# Promises

Promises are special JS objects.

- how to create these objs
- how to consume
- properties

# How promises work behind the scene??

The promise object we create has 4 major properties

1) status / State
2) value
3) onFulfillment
4) onReject

* status/state → status shows current promise status

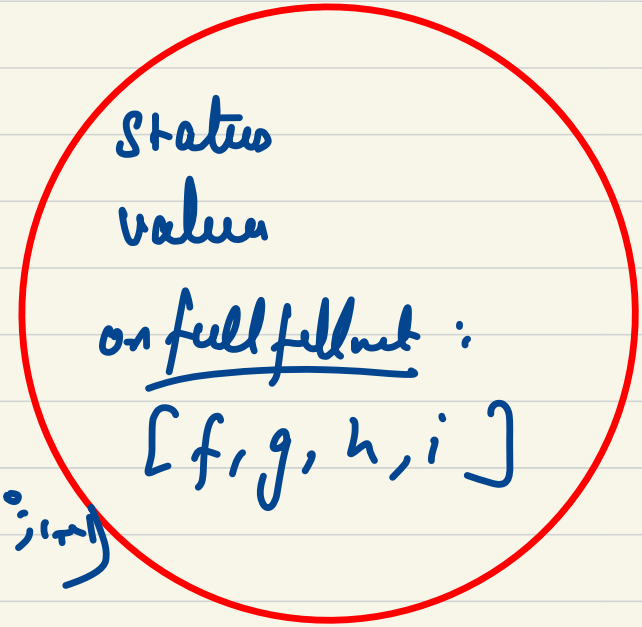
1) pending state
2) fulfilled state → success
3) rejected state → error

* value → when status of the promise is pending, this value property is undefined. the moment promise is resolved, the value property is updated from undefined to the new value (this value we can consider as the returned value / resolved value)

status → fulfilled

So the value property acts like a placeholder till the time promise finishes.

* onfulfillment → This is an array, which contains functions that we attach to our promise object. (To a promise object we can attach some func^n using .then() method). When the value property is updated from undefined, to the new value, JS gives chance to these attached func^n one by one with the value property as their argument (if there is no piece of code in the call stack & global code is gone)




for (i=0; i<10^10; i++)
{

return new Promise (function (resolve, reject) {

3)

Promise constructor

this constructor takes callback as argument

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

// write here

});
```

→ To create a promise call the promise constructor.

→ the promise constructor takes a callback as an argument.

→ the callback passed inside constructor, expects 2 arguments resolve, reject func?

→ then inside the callback write your logic

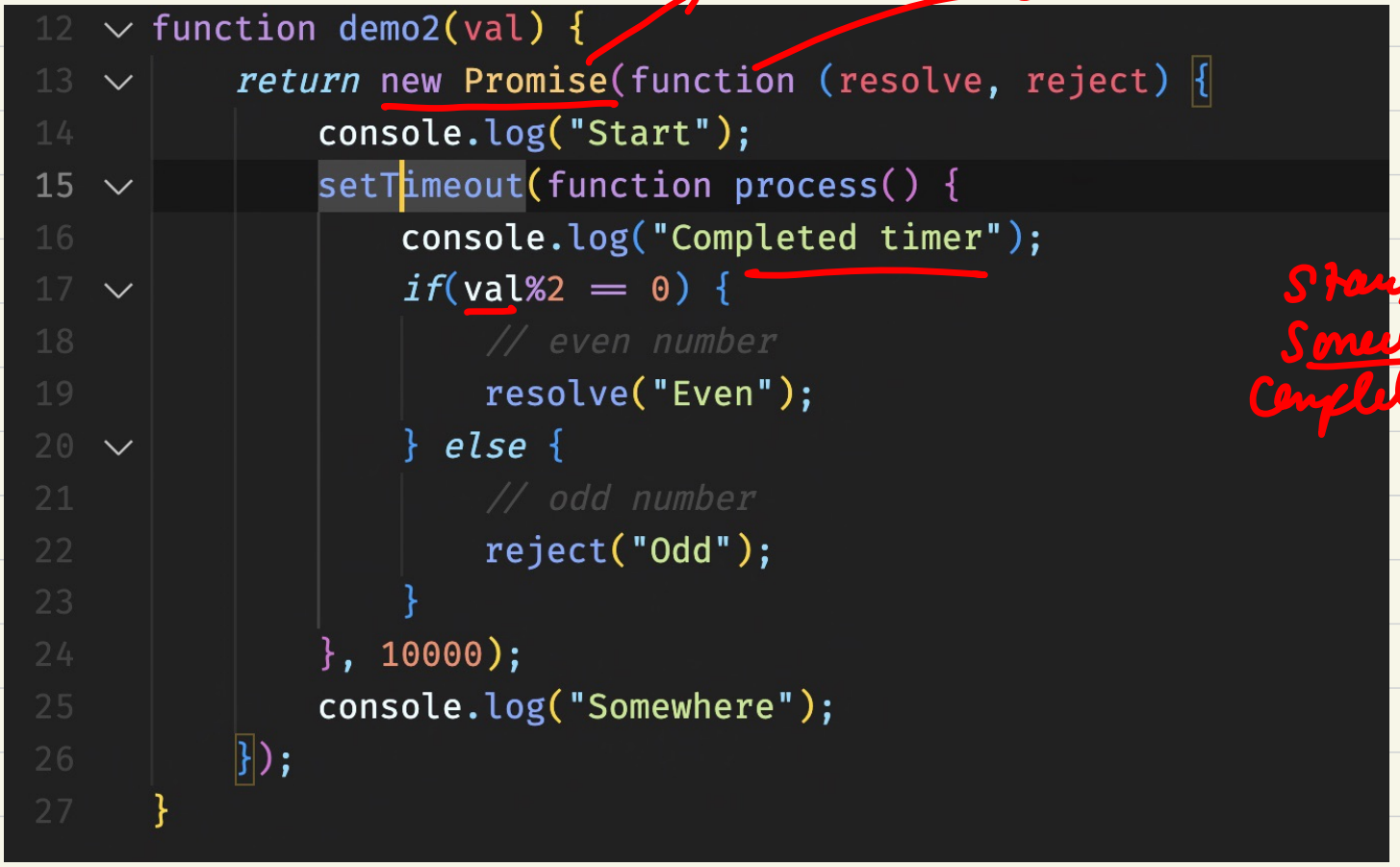
→ if you want to return something on success,
then call resolve func with whatever value
you want to return.

Q» When do we consider a promise fulfilled??

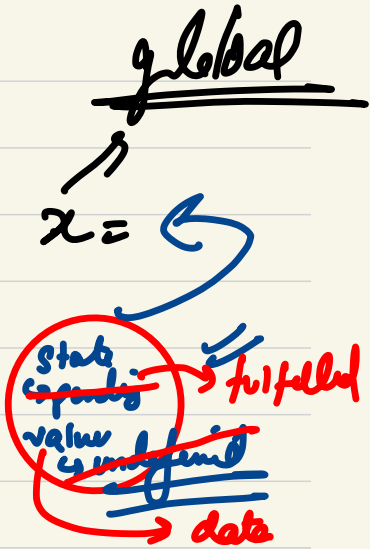
→ if we call resolve ( ) func", we consider it fulfilled.

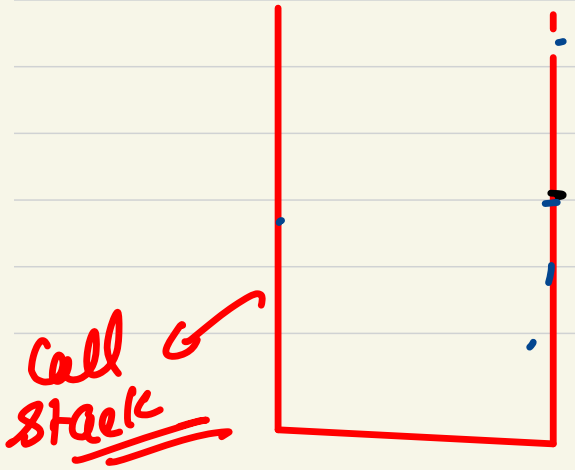
→ we consider it rejected if we call reject ();

# Creation of a promise obj is sync.


function demo2(val) {
return new Promise(function (resolve, reject) {
console.log("Start");
setTimeout(function process() {
console.log("Completed timer");
if(val%2 == 0) {
// even number
resolve("Even");
} else {
// odd number
reject("Odd");
}
}, 10000);
console.log("Somewhere");
});
}

```
function fetchData(url) {
    return new Promise(function (resolve, reject) {
        console.log("going to start the download");
        setTimeout(function process() {
            let data = "Dummy downladed data";
            console.log("download completed");
            resolve(data);
        }, 10000);
        console.log("Timer to mimic download started");
    });
}

console.log("Starting the program");
console.log("We are expecting to mimic a downloader");
x = fetchData("www.google.com");
console.log("New promise object created successfully, but downloading still going on");

```



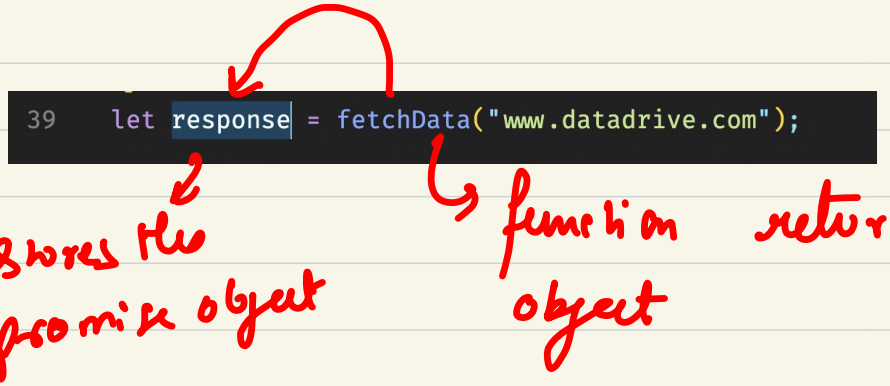

Starting the program
we are expecty ----downloade
going to start download
timer to mimic download started
New prom obj ..... on.
download completed

# Consuming a promise

The promise consumption is the main beauty, using which we will avoid inversion of control.

Whenever we call a function, returning a promise, we will get a promise object which is like any JS object that we can store in a variable.

→ Now, the question, will JS wait here ??




Q. Will JS wait here for promise to be resolved if it involves any async piece of code ?!

→ if creation of promise involves sync piece of code it will wait, otherwise won't.

```
function fetchData(url) {
    return new Promise(function (resolve, reject) {
        console.log("Started downloading from", url);
        // setTimeout(function processDownloading() {
        //     let data = "Dummy data";
        //     console.log("Download completed");
        //     resolve(data);
        // }, 7000);
        for(let i = 0; i < 10000000000; i++) {}
        resolve("dummy data");
    });
}
```

this callback is having a long sync piece of code, so JS will have to wait for promise object creation.

And just after the for loop, we also resolve the promise so we get a resolved promise.

```
6   function fetchData(url) {
7       return new Promise(function (resolve, reject) {
8           console.log("Started downloading from", url);
9           setTimeout(function processDownloading() {
10              let data = "Dummy data";
11              console.log("Download completed");
12              resolve(data);
13          }, 7000);
14      });
15  }
```

→ Promise object will get created easily as there is no blocking piece of code, but initially it will be pending. As the fulfillment happens after a time of 7 sec.

Now technically, when promise gets resolved, we have to execute some functions.

→ We can use .then() function on the promise object, to bind the functions we want to execute once we fulfill a promise.

The .then() function takes function as an argument that we want to execute after promise fulfills, and the argumen function takes value property as parameter.

```
function fetchData(url) {
    return new Promise(function (resolve, reject) {
        console.log("Started downloading from", url);
        setTimeout(function processDownloading() {
            let data = "Dummy data";
            console.log("Download completed");
            resolve(data);
            console.log("hello");
            // resolve("sanket");// these lines will not b
            // resolve(12345);
        }, 7000);
    });
}
```

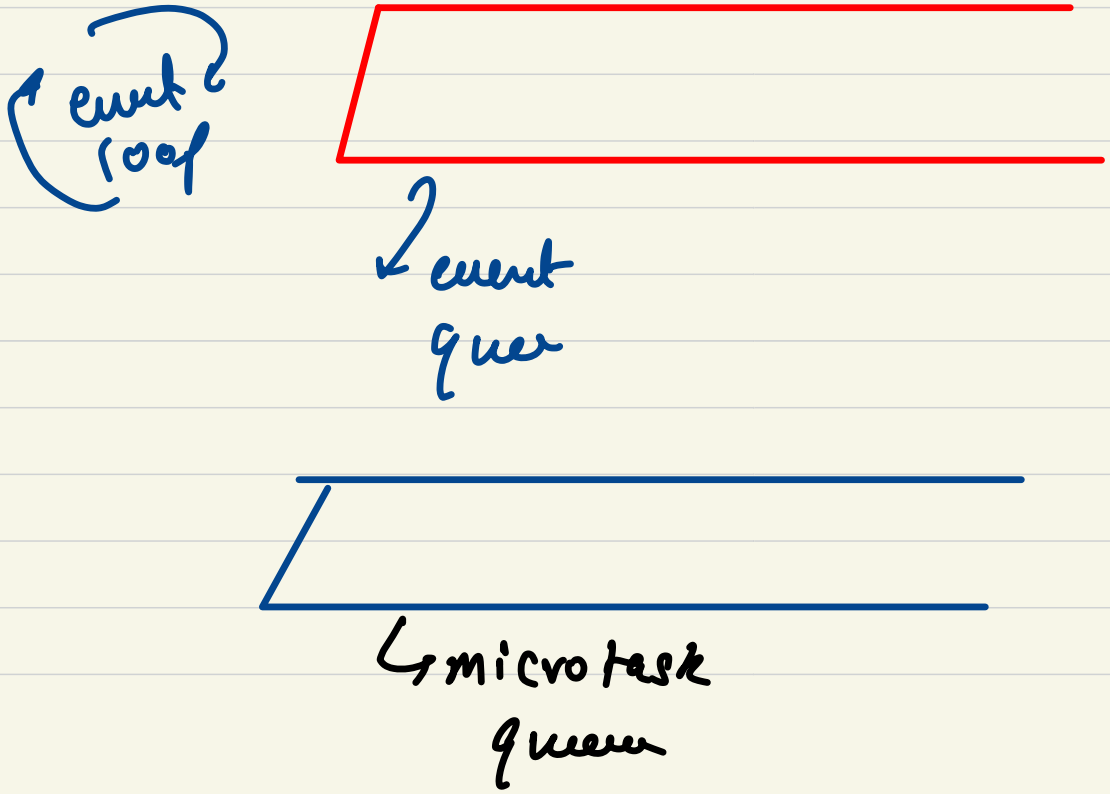

hello

```
download promise = fetchdata ("www.google.com");

x = download promise . then ( function f (value) {
        console.log (value)
        return "sanket";
})
```

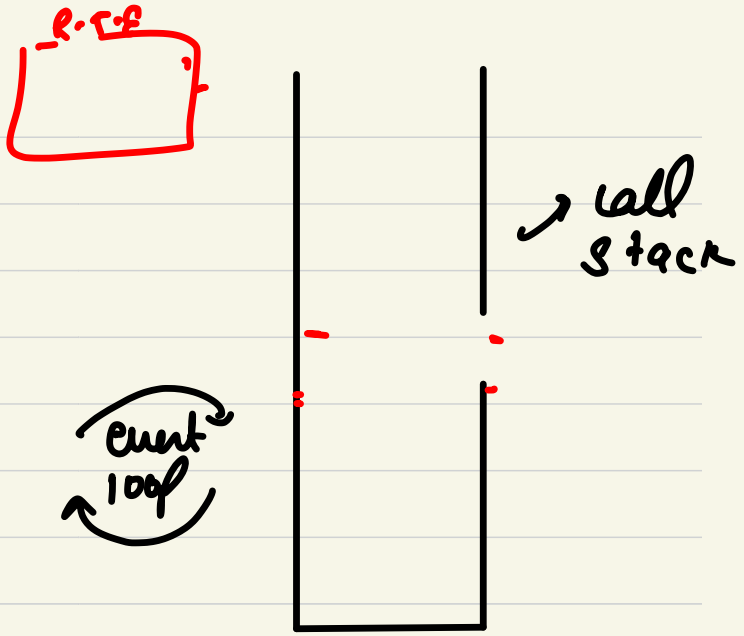
x ← new promise

the .then function itself returns a new promise.

```
let downloadPromise = fetchData("www.datadrive.com");
downloadPromise
.then(function processDownload(value) {
    console.log("donwloading done with following value", value);
    return value;
})
.then(function processWrite(value) {
    return writeFile(value);
})
.then(function processUpload(value) {
    return uploadData(value, "www.drive.google.com");
});
```



event loop
event queue
microtask queue

```js
console.log("Start of the file");

setTimeout(function timer1() {
    console.log("Timer 1 done");
}, 0);

for(let i = 0; i < 10000000000; i++) {
    // something
}

let x = Promise.resolve("Sanket's promise");
x.then(function processPromise(value) {
    console.log("Whose promise ? ", value);
});

setTimeout(function timer2() {
    console.log("Timer 2 done");
}, 0);

console.log("End of the file");
```



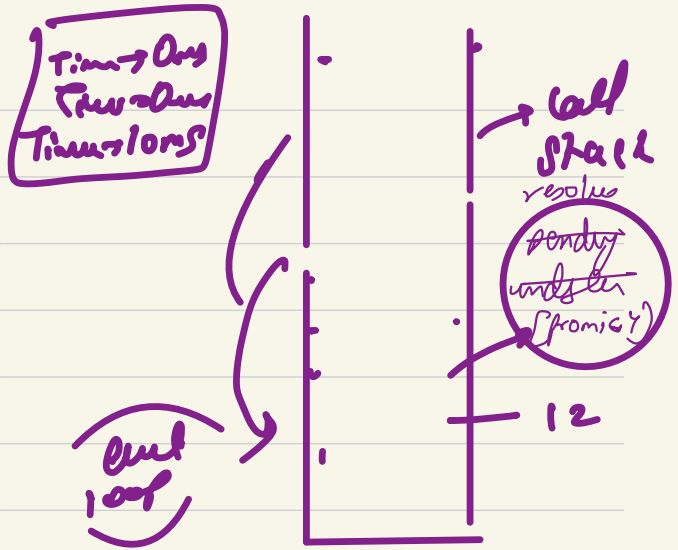

Start of file
End of file
Whose promise, Sanket's promise
Timer 1 done
Timer 2 done
x = resolved value
Sanket's promise
onfulfillment → processPromise

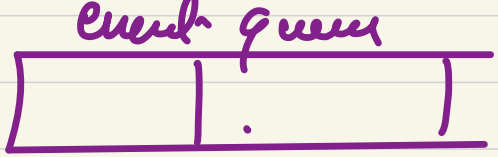
Microtask queue has a higher priority.

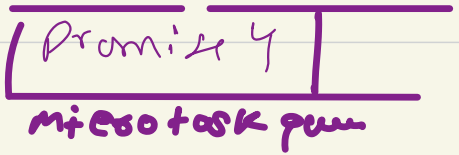
promise → callbacks → microtask queue

normal callbacks → event queue.

```js
function dummyPromise() {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            resolve("Timer's promise")
        }, 10000);
    });
}
console.log("Start of the file");

setTimeout(function timer1() {
    console.log("Timer 1 done");
    let y = dummyPromise();
    y.then(function promiseY(value) {
        console.log("Whose promise ?", value);
    });
}, 0);


let x = Promise.resolve("Sanket's promise");
x.then(function processPromise(value) {
    console.log("Whose promise ? ", value);
});

setTimeout(function timer2() {
    console.log("Timer 2 done");
}, 0);

console.log("End of the file");
```

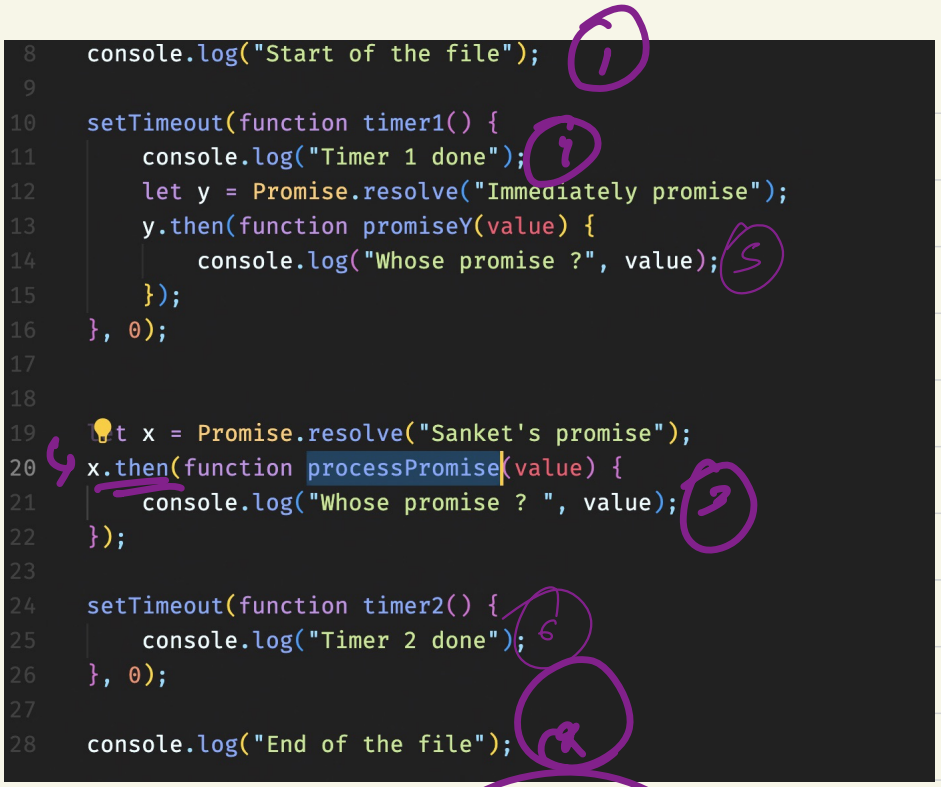


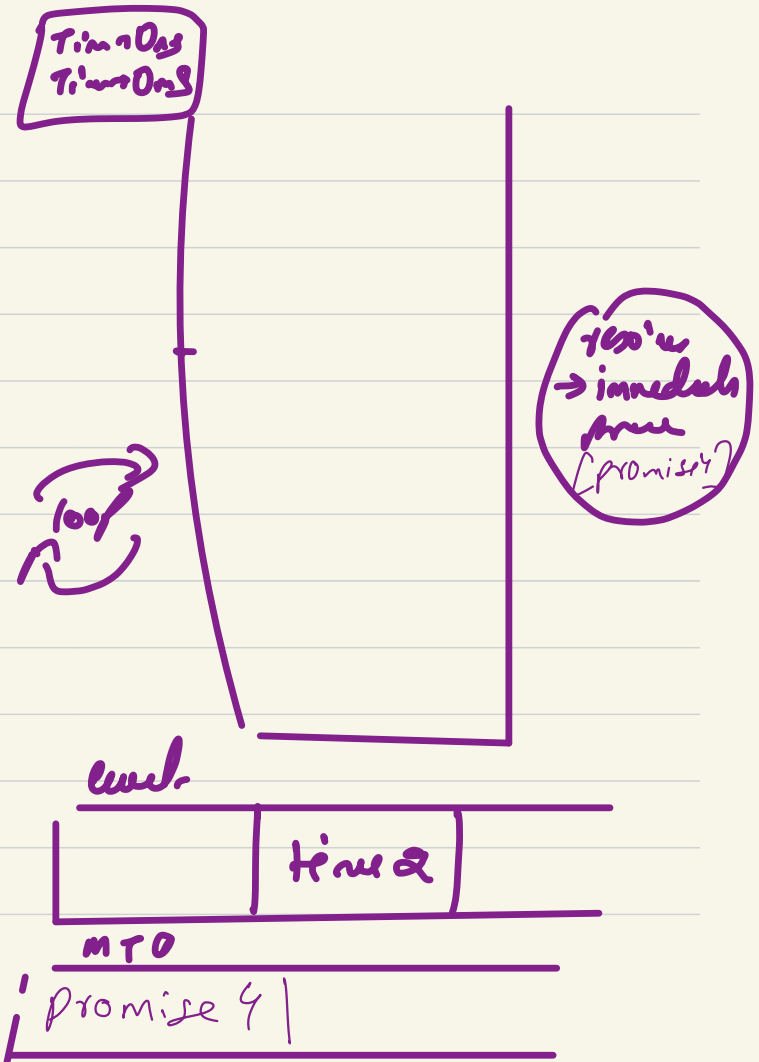



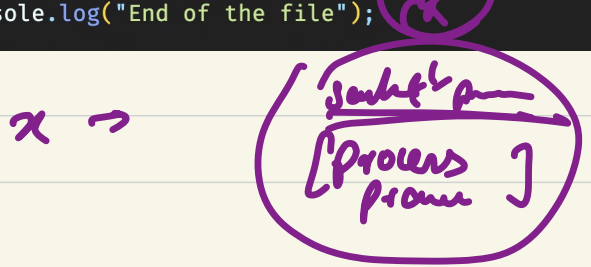




x = resolved value → Sanket's promise

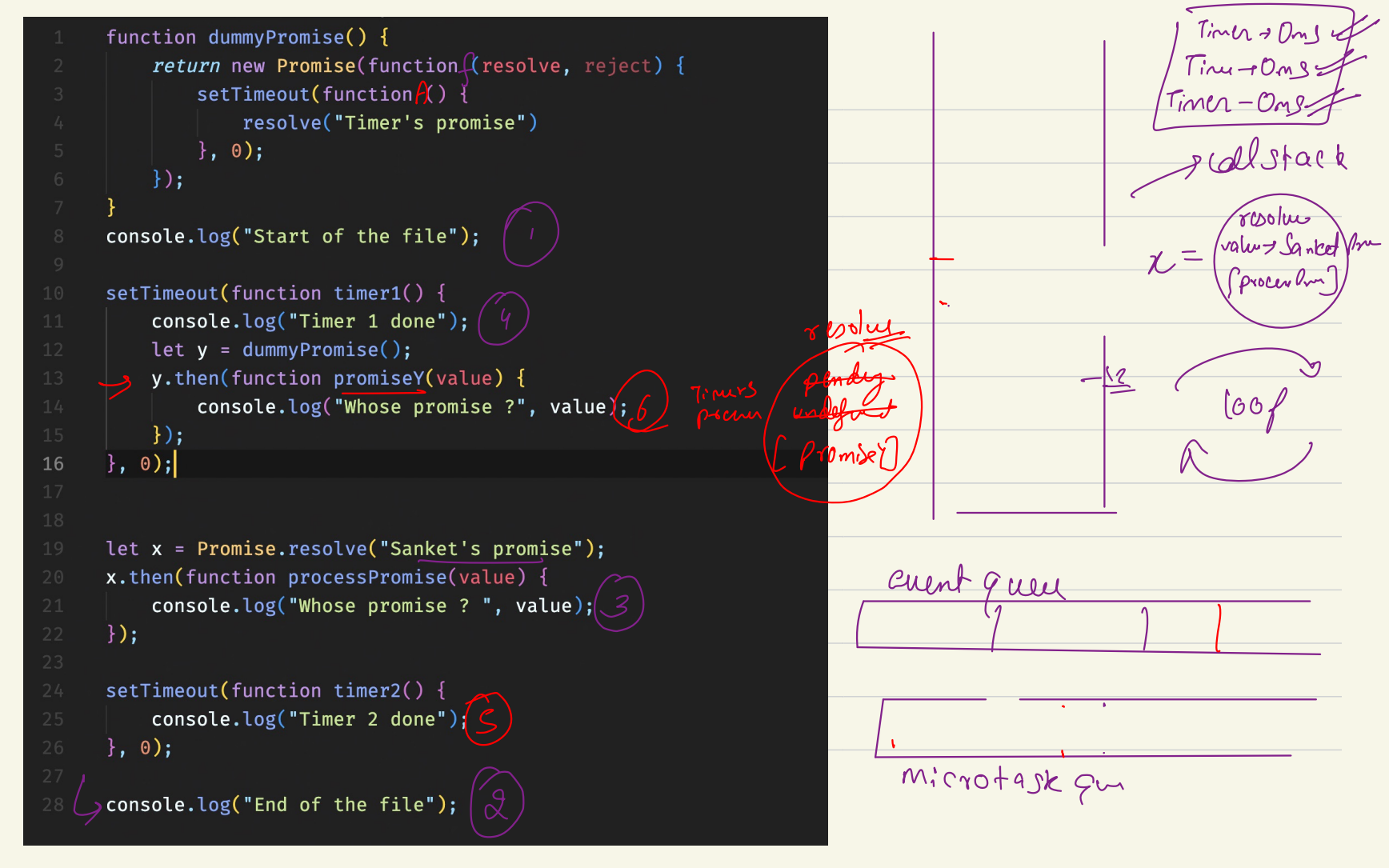
```
8   console.log("Start of the file");
9
10  setTimeout(function timer1() {
11      console.log("Timer 1 done");
12      let y = Promise.resolve("Immediately promise");
13      y.then(function promiseY(value) {
14          console.log("Whose promise ?", value);
15      });
16  }, 0);
17
18
19  let x = Promise.resolve("Sanket's promise");
20  x.then(function processPromise(value) {
21      console.log("Whose promise ? ", value);
22  });
23
24  setTimeout(function timer2() {
25      console.log("Timer 2 done");
26  }, 0);
27
28  console.log("End of the file");
```





```js
function dummyPromise() {
    return new Promise(function f(resolve, reject) {
        setTimeout(function A() {
            resolve("Timer's promise")
        }, 0);
    });
}
console.log("Start of the file");

setTimeout(function timer1() {
    console.log("Timer 1 done");
    let y = dummyPromise();
    y.then(function promiseY(value) {
        console.log("Whose promise ?", value);
    });
}, 0);


let x = Promise.resolve("Sanket's promise");
x.then(function processPromise(value) {
    console.log("Whose promise ? ", value);
});

setTimeout(function timer2() {
    console.log("Timer 2 done");
}, 0);

console.log("End of the file");
```



# async & await

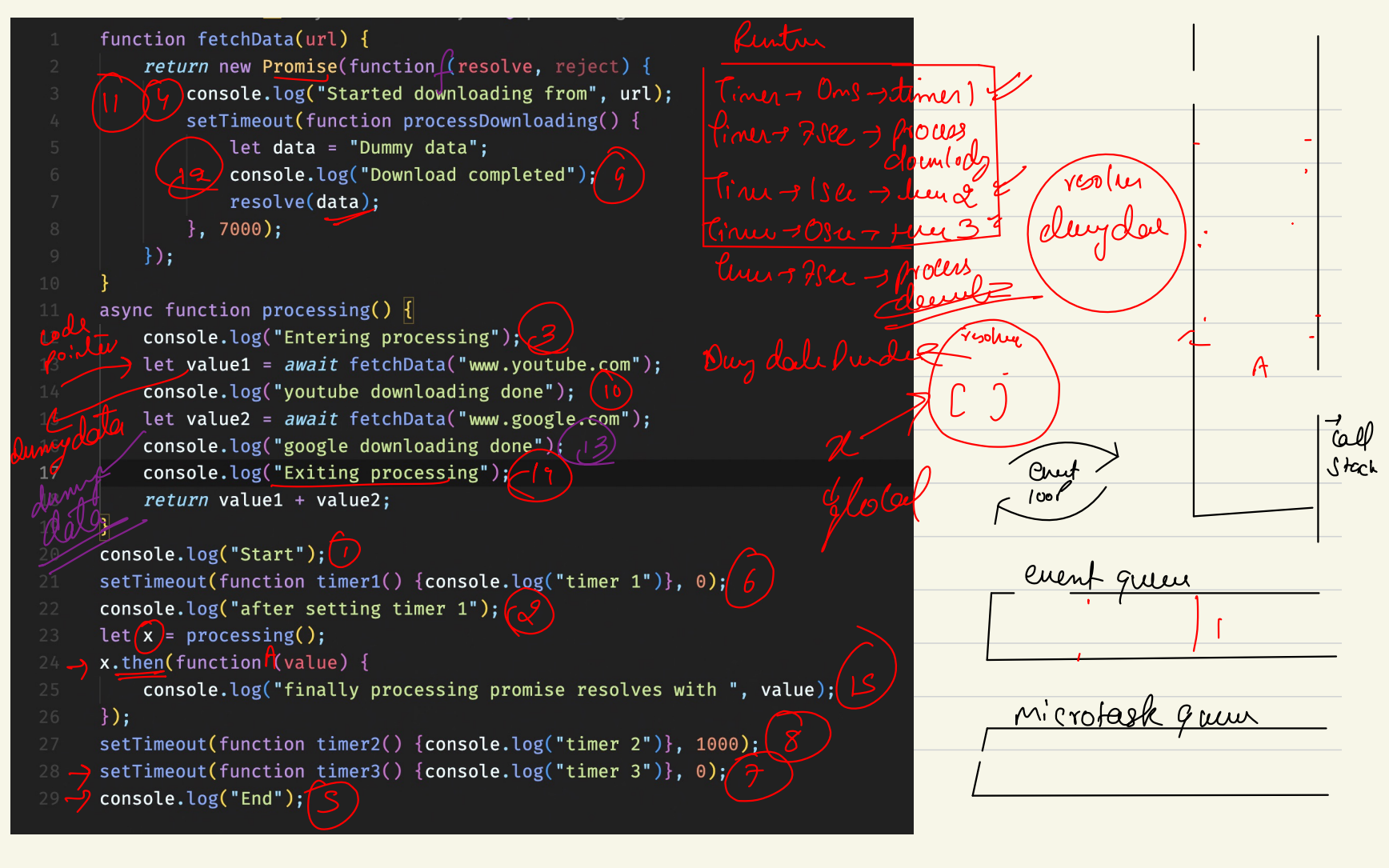
↳ we can declare a function **async**.

→ if you declare a func$^{n}$ async, it does the following →

1. It allows the use of await keyword.
2. if you declare a function async, it allows consumption of a promise using await.
3. an async function always converts your

returns value to a promise.

```
function fetchData(url) {
    return new Promise(function (resolve, reject) {
        console.log("Started downloading from", url);
        setTimeout(function processDownloading() {
            let data = "Dummy data";
            console.log("Download completed");
            resolve(data);
        }, 7000);
    });
}
async function processing() {
    console.log("Entering processing");
    let value1 = await fetchData("www.youtube.com");
    console.log("youtube downloading done");
    let value2 = await fetchData("www.google.com");
    console.log("google downloading done");
    console.log("Exiting processing");
    return value1 + value2;
}
console.log("Start");
setTimeout(function timer1() {console.log("timer 1")}, 0);
console.log("after setting timer 1");
let x = processing();
x.then(function (value) {
    console.log("finally processing promise resolves with ", value);
});
setTimeout(function timer2() {console.log("timer 2")}, 1000);
setTimeout(function timer3() {console.log("timer 3")}, 0);
console.log("End");
```



```
Start
after setting timer 1
Entering processing
Started downloading from www.youtube.com
End
timer 1
timer 3
timer 2
Download completed
youtube downloading done
Started downloading from www.google.com
Download completed
google downloading done
Exiting processing
finally processing promise resolves with  Dummy dataDummy data
```

→ Inside async func^n thus look sync, but overall its async.