Promises and Async Js

There are three big humps in Full Stack

1. The async nature of JS
2. React
3. JS to TS

What all we need to know Before we proceed to HTTP?

Definitely: callback

Good to know: JS architecture and Promises

Callback

```
function square(n) {
    return n * n;
  }

  function cube(n) {
    return n * n * n;
  }

  function sumOfSquares(a, b) {
    let square1 = square(a);
    let square2 = square(b);
    return square1 + square2;
  }

  function sumOfCube(a, b) {
    let square1 = cube(a);
    let square2 = cube(b);
    return square1 + square2;
  }
```

```
let ans1 = sumOfSquares(1, 2);
console.log(ans1);//5

let ans2 = sumOfCube(1, 2);
console.log(ans2);//9
```

The problem in this code is that we are violating the DRY(Don't repeat yourself) principle.It will become more complex if we want to find sum of two numbers' quad(x*x*x*x)

Now we will use callback.

```
function square(n) {
    return n * n;
  }

  function cube(n) {
    console.log("Cube called");
    return n * n * n;
```

```
  }

  function sumOfSomething(a, b, callbackFn) {
    let some1 = callbackFn(a);
    let some2 = callbackFn(b);
    return some1 + some2;
  }

  let ans = sumOfSomething(1, 2, cube);
console.log(ans);
```

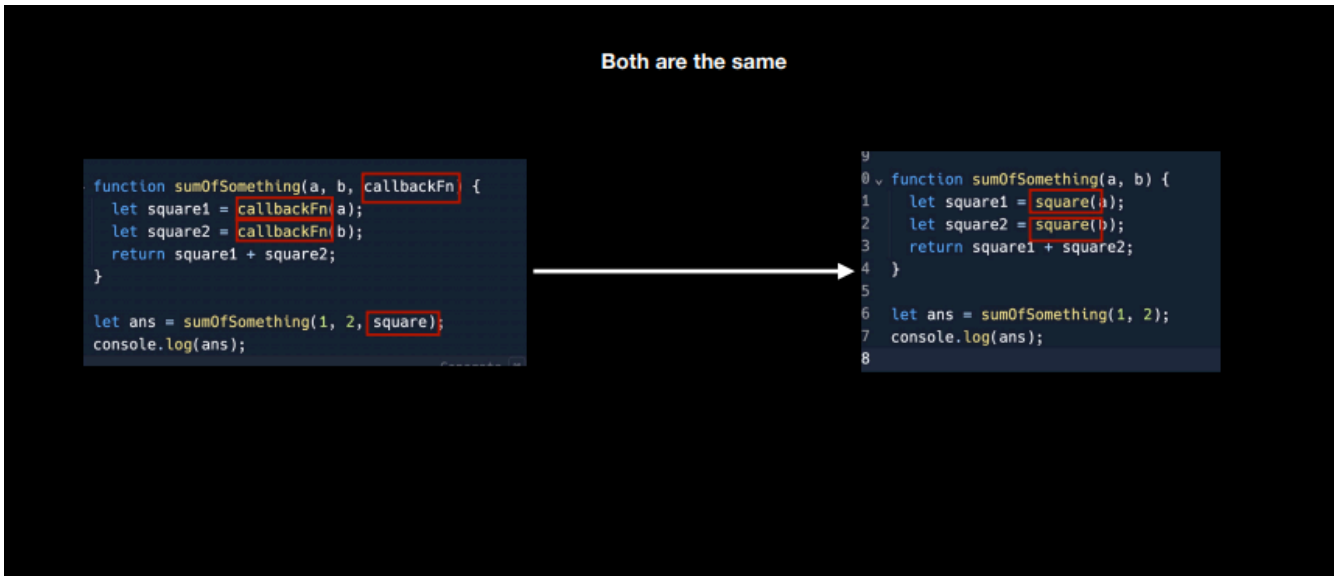
callbackFn we are passing function such that we can create a generic logic function

Output:

Cube called

Cube called

9



We don't need code in synchronous code.

Async function

Your JS thread doesn’t have access to everything immediately

There are some task it need to **wait** for

1. Reading a file.
2. Sending a network request
3. A deliberate timeout

Till now we have discussed normal function.

Eg:

Javascript asks for a file from OS. Os can’t immediately return it, it has to check whether the file is present or not, It is accessible or not

It is important to be async because the thread will be stuck and it won't proceed to other things.

```
function onDone(Content){
    console.log(Content);
}

readFile("a.txt",onDone)//async call
// we don't have to call the onDone function, It will be called.

console.log("Done");
for(let i=0;i<10;i++){

}
```

Eg

```
function onDone(){
    console.log("hi there");
}
setTimeout(onDone,1000);
```

```
console.log("after setTimeout");
```

We can do like after setTimeout runs and control does its thing like running a big loop and after it prints hi there. Thread doesn't get stuck when that something is asynchronous

Output:

after setTimeout

hi there.

Let’s see an async function call

Code:

```
const fs = require("fs");
// importing file system library
let a=1;
console.log(a);

fs.readFile("a.txt",'utf-8',(err,data) =>{
    // make sure your terminal is in the correct directory, utf-8 is the format of what are we reading
    // a.txt file and this code should be in the same directory
    // err is the first argument and data is the second argument that is api of this readFile function provided by fs
    console.log("data read from the file is ")
    console.log(data);
} )

let ans = 0;
for(let i=0; i<100;i++){
    ans += i;
}
```

```
console.log(ans);
```

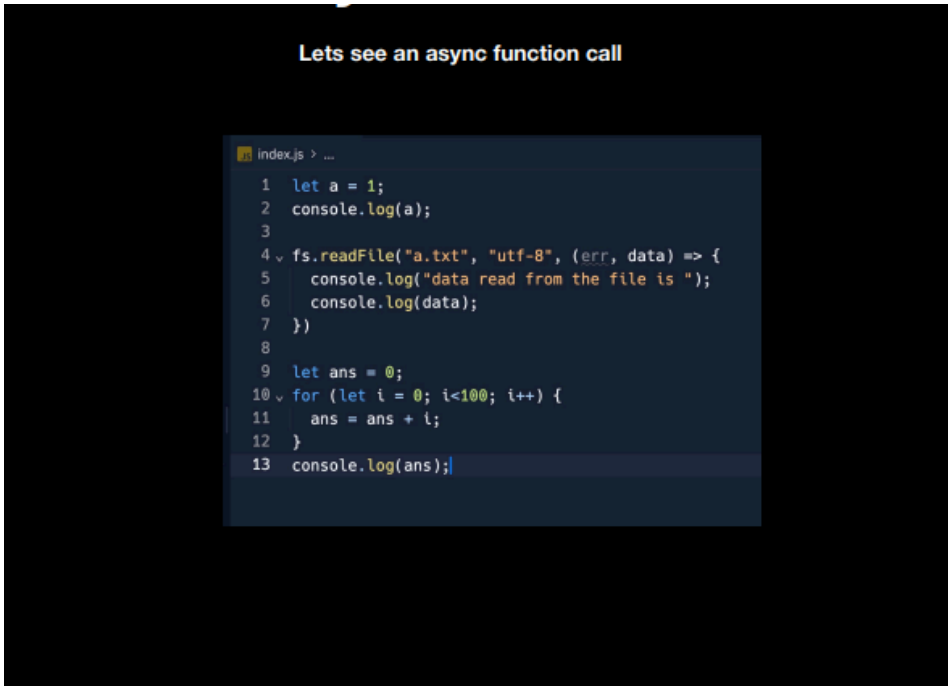
Output:

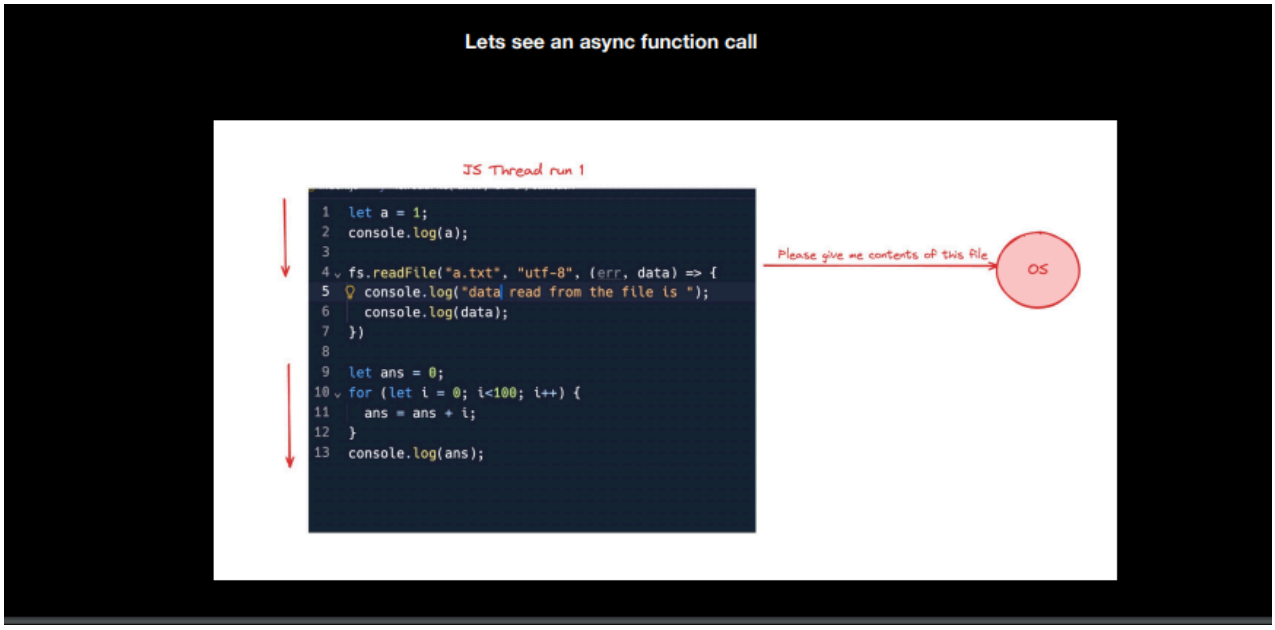
```
PS C:\Users\NTC\Desktop\Dev\Week2\asyncJs> node "c:\Users\NTC\Desktop\Dev\W
eek2\asyncJs\practice.js"
1
4950
data read from the file is
"hello i am speaking from the a.txt"
PS C:\Users\NTC\Desktop\Dev\Week2\asyncJs>
```

If there is some error

```
PS C:\Users\NTC\Desktop\Dev\Week2\asyncJs> node "c:\Users\NTC\Desktop\Dev\W
eek2\asyncJs\practice.js"
1
[Error: ENOENT: no such file or directory, open 'C:\Users\NTC\Desktop\Dev\W
eek2\asyncJs\a1.txt'] {
  errno: -4058,
  code: 'ENOENT',
  syscall: 'open',
  path: 'C:\\Users\\NTC\\Desktop\\Dev\\Week2\\asyncJs\\a1.txt'
}
data read from the file is
undefined
```

Now let's see the async function call

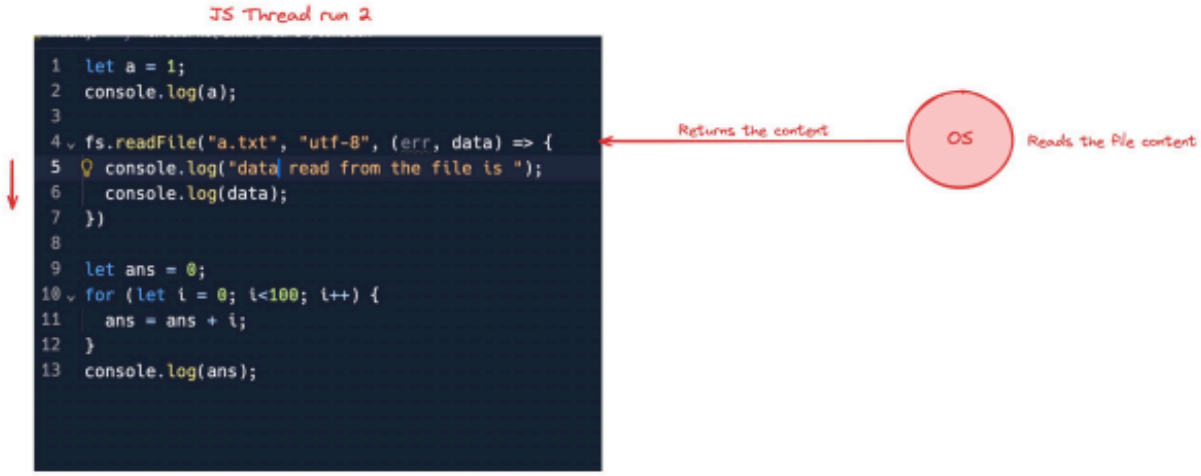




It to tell the OS please give me the content of the file, figure out if someone else is using the content, if then give me the content of file after some time, it is deleted it's fine to let us know and if it exists return it And then proceed

to do other operations while waiting for the OS to return(some kind of access to file content) file or error.

JS thread is idle, while OS is searching for file, confirming file exist or not , someone is using or not etc.



OS after finding file will call the callback and then java thread will start working on the function inside.

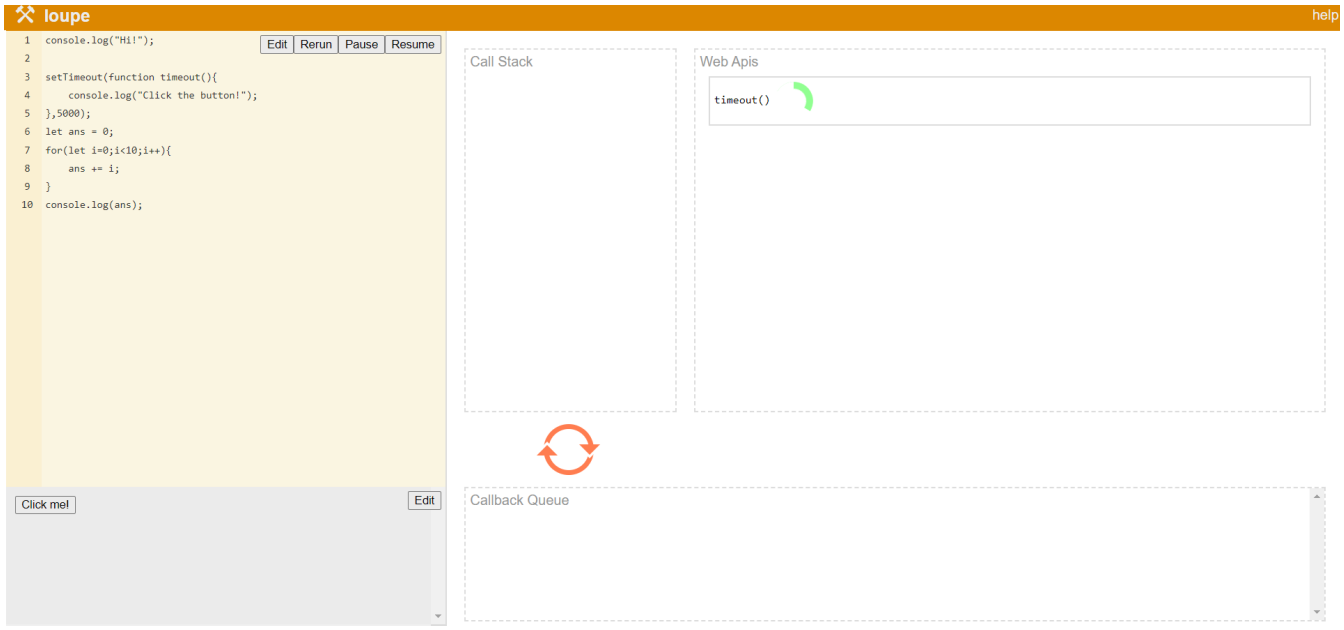
If you want to visualize it.

Control reach the line in this sequence

2 -> 4,6 -> 9,10,11,12,13 ->5(the content inside the function(callback) runs after 5 secs)

Callback will be called only when thread is idle.(all synchronous stuff is done)

Javascript Architecture



setTimeout() webApis , browser provide it as webApi

Similarly nodejs also provide.

Call stack
Web Apis
(Asynchronous function will be delegated here)
(Can maintain multiple clocks)
Event loop
Callback Queue

If we do a simple synchronous function, only Call stack is being used.
In the above example, if all the remaining synchronous code is dependent on the file content, we will write it inside the callback.

Q/Na
Use case of async function
Eg: Creating maggi
What if there is setTimeout() function inside another setTimeout()
The inner setTimeout() will be called in Web Apis section only when the outer setTimeout() is called from the callback queue to Call stack using Event loop

Promises

Syntactical sugar (cleaner way to write callback)
More readable way to write async function

How would you create your async function?

We make our function of our own for example

Function readAndWriteToFile(){

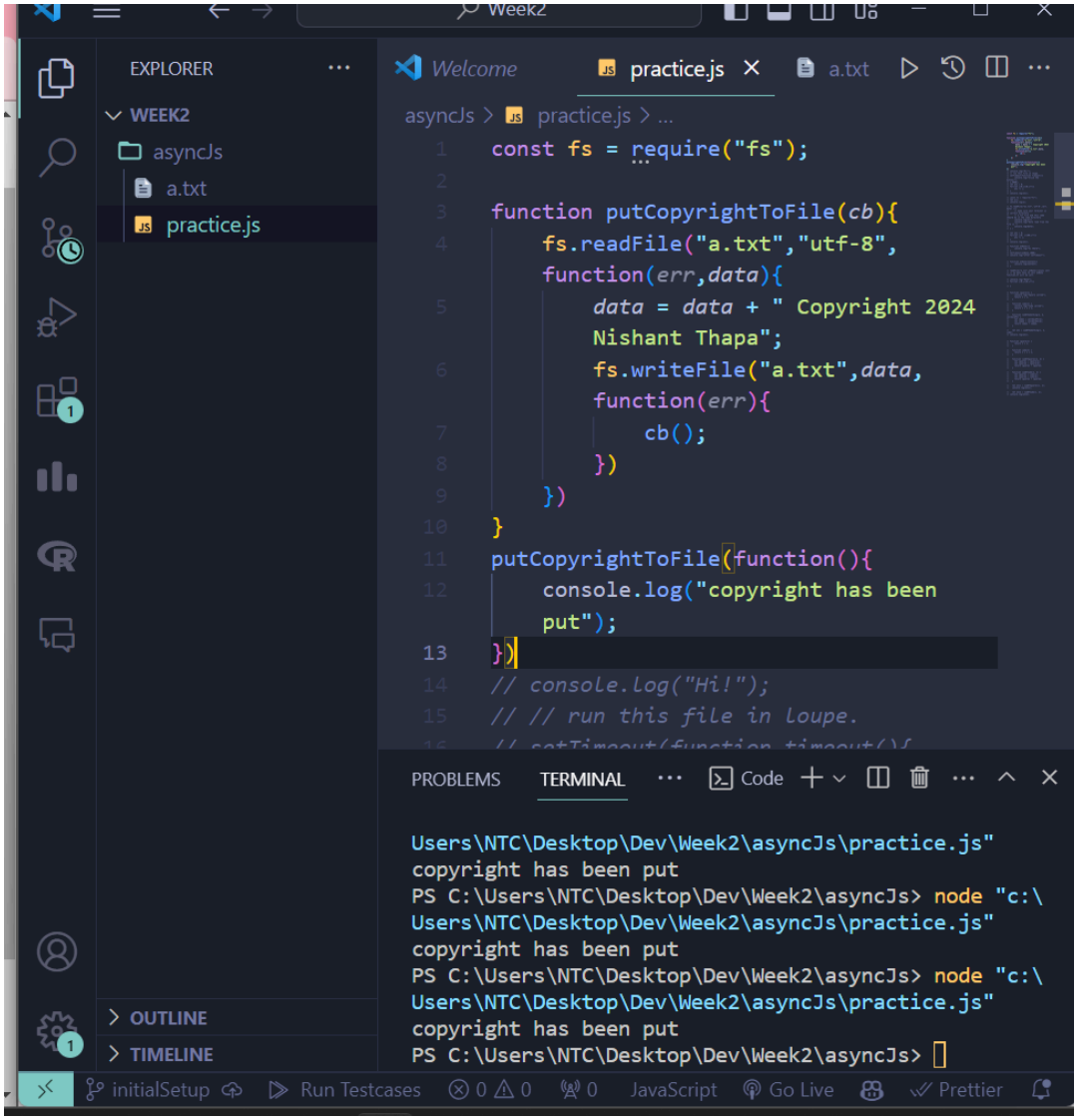
}

But this file under the hood uses readFile() and writeFile()

Wrapping JS async function and doing something.

Eg: Putting copyright text at the end

```
const fs = require("fs");

function putCopyrightToFile(cb){
    fs.readFile("a.txt","utf-8",function(err,data){
        data = data + " Copyright 2024 Nishant Thapa";
        fs.writeFile("a.txt",data,function(err){
            cb();
        })
    })
}
putCopyrightToFile(function(){
    console.log("copyright has been put");
})
```


Week2
EXPLORER
WEEK2
asyncJs
a.txt
practice.js
Welcome
practice.js
a.txt
asyncJs > practice.js > ...
const fs = require("fs");
function putCopyrightToFile(cb){
fs.readFile("a.txt","utf-8",
function(err,data){
data = data + " Copyright 2024
Nishant Thapa";
fs.writeFile("a.txt",data,
function(err){
cb();
})
})
}
putCopyrightToFile(function(){
console.log("copyright has been
put");
})
// console.log("Hi!");
// // run this file in loupe.
PROBLEMS
TERMINAL
Code
Users\NTC\Desktop\Dev\Week2\asyncJs\practice.js"
copyright has been put
PS C:\Users\NTC\Desktop\Dev\Week2\asyncJs> node "c:\
Users\NTC\Desktop\Dev\Week2\asyncJs\practice.js"
copyright has been put
PS C:\Users\NTC\Desktop\Dev\Week2\asyncJs> node "c:\
Users\NTC\Desktop\Dev\Week2\asyncJs\practice.js"
copyright has been put
PS C:\Users\NTC\Desktop\Dev\Week2\asyncJs>
OUTLINE
TIMELINE
initialSetup
Run Testcases
JavaScript
Go Live
Prettier

Approach #1

Wrapping another async function.

```
function myOwnSetTimeout(fn,duration){
    setTimeout(fn,duration);
}
myOwnSetTimeout(function(){
    console.log("Hi There");
},1000);
```

This approach uses a callback. You have created a function where other people can send a callback. This is good but could lead to **callback hell(** unnecessary indentation due to callbacks**).**

E.g. of callback hell

Create a function that logs something after 1s and then waits for 2 seconds to log another thing.

```
setTimeout(function(){
    console.log("Hi there");
```

```
    setTimeout(function(){
        console.log("How are you");
        setTimeout(function(){
            console.log("I am fine");
            setTimeout(function(){
                console.log("Thank you");
            },2000)
        },2000)
    ,2000})
})

// This way of writing callback is not same as above because the second
setTimeout() will run only when the first setTimeout() is completed.
// Hence it wont be exact 4 sec.
// setTimeout(function(){},2000)
// setTimeout(function(){},4000)
```

Very bad nesting. It doesn't look clean as it grows.

We want to code(this code will be synchronous, hence won't work for us we still want to have async nature otherwise JS thread will be stuck in waitFor(1000) and doesn’t does the rest of the work) something like this

```
waitFor(1000);
console.log("Hello");
waitFor(2000);
console.log("Hi");
waitFor(3000);
console.log("How are you");
```

Which is asynchronous

(asynchronous system are very common in backend)

Promises let us fix it(Callback hell).

Non-promisified	Promisified
return X	return Promise
Callback as input	callback Not as input

In Non-promisified, callback is a way to tell the caller that something has been done. Here caller will send a Callback as input

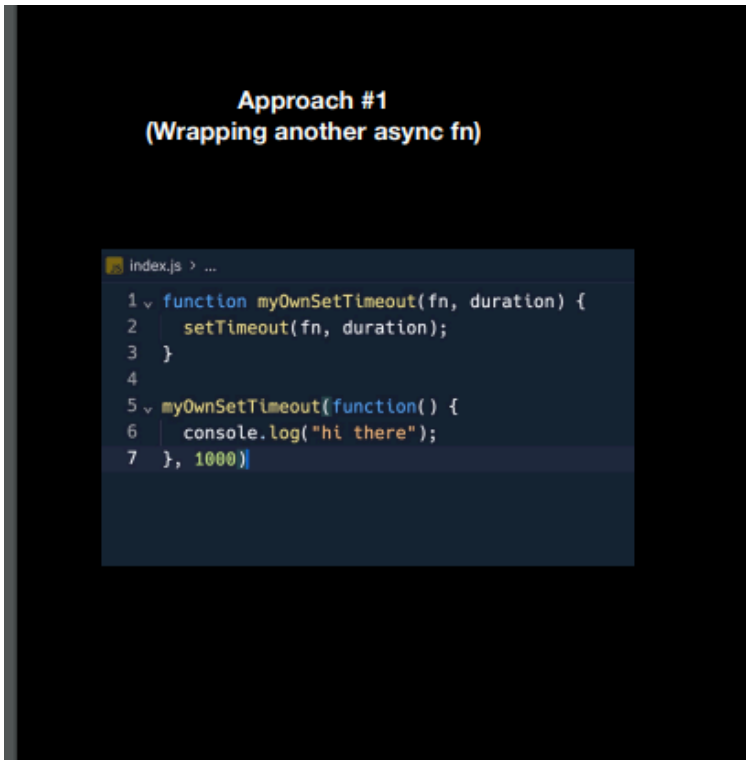
While in Promisified, the promise that it return is the way the caller will get a callback (what you want to do after I do my thing). Here caller will get a promise as input

Eg

```
// promisfy this async function
function myOwnSetTimeout(callback,duration){
    setTimeout(callback,duration);
}
// Calling this function
// myOwnSetTimeout(function(){},10000);

// Here we don't accept a callback
// Then how the person calling will run it after the duration is passed
// promisfy means it returns Promise
// Promise is just another class like the Date class
// const p = new Promise();
function promisifiedMyOwnSetTimeout(duration){
   const p = new Promise(function(resolve){
    // input resolve is also a callback function, that function has
resolve as the first argument
    setTimeout(function(){
        resolve();
    },duration);
    //After the duration is passed, we will call the resolve function
   })
```

```
    return p;
}
// async await syntax, promise chaining
const ans = promisifiedMyOwnSetTimeout(1000);
ans.then(function(){
    console.log("after set timeout");
})

// console.log(ans); //Promise { <pending> }
// pending means it is not resolved yet

// myOwnSetTimeout(function(){
//     console.log("after settimeout")
// },1000);
```

Approach #1



Approach #2

Approach #3

Q/Na

1. If readfile function as promisfy Vs async

```
const fs = require("fs");
// readFile is async function
fs.readFile("a.txt","utf-8",function(err,data){
    // control reach here
})
// if the read file was promisified
// readfile as promisfy
fs.readFile("a.txt","utf-8").then(function(err,data){

})
//  fs promisfy
```

2. We will use synchronous function unless we do
    - Do a network call
    - sleep/wait for some time
    - Read a file
    - Database call

      (we generally use a callback function, and we should know how to call a promisified function)
3. Promise has three state
    - pending state
4. Our Js thread is mostly idle, as it is very fast. Never will there be an expensive function that keeps the thread busy for minutes.
5. When we are doing readfile it is OS that is hitting kernel, doing calls, and returning
6. Browser and nodejs provide different webapis.

   Js in the browser cannot read your file, whereas nodejs can read a file

7. Promise also gets inside the callback queue, since it is a syntactical sugar
8. Why is promise used?

```
// callback => callback hell
// promise => allow us to use async/await (can only be used when a function is promisified)
```

9. If there is resolve, reject in promise then why should we use catch?
10. How to call a promisified function?
11. We want to do some tasks in this sequence
    - asyncFunction
    - Some sync task
    - Want my work which async return
    - Some other sync task

```
function someSyncTask1(){
    console.log("some sync Task 1");
}
function someSyncTask2(){
    console.log("some sync Task 2");
}
setTimeout(function(){
    someSyncTask2();
},1000)
someSyncTask1();
```

12. Examples

    How will I call one promisified setTimeout after another promisified setTimeout?

```
function promisifiedTimeout(duration){
    const p = new Promise(function(resolve){
        setTimeout(resolve,duration);
```

```
    })
    return p;
}
promisifiedTimeout(1000).then(function(){
    console.log("first is done")
    promisifiedTimeout(2000).then(function(){
        console.log("second is done")
        //I still see something like callback hell
    });
})
```

But this still looks something like callback hell.

```
// promise chaining
promisifiedTimeout(1000).then(function(){
    console.log("first is done")
    return promisifiedTimeout(2000)
    }).then(function(){
        console.log("second is done")
        //I still see something like callback hell
    });
```

13. Golang has subroutines hence it is not single-threaded

    In rust, there is thread.spawn(function(){

    })

14. resolve and .then()

```
const p = new Promise(function(resolve){
    resolve("hi there");
})
p.then(function(arg){
    //When resolve is called, then is called
})
```

Promise is like a global class, part of core JS syntax

15. Js still operates on the single core the task offloaded work on the other core (C++ core)
16. What is resolve inside the Promise

```
function fn(resolve){
    for(i=1;i<100;i++){
        a=a+i;
    }
    resolve(a);
}
const p = new Promise(fn);
// Promise class expects a function as an argument and that function takes an argument that is resolve.

// if p is promisified
p().then(function(arg){

})

//If p is not promisified a simple callback
p(function(arg){

})
```

17. Can we return array of Promises

```
function getPromises(){
    let p1 = new Promise();
    let p2 = new Promise();
    return [p1,p2];
    // returning array of promises
}
const x = getPromises();
x[0].then(function(){})
```

18. Handling errors in Promises

```
function myOwnPromisifiedTimeout(){
    return new Promise(function(onDone,onError){
        onError();
    })
}
const p = myOwnPromisifiedTimeout();
p.then(function(){
    //When someone calls onDone function the control reaches here
})
p.catch(function(){
    //When someone calls onError function the control reaches here
})
```

You are passing this function as an argument (which is inside Promise:

```
function(onDone,onError){
        onError();
    }
```

), someone else is going to call this function, and whenever someone else calls that function they will pass two arguments which are function themselves.

Which will be called Inside the Promise.

19. Can we read files synchronously

```
// We are read file synchronously
const fs = require("fs");
let data = fs.readFileSync("a.txt","utf-8")
console.log(data);
console.log("after reading file");
```

Yes, we can but we should, because it is not wise to make Js thread get stuck there and do nothing else.

Reading of file may take many seconds and at that time thread is idle and not doing other expensive operations

20. Some common mistakes

```
function sum(a,b){
    return a+b;
}
setTimeout(sum(1,2),1000);
// 'ERR_INVALID_CALLBACK'
//This is the same as calling
// setTimeout(3,1000);
// but setTimeout expects a callback function as the first argument
//The correct way is
setTimeout(function(){
    console.log(sum(1,2));
},1000);
```

21. Google Chrome can run multiple threads
22. Promise class Copy

Check out in around **2hr:55mins** in 2.1 Video

Example of Promise:

Seeing where the control goes when we use Promise

```
console.log("at the top")
function promisifiedTimeout(){
    console.log("function called")
    return new Promise(function(resolve){
        console.log("inside promise callback")
        setTimeout(function(){
            console.log("inside settimeout")
            resolve("Dude Goodnight, chill broooo!!");
        },5000)
    })
}
console.log("im at the middle of the code")
promisifiedTimeout().then(function(value){
```

```
    console.log("inside promisified then")
    console.log(value);
})
```

Output:

```
eek2\asyncJs\practice.js"
at the top
im at the middle of the code
function called
inside promise callback
inside settimeout
inside promisified then
Dude Goodnight, chill broooo!!
PS C:\Users\NTC\Desktop\Dev\Week2\asyncJs>
```

Control reaches top(logs **“at the top”**)

Then it defines the promisifiedTimeout() function but nothing is logged.

Then it logs (**“im at the middle of the code”**)

Then we call promisifiedTimeout() control logs (**“function called”**)

promisifiedTimeout() function returns promise, then whatever is return on it we will do .then(), but it still hasn’t been called but it should be called soon.

Control then is inside the promise and then it logs (**“inside promise callback”**)

Then it gets inside the setTimeout() function(wait for 5 seconds, basically setTimeout() goes inside the webApis section of JS architecture and when it is called) and then it logs (**“inside settimeout”**) then resolve happens and then control reaches inside .then() and then it logs (**“inside promisified then”**) and then the value is logged (**“Dude Goodnight, chill broooo!!”**)