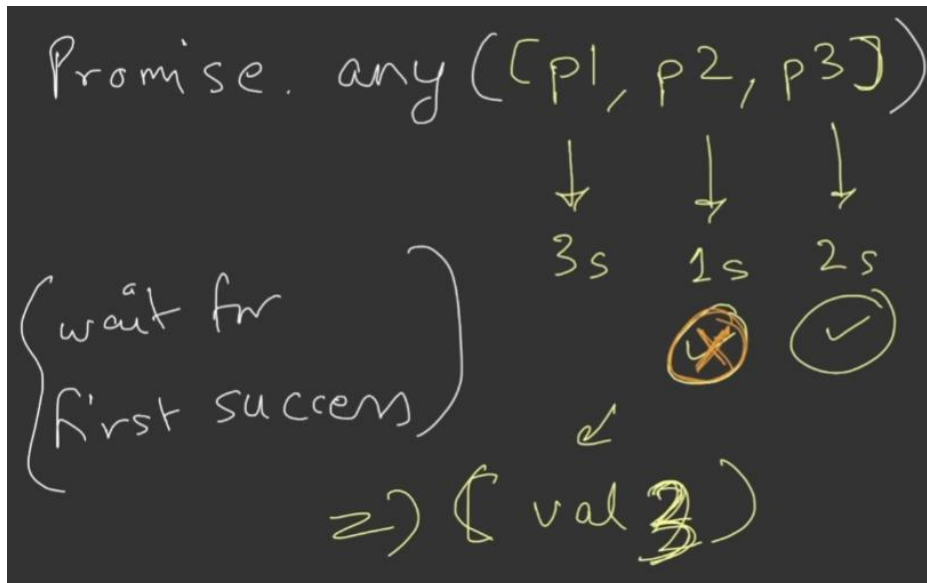
- 1 message **p2 fail**
- 1 user mes... >
- 1 error
- No warnings
- No info

Promise.any(<Iterable>) – (Success seeking API- receives only the first succeeded promise value)

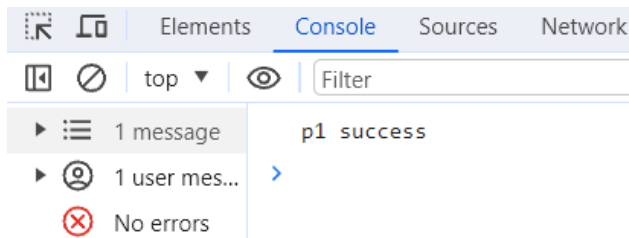
Similar to race API, the only difference is any API will wait for the first promise to get successful. It returns the first successful promise result.

Case 1: If any of the promise is resolved

As soon as one of the promises is resolved, any API returns the resolved promise result.

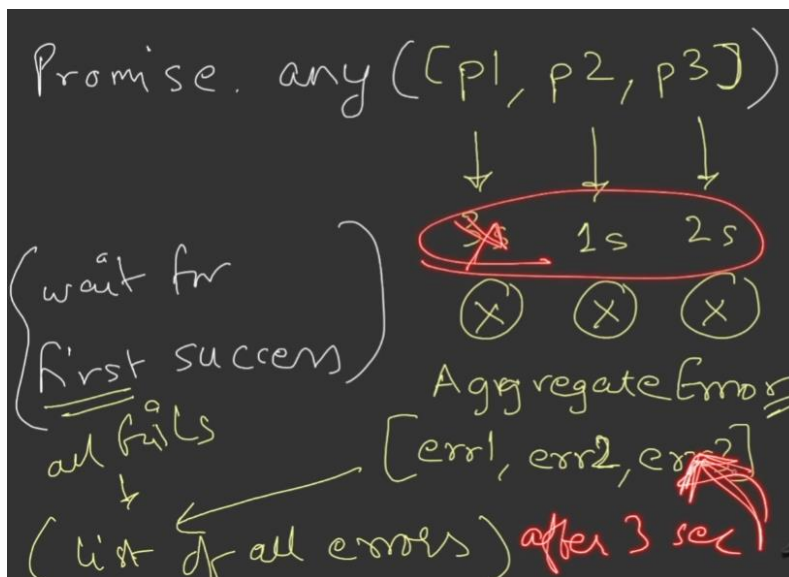


```
index.js  X
index.js > ...
1  const p1 = new Promise((resolve, reject) => {
2    |   setTimeout(() => {
3    |     | resolve("p1 success");
4    |   }, 3000);
5  });
6
7  const p2 = new Promise((resolve, reject) => {
8    |   setTimeout(() => {
9    |     | reject("p2 failed");
10   |   }, 1000);
11  });
12
13 const p3 = new Promise((resolve, reject) => {
14   |   setTimeout(() => {
15   |     | reject("p3 failed");
16   |   }, 2000);
17  });
18
19 Promise.any([p1, p2, p3])
20   .then((res) => console.log(res))
21   .catch((err) => console.error(err));
--
```

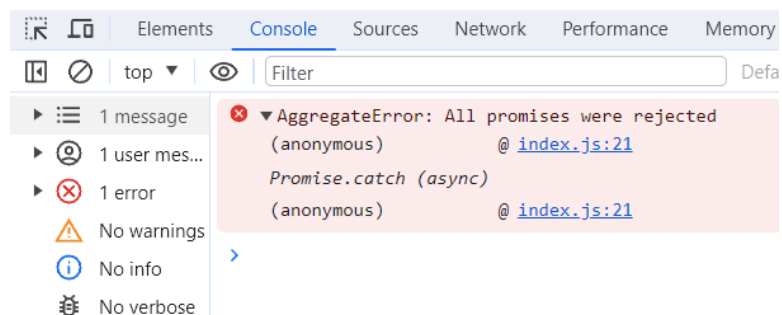


Case 2: If all of the promise is rejected

This API will return an aggregate error which will be an array of all three errors thrown by all three promises.



```
index.js x
index.js > p3 > <function> > setTimeout() callback
1  const p1 = new Promise((resolve, reject) => {
2    |   setTimeout(() => {
3    |     reject("p1 failed");
4    |   }, 3000);
5  });
6
7  const p2 = new Promise((resolve, reject) => {
8    |   setTimeout(() => {
9    |     reject("p2 failed");
10   |   }, 1000);
11  });
12
13  const p3 = new Promise((resolve, reject) => {
14    |   setTimeout(() => {
15    |     reject("p3 failed");
16    |   }, 2000);
17  });
18
19  Promise.any([p1, p2, p3])
20    .then((res) => console.log(res))
21    .catch((err) => console.error(err));
```



```
index.js x
index.js > catch() callback
1  const p1 = new Promise((resolve, reject) => {
2    setTimeout(() => {
3      reject("p1 failed");
4    }, 3000);
5  });
6
7  const p2 = new Promise((resolve, reject) => {
8    setTimeout(() => {
9      reject("p2 failed");
10   }, 1000);
11 });
12
13 const p3 = new Promise((resolve, reject) => {
14   setTimeout(() => {
15     reject("p3 failed");
16   }, 2000);
17 });
18
19 Promise.any([p1, p2, p3])
20   .then((res) => console.log(res))
21   .catch((err) => {
22     console.error(err)
23     console.error(err.errors)
24   });
25
```

Elements Console Sources Network Performance Memory >>

top Filter Default le

2 messages 2 user mes... 2 errors No warnings No info No verbose

AggregateError: All promises were rejected

(3) ['p1 failed', 'p2 failed', 'p3 failed']

(anonymous) @ index.js:23

Promise.catch (async)

(anonymous) @ index.js:21

Meaning of promise has been settled –

Promise is either rejected or resolved.

