Problems with callbacks

1. Callback - hell

- when you want a function to be executed at some later point of time.
- allows to do **asynchronous programming**
- we pass a callback function to another function.
- a function will call a callback function

Example in an e commerce website, you need to do the following.

1. create order
2. proceed to payment ← [this step should call the proceed to payment function]
3. show order summary

code starts growing → horizontal.

```
JS index.js > api.createOrder() callback
1
2   const cart = ["shoes", "pants", "kurta"];
3
4   api.createOrder(cart, function () {
5
6       api.proceedToPayment(function () {
7
8           api.showOrderSummary(
9
10              function () {
11                  api.updateWallet()
12              }
13
14          )
15
16      })
17
18  })
19
```

UNREADABLE & UNMAINTAINABLE

2. inversion of control

→ when you lose control of your code.

> ji's function ko callback pass kiya hai, it is their responsibility to call that callback function.

you lose the control. RISKY !!

Q. what are the 2 problems with callbacks?

1. call back hell. **pyramid of doom**

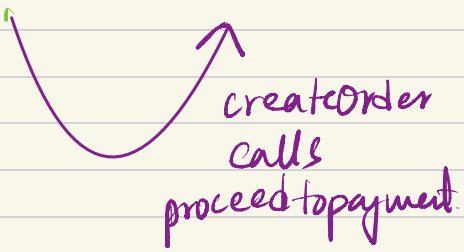
when we pass too many callbacks to callbacks to callbacks, it creates a big callback hell. the code starts to grow horizontally rather than vertically. it is difficult to read and manage.

2. inversion of control.

you lose control of the code with callback. The responsibility of calling the callback function is on the function that it's passed to. unreliable and unpredictable behaviours

Promises

can only do proceed to payment once createOrder is called.



problem in this

blindly trusting createOrder api. It is a blackbox.
we dont know what's happening in there.
we should be responsible for our own api being called.

Promise → empty object with { data : undefined } (fills with value after being executed)

const promise = creatOrder(cart);

attach your callback to the PROMISE object.

```
promise.then ( callbackfunction)
```

↓ This is promise.

how is this better?

1. attaching a callback function to promise object
   - ↳ we have the control of when the callback is called.
   - as soon as promise object is filled with data, automatically calls the call back.

2. We have a guarantee that 100% the call back will be called.

Preview of promise object

1. pending
2. fulfilled
3. rejected

states

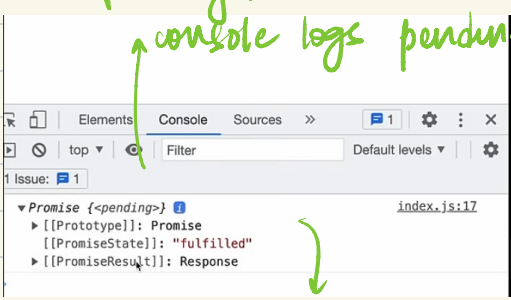
```
▼ Script
    GITHUB_API: "https://api.github.c
  ▼ user: Promise
    ▶ [[Prototype]]: Promise
      [[PromiseState]]: "pending"
      [[PromiseResult]]: undefined
```

→ state is pending

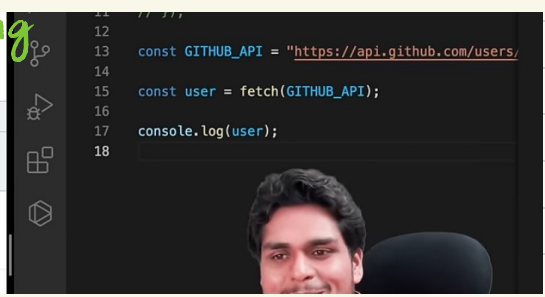
↓

no data in the object as function has not returned anything

when fetch returns the promise, it is in pending state.



console logs pending



eventually data comes in and current state is shown.

Then in the fulfilled state we can use the promise data.

```
console.log(user);

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

user.then(callback)

promise objects - immutable

→ a filled promise object cannot be changed. just passed.

2. what is a promise

promise is an object that represents the eventual completion of an async operation

This is us consuming the promise.

↳ MDN web docs.

call back hell issue resolved.

↳ promise chaining

keep adding .then to the function call itself.

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

return

return

return

promise chain.

everything passes down the drain.

always return a then from a promise.

Sometimes arrow function is used too, for simpler readability.

Writing our own promise.

we are on the production side.

```
function createOrder(cart) {

    const pr = new Promise(function(resolve, reject){
        // createOrder
        // validateCart
        // orderId
        if(!validateCart(cart)) {
            const err = new Error("Cart is not valid");
            reject(err);
        }
        // logic for createOrder
        const orderId = "12345";
        if(orderId) {
            resolve(orderId);
        }

    });
```

accept

rejecting

accepting

index.js > createOrder

```
1   const cart = ["shoes", "pants", "kurta"];
2
3   const promise = createOrder(cart); // orderId
4
5   promise.then(function (orderId) {
6     console.log(orderId);
7     //proceedToPayment(orderId);
8   });
9
10  /// Producer
11
12  function createOrder(cart) {
13    const pr = new Promise(function (resolve, reject) {
14      // createOrder
15      // validateCart
16      // orderId
17      if (!validateCart(cart)) {
18        const err = new Error("Cart is not valid");
19        reject(err);
20      }
21      // logic for createOrder
22      const orderId = "12345";
23      if (orderId) {
24        resolve(orderId);
25      }
26    });
27
28    return pr;
29  }
30
```

if any steps fail, all steps fail. catch at the end.

Catch function in promise.

→ on consumer side, catch is called when the reject is called

→ to handle the errors gracefully.

always pass down the chain.

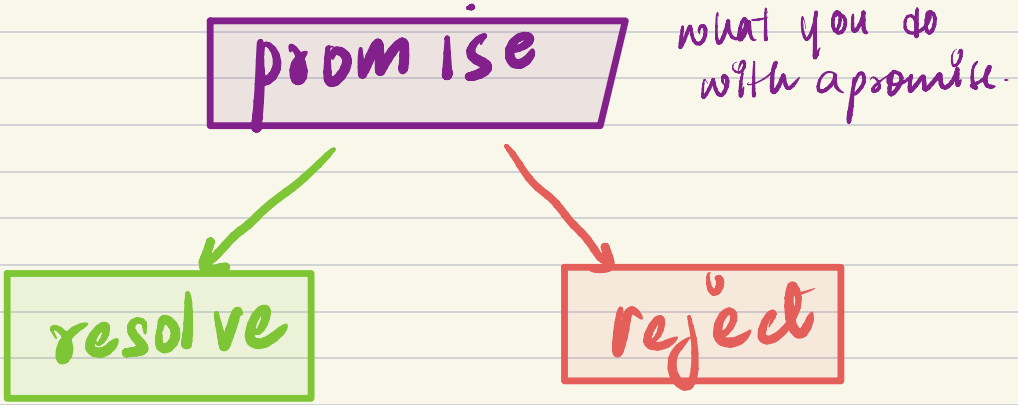
↓

* you can add the catch statement in between some .then statements.

↑ then then
catch

(only responsible for the then statements above it)

we cannot resolve a promise once.



PROMISE APIs

Promise.all([p1, p2, p3])

input : array of promises (iterable)

p1 → 3s
p2 → 1s
p3 → 2s

fail-fast

takes 3 s

makes 3 parallel api calls and gets result.

waits for all of them to finish, before giving result

success case:

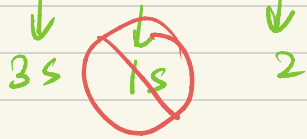
all p1, p2, p3 result in success

then promise.all will give result

[val1, val2, val3]

failure case : if any one fails

promise.all([p1, p2, p3])



as soon as any of these promises get rejected, Promise fails and gives ERROR

as soon as error happens, after 1sec the promise fails. does not wait for p1, p3

what if i want success results?

Promise.allSettled([p1, p2, p3])

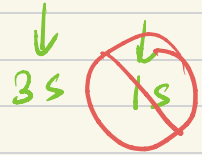
success case;

same output array of success results.

failure case,

waits for all promises to get settled → success / → failure

promise.all([p1, p2, p3])



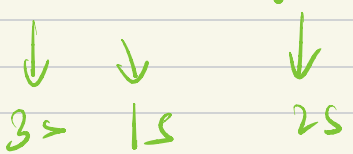
3s. waits for all.

[val1, err2, val3]

whatever the values were are presented as array

Promise.race([p1, p2, p3])

its a race!



winner

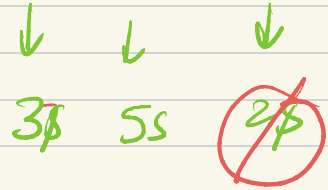
gives the value of the 1st settled promise

gives result as (val2)

↑ success case

failure case

promise.all([p1, p2, p3])



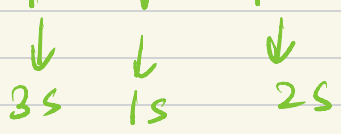
error is thrown after 1st failed promise.

2s → ERROR (p3)

Promise.any([p1, p2, p3])

wait for the 1st promise to get successful.

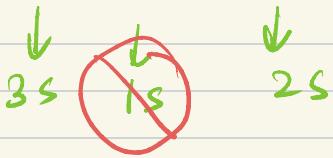
promise.all([p1, p2, p3])



→ val2

fail case

promise.all([p1, p2, p3])



ignores p2 and waits for next success

↳ val 3 returned

what if everything fails?

↳ returned result = [err1, err2, err3]

aggregated error
of all the three
errors

ASYNC

async function always returns a promise

↳ either directly returns a promise

OR

↳ returns a value that will automatically get wrapped as a promise and promise gets returned

```
11
12  // always returns a promise
13  async function getData() {
14    return "Namaste";
15  }
16
17  const dataPromise = getData();
18
19  dataPromise.then((res) => console.log(res));
20
```

↳ to use the promise.

if you have a predefined promise and you return it from the async function, the promise itself is resolved. no wrapping of promise inside a promise

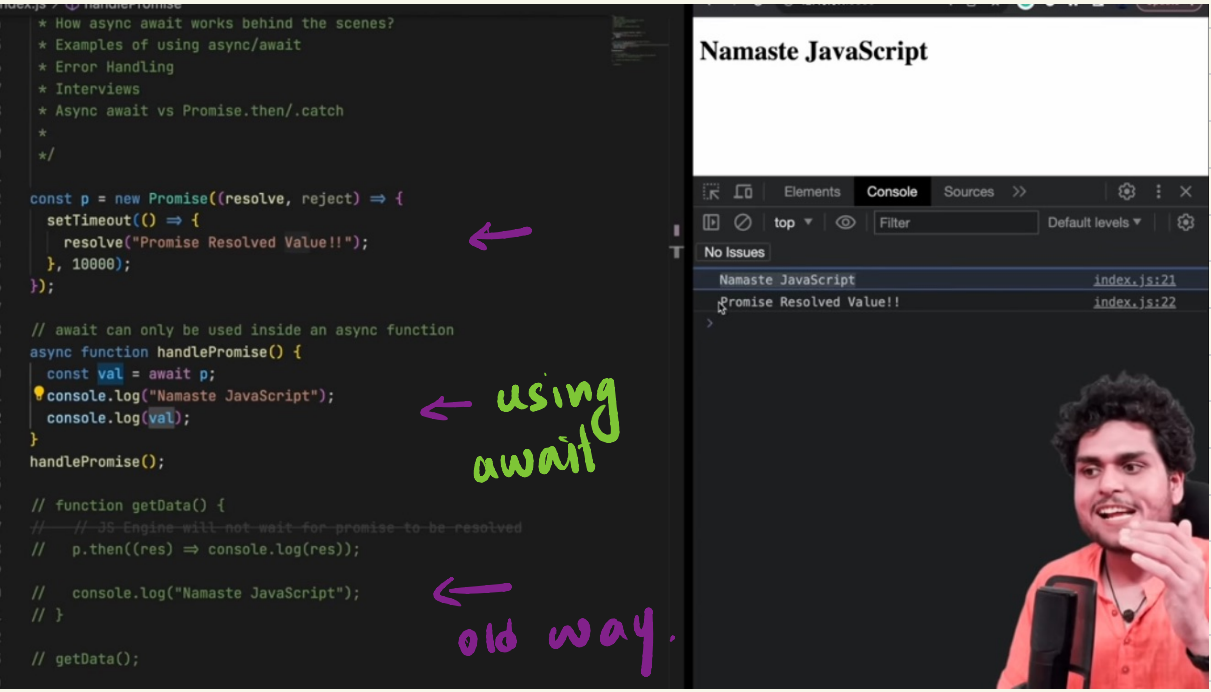
Async + Await → handles promises

Await is a keyword that can only be used inside an async function.

you can await a promise.

```
async function handlePromise() {
    const val = await p;
    console.log(val);
}
handlePromise();        // call async fun^n.
```

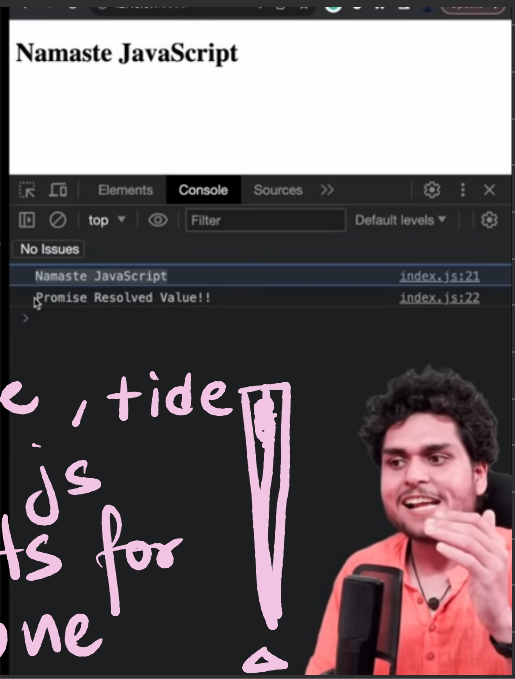
working of async await.



The old way

how did we handle promises before async await?

```
 * How async await works behind the scenes?
 * Examples of using async/await
 * Error Handling
 * Interviews
 * Async await vs Promise.then/.catch
 *
 */

const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Promise Resolved Value!!");
  }, 10000);
});

// await can only be used inside an async function
async function handlePromise() {
  const val = await p;
  console.log("Namaste JavaScript");
  console.log(val);
}
handlePromise();

// function getData() {
//   // JS Engine will not wait for promise to be resolved
//   p.then((res) => console.log(res));

//   console.log("Namaste JavaScript");
// }

// getData();
```



when the getdata function is called,

- the code is scanned line by line (GEC)
- the "p" has the promise and getdata has the whole function
- in the second scan, the JS engine will NOT wait for promise to be resolved while executing.

- it will quickly finish executing everything, treat the promise as a callback function, puts it in seperate execution contexts and prints ("Namaste Javascript") right away.
- When the global execution context is empty, the promise which has now completed its run (as timer ran out) after 10 sec is pushed into callback queue.

And from there is comes into the GEC, where it finishes executing and prints "Promise resolved value".

The Async -await way.

```
 * How async await works behind the scenes?
 * Examples of using async/await
 * Error Handling
 * Interviews
 * Async await vs Promise.then/.catch
 *
 */

const p = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Promise Resolved Value!!");
  }, 10000);
});

// await can only be used inside an async function
async function handlePromise() {
  const val = await p;
  console.log("Namaste JavaScript");
  console.log(val);
}
handlePromise();

// function getData() {
//   // JS Engine will not wait for promise to be resolved
//   p.then((res) => console.log(res));

//   console.log("Namaste JavaScript");
// }

// getData();
```

Namaste JavaScript

Elements Console Sources

top Filter Default levels

No Issues

Namaste JavaScript index.js:21

Promise Resolved Value!! index.js:22

after 10 s.

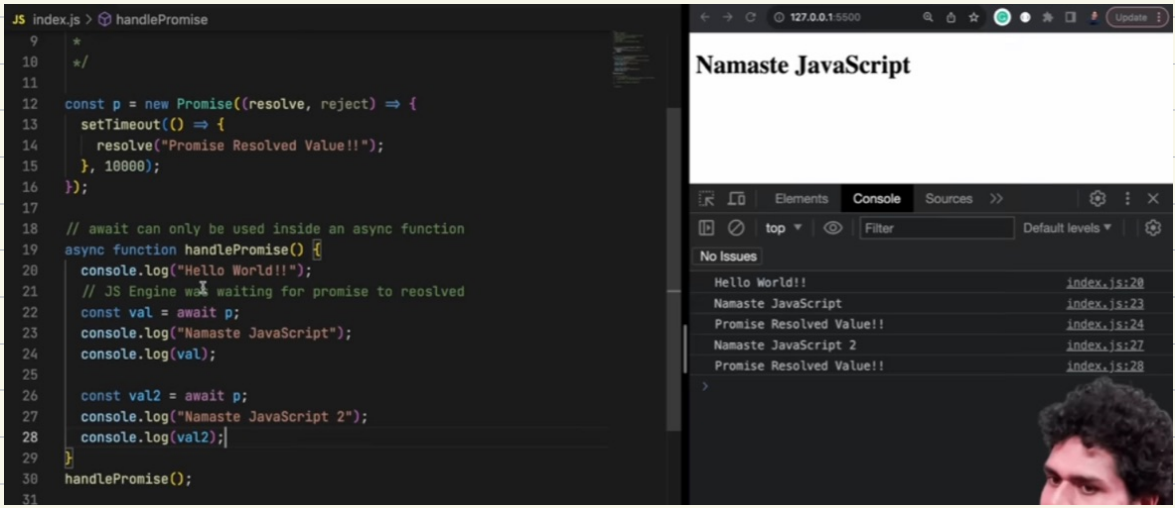
at the line const val = await p, the js engine waits for promise to get resolved and then executes the other stuff

↓

appears to wait!

if the promise is resolved twice, even then the await will wait for 10sec only and then prints everything at once.

parallel run.



both promises resolved after 10 s.

"this" Keyword

* in global space the value of "this" is window object → in browsers.

works differently in { strict mode / non-strict mode.

* Inside a function, it depends on strict or non strict mode

If the value of this keyword is

undefined → null

then, this will be replaced with global object.

Q. What is the value of "this" keyword inside a function?

- The value is undefined.
- It have this substitution in non strict mode in which this is replaced with the global object.

→ using strict mode ←

* this keyword value depends on how the function is called.

- if function is called normally,

  ex, `x();`

  then value of this → undefined.

- if function called using a reference

  `window.x();` → window object.

Q. difference between function and method

when you write a function inside an object then it's called a method.

```
const obj = {
    a : 10,
    x : function () {
        console.log (this.a)
    }
}
```

method ← x : function () {}

of an object obj

console.log (this.a) → prints 10 → obj.a.

obj.x() → {a : 10, x : f}

the object itself.

-Call apply bind ——

used when you want to share methods.

method.call → whatever you send this will override the value of this inside the method.

look into this video

Arrow function → dont have this

↓

Enclosing lexical context

↓ where the this object local, it will check its lexical context.

→ inside arrow functions this will be the global object.

this inside dom elements ⇒ reference to HTML element