# <u>Promise Constructor</u>

$$\begin{tikzcd}
 & \text{value} \arrow[r] & \text{default} \arrow[r] & \underline{\text{undefined}} \\
\underline{\text{Promises}} \arrow[ur] \arrow[dr] & & & \\
 & \text{State} \arrow[r] & \text{default} \arrow[r] & \underline{\text{Pending}}
\end{tikzcd}$$

```
new Promise ( function exec ( res , rej ) {



})
```

→ At the time when the constructor generates a new Promise object, it also generates a pair of func, called as resolve & reject.

→ generally the executor callback, wraps some async/sync operation

→ the executor is called sync.

## Consuming A Promise

assume this returns a promise.

```
let p = fetch("          ");
```

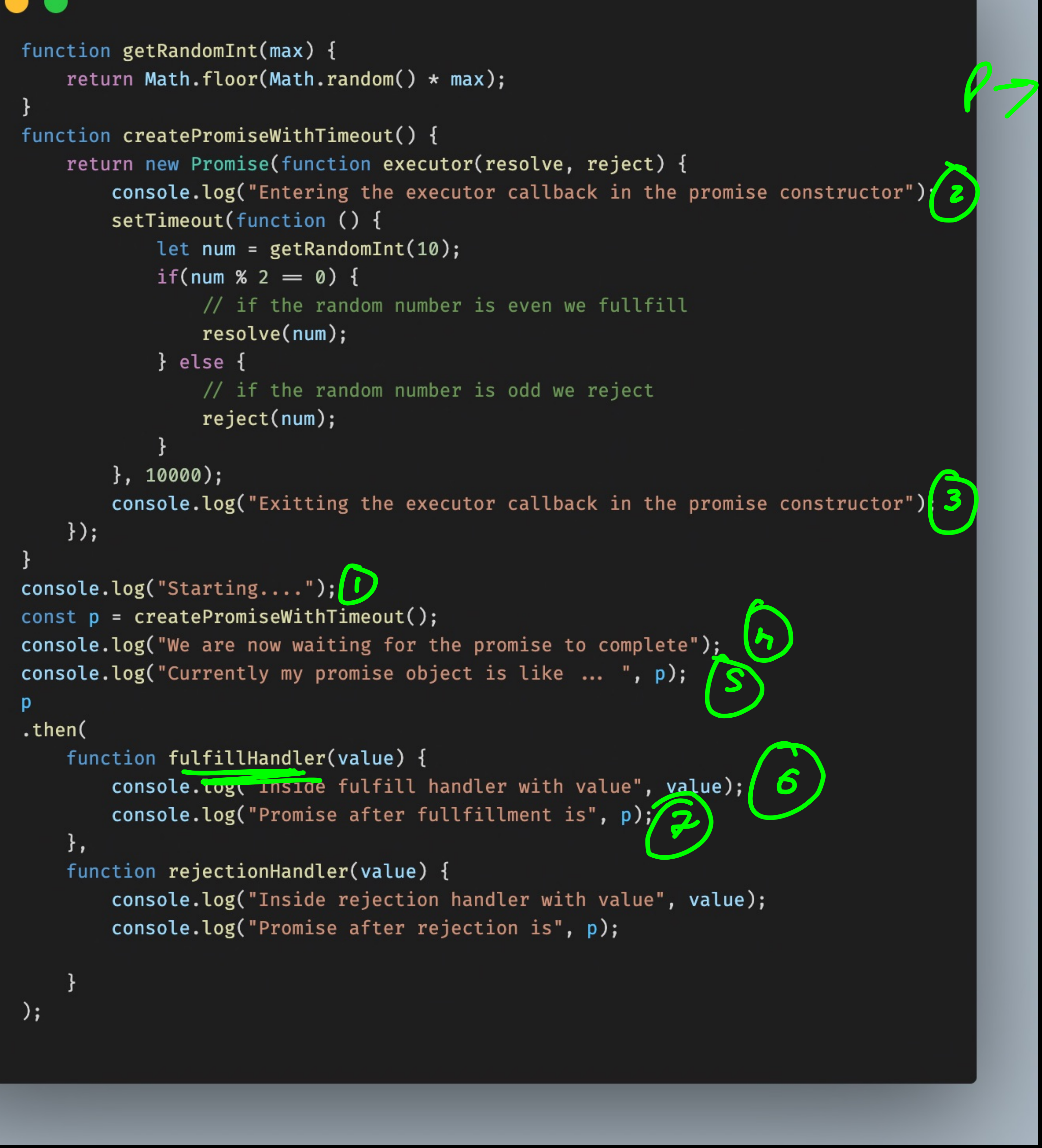
↳ attach the functionality that we need to execute once the promise is fulfilled or rejected

p. then (fulfillmenthandler, rejectionhandler )

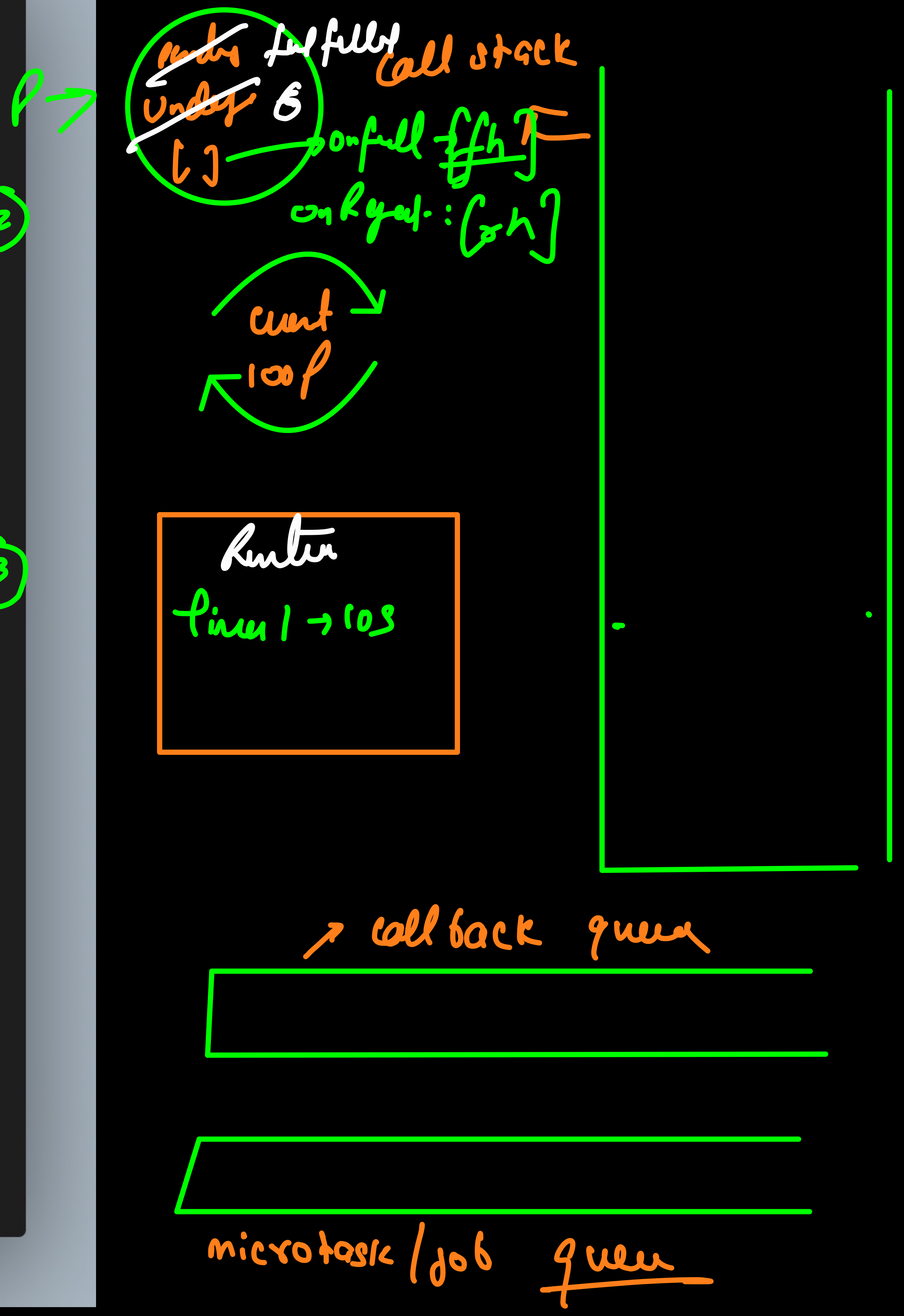
p. then ( ______ ___ )

these are handler function , that we have to implement ourselves.

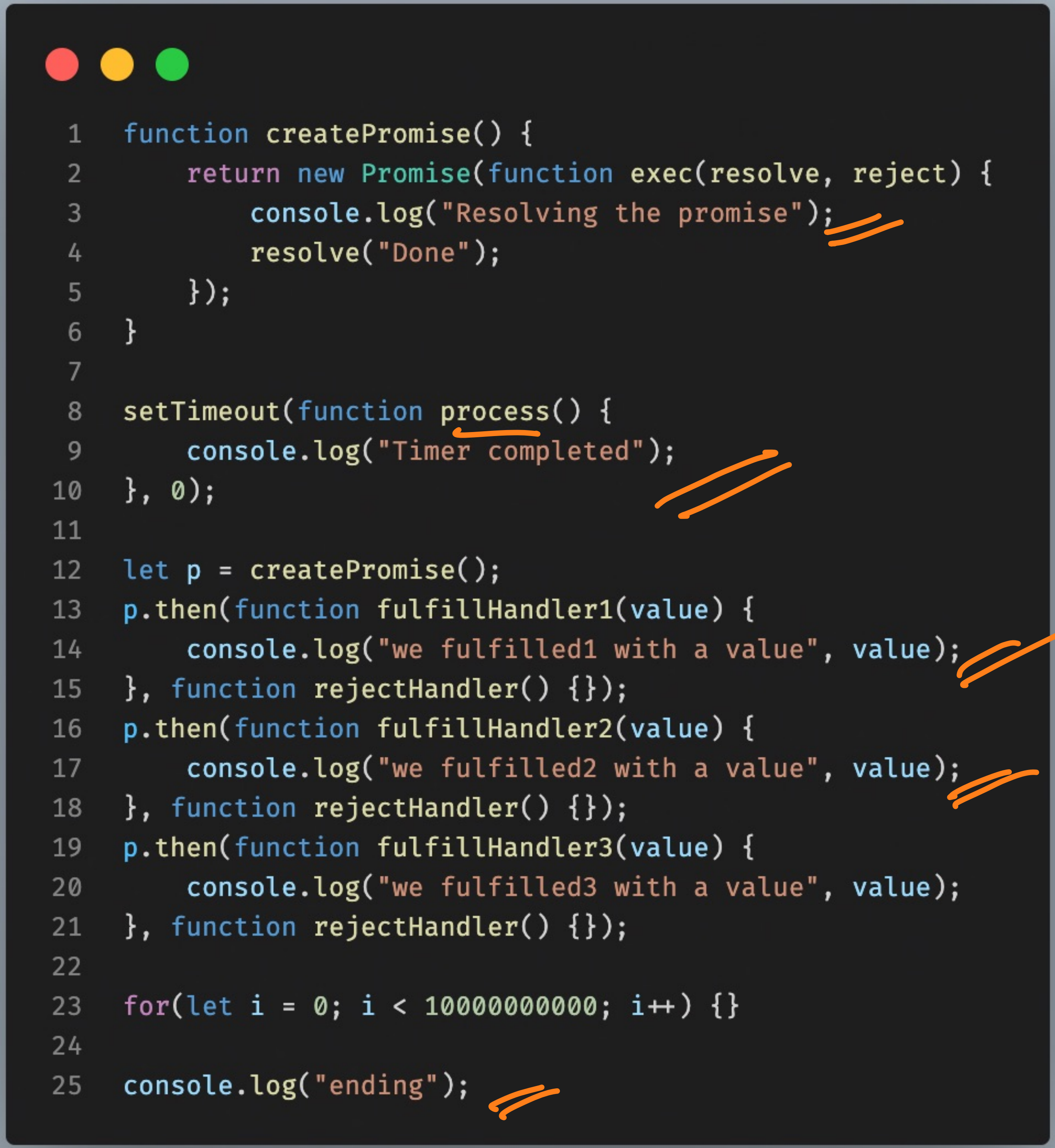
Value :
State :
On fulfillment : [f, g ]
on rejection : [h, k ]



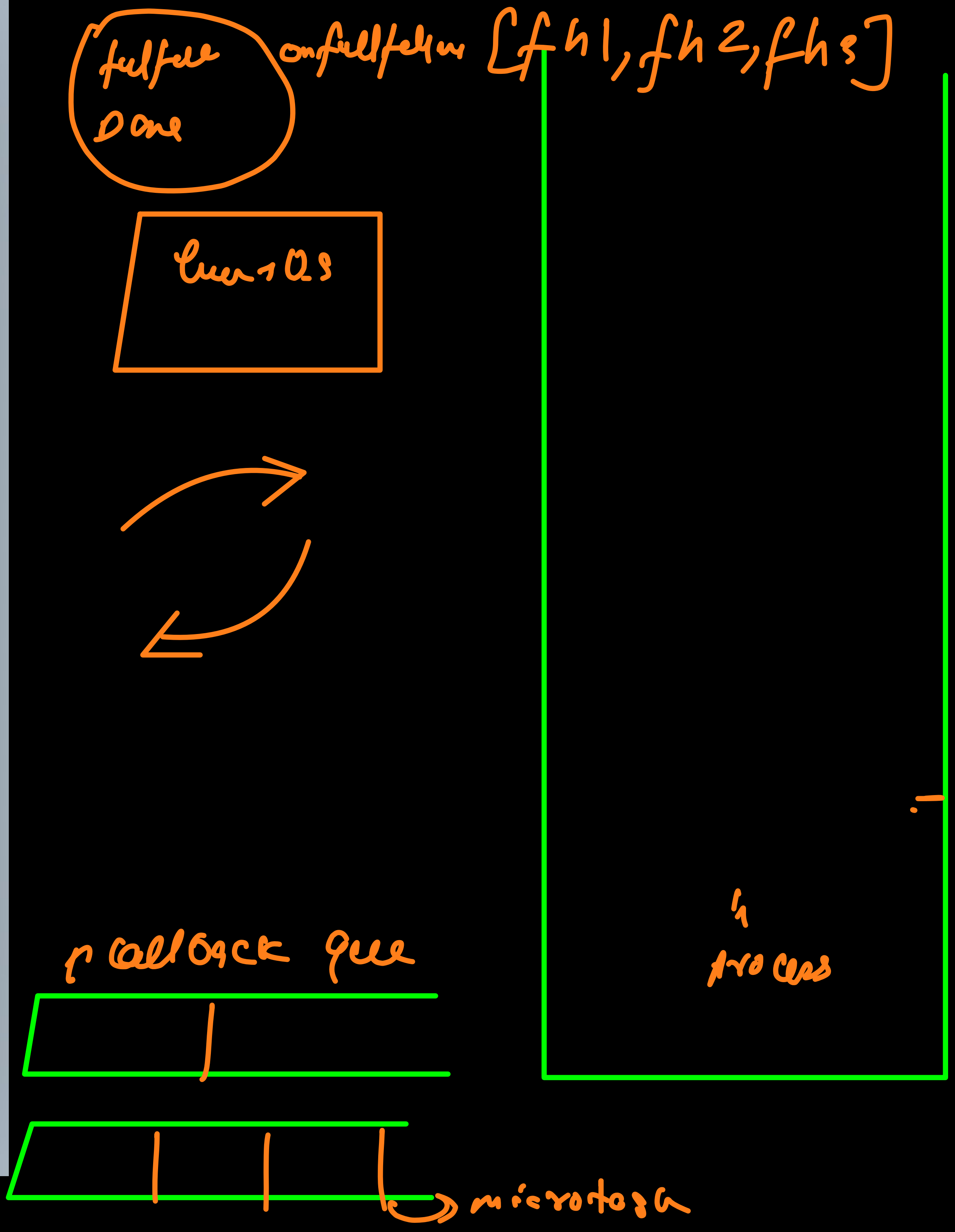
```javascript
function getRandomInt(max) {
    return Math.floor(Math.random() * max);
}
function createPromiseWithTimeout() {
    return new Promise(function executor(resolve, reject) {
        console.log("Entering the executor callback in the promise constructor");
        setTimeout(function () {
            let num = getRandomInt(10);
            if(num % 2 == 0) {
                // if the random number is even we fullfill
                resolve(num);
            } else {
                // if the random number is odd we reject
                reject(num);
            }
        }, 10000);
        console.log("Exitting the executor callback in the promise constructor");
    });
}
console.log("Starting....");
const p = createPromiseWithTimeout();
console.log("We are now waiting for the promise to complete");
console.log("Currently my promise object is like ... ", p);
p
.then(
    function fulfillHandler(value) {
        console.log("Inside fulfill handler with value", value);
        console.log("Promise after fullfillment is", p);
    },
    function rejectionHandler(value) {
        console.log("Inside rejection handler with value", value);
        console.log("Promise after rejection is", p);

    }
);
```

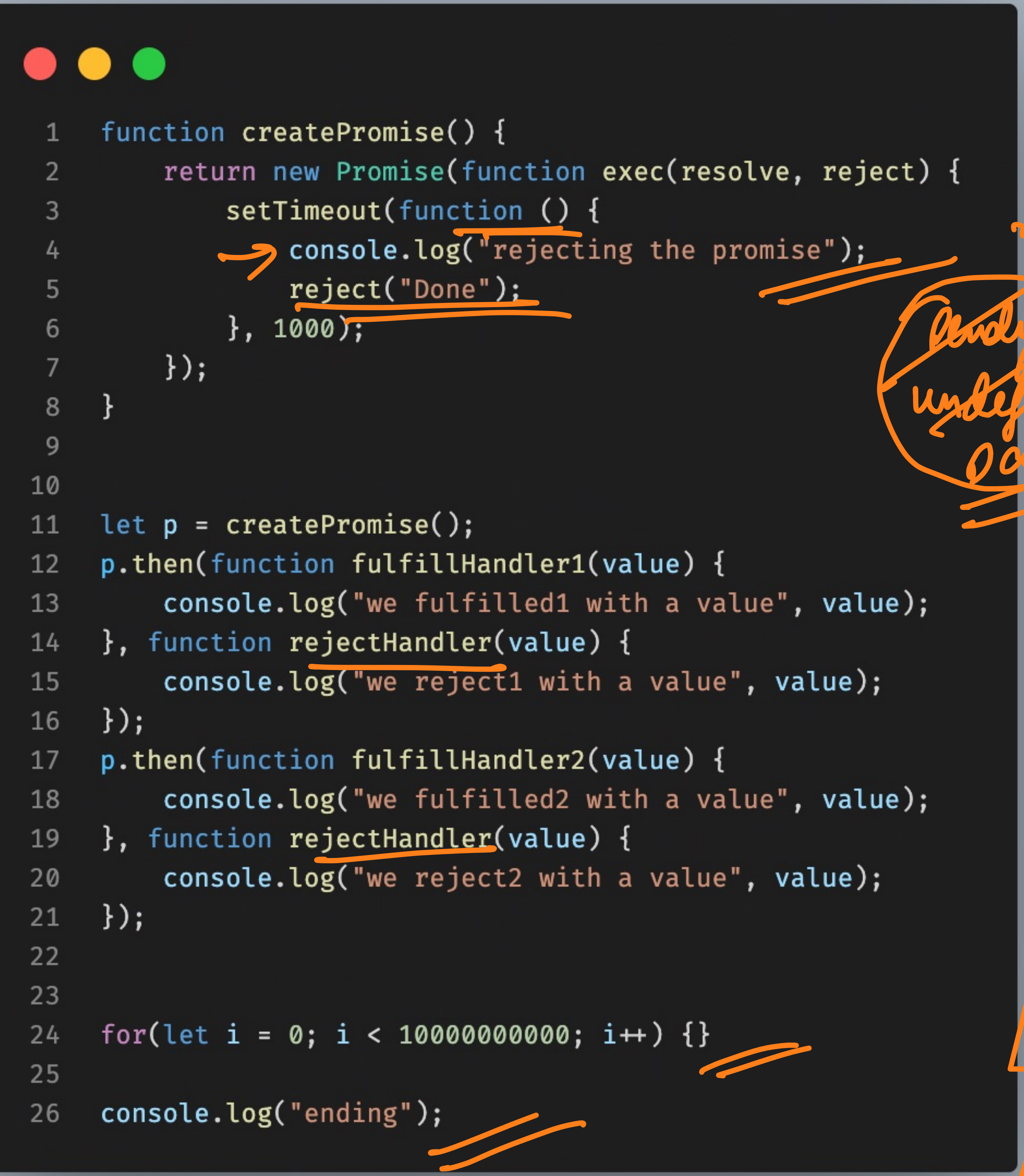


→ At any point of time, if event loop has a choice to pick from microtask queue or call back queue (macrotask queue) then it always gives preference to microtask **queue**.

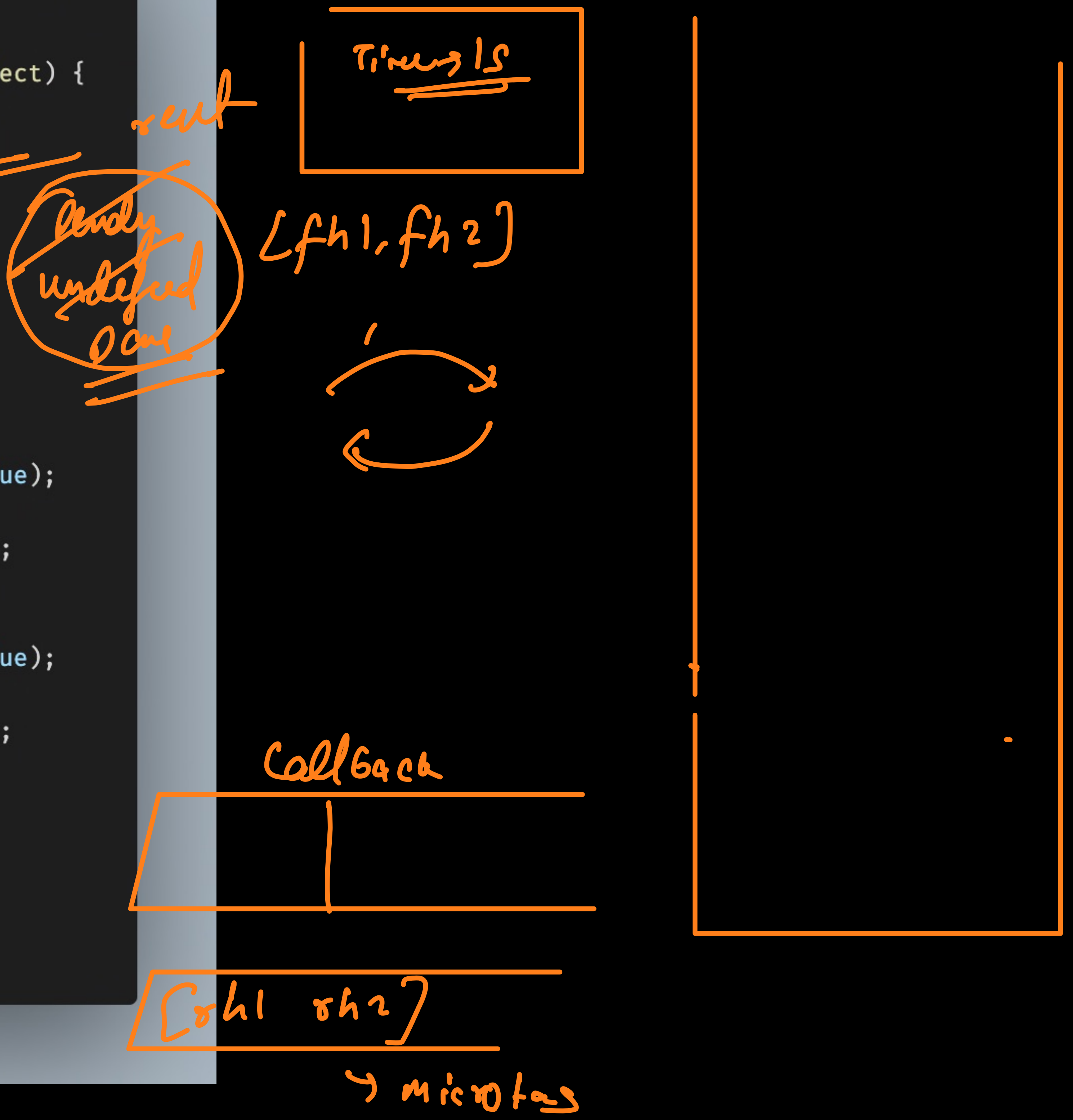


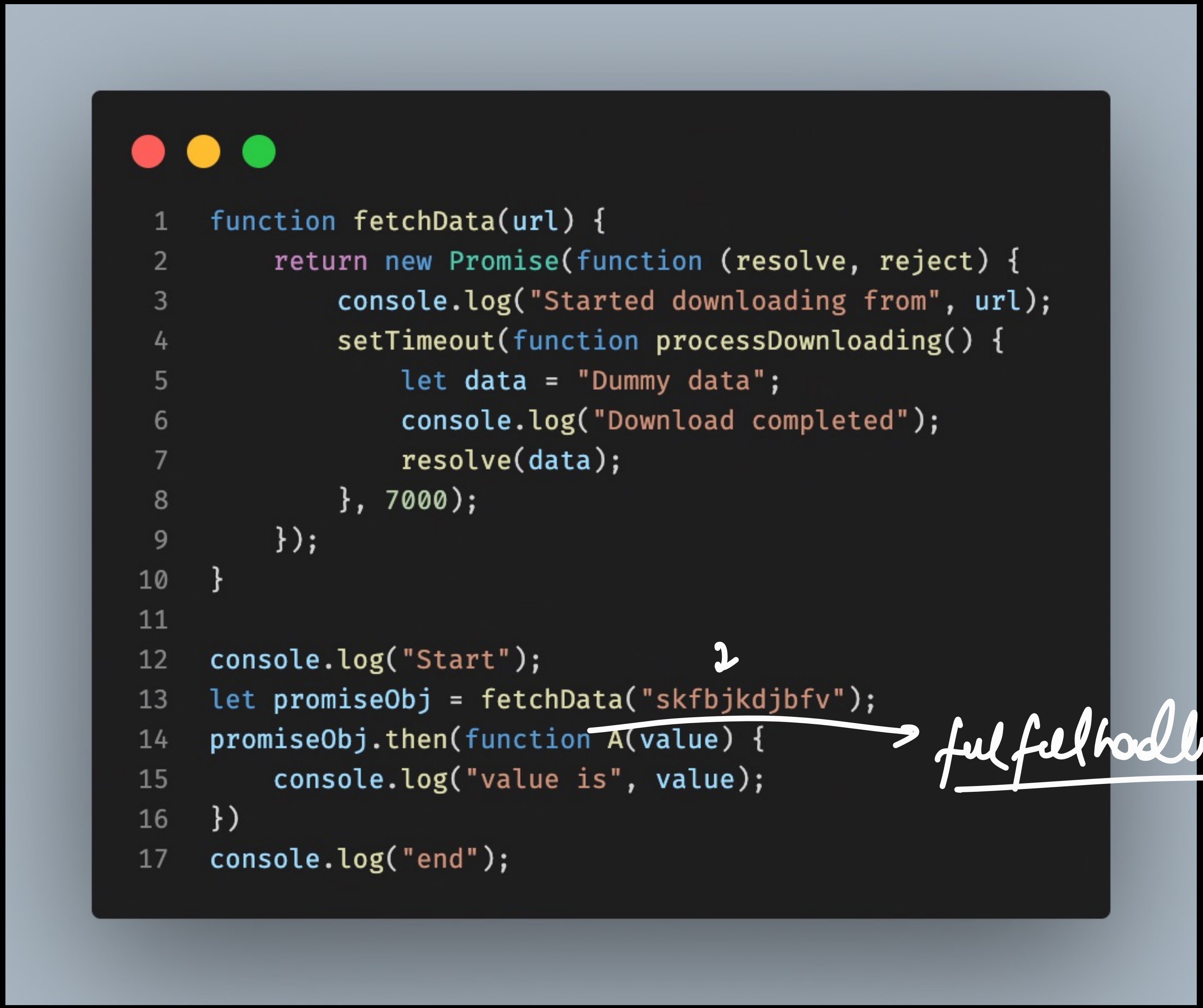
```javascript
function createPromise() {
    return new Promise(function exec(resolve, reject) {
        console.log("Resolving the promise");
        resolve("Done");
    });
}

setTimeout(function process() {
    console.log("Timer completed");
}, 0);

let p = createPromise();
p.then(function fulfillHandler1(value) {
    console.log("we fulfilled1 with a value", value);
}, function rejectHandler() {});
p.then(function fulfillHandler2(value) {
    console.log("we fulfilled2 with a value", value);
}, function rejectHandler() {});
p.then(function fulfillHandler3(value) {
    console.log("we fulfilled3 with a value", value);
}, function rejectHandler() {});

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

console.log("ending");
```





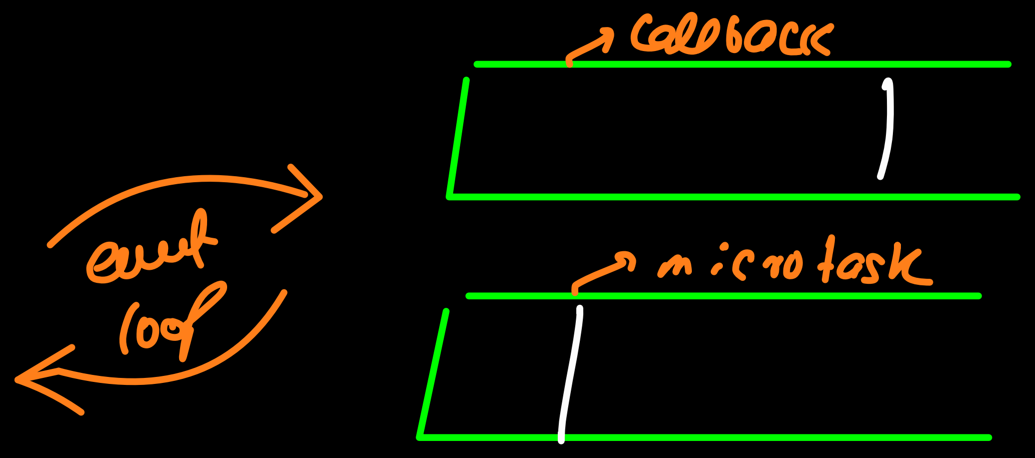
```javascript
function createPromise() {
    return new Promise(function exec(resolve, reject) {
        setTimeout(function () {
            console.log("rejecting the promise");
            reject("Done");
        }, 1000);
    });
}


let p = createPromise();
p.then(function fulfillHandler1(value) {
    console.log("we fulfilled1 with a value", value);
}, function rejectHandler(value) {
    console.log("we reject1 with a value", value);
});
p.then(function fulfillHandler2(value) {
    console.log("we fulfilled2 with a value", value);
}, function rejectHandler(value) {
    console.log("we reject2 with a value", value);
});


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

console.log("ending");
```

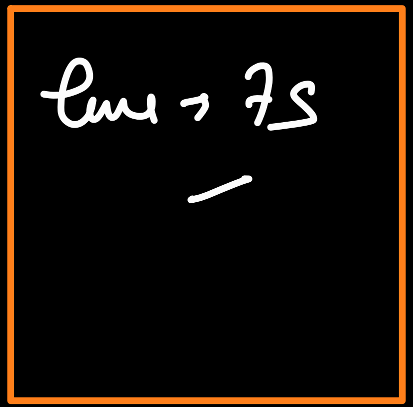


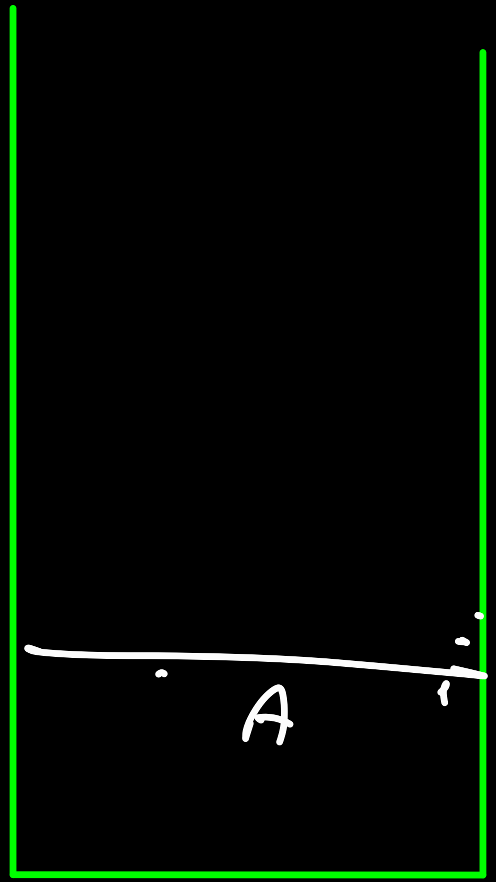


```javascript
function fetchData(url) {
    return new Promise(function (resolve, reject) {
        console.log("Started downloading from", url);
        setTimeout(function processDownloading() {
            let data = "Dummy data";
            console.log("Download completed");
            resolve(data);
        }, 7000);
    });
}

console.log("Start");
let promiseObj = fetchData("skfbjkdjbfv");
promiseObj.then(function A(value) {
    console.log("value is", value);
})
console.log("end");
```

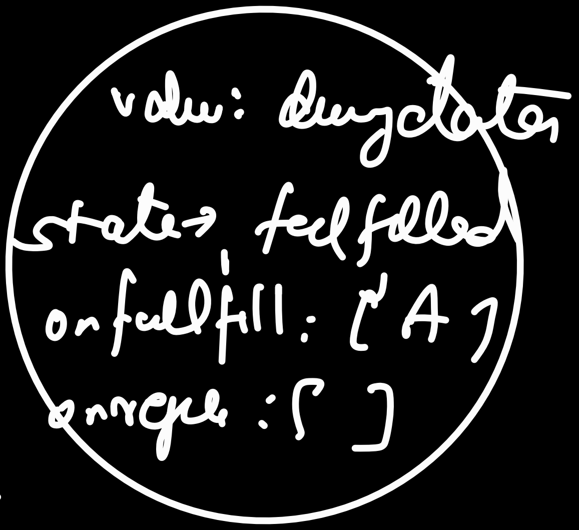
fulfilled

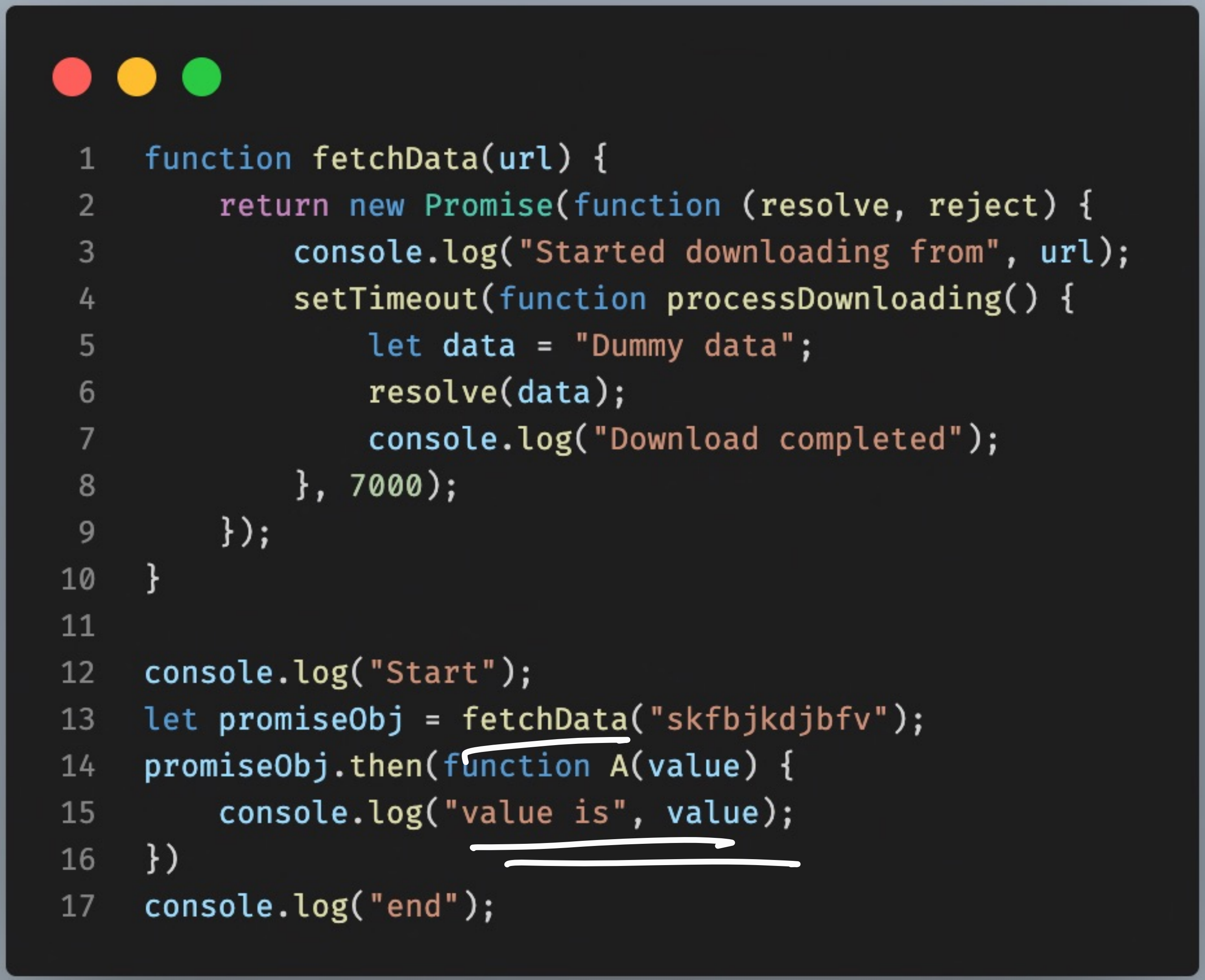




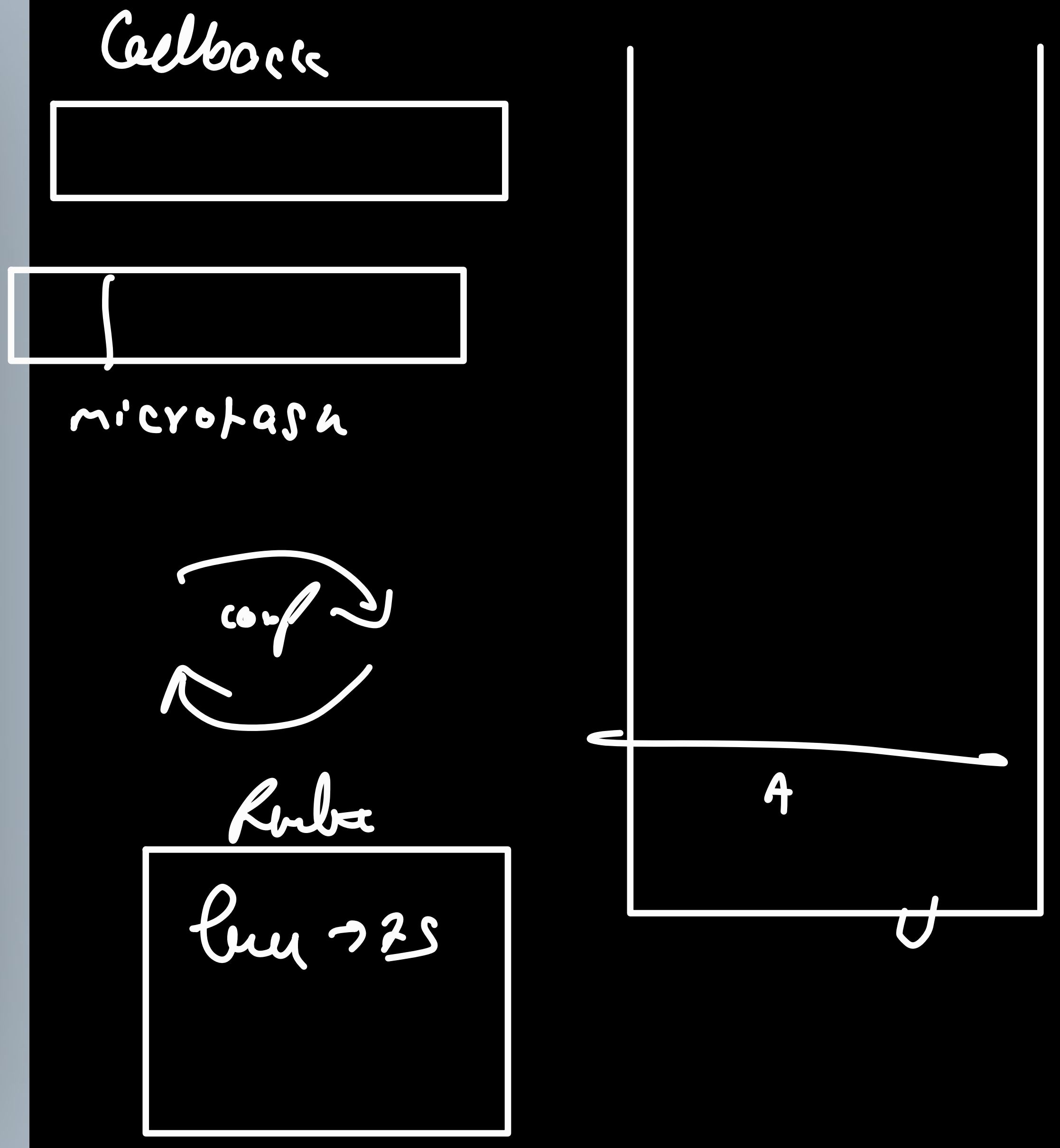


start
started downloading from —
end.
download complete
value is dummy data.





```javascript
function fetchData(url) {
    return new Promise(function (resolve, reject) {
        console.log("Started downloading from", url);
        setTimeout(function processDownloading() {
            let data = "Dummy data";
            resolve(data);
            console.log("Download completed");
        }, 7000);
    });
}

console.log("Start");
let promiseObj = fetchData("skfbjkdjbfv");
promiseObj.then(function A(value) {
    console.log("value is", value);
})
console.log("end");
```

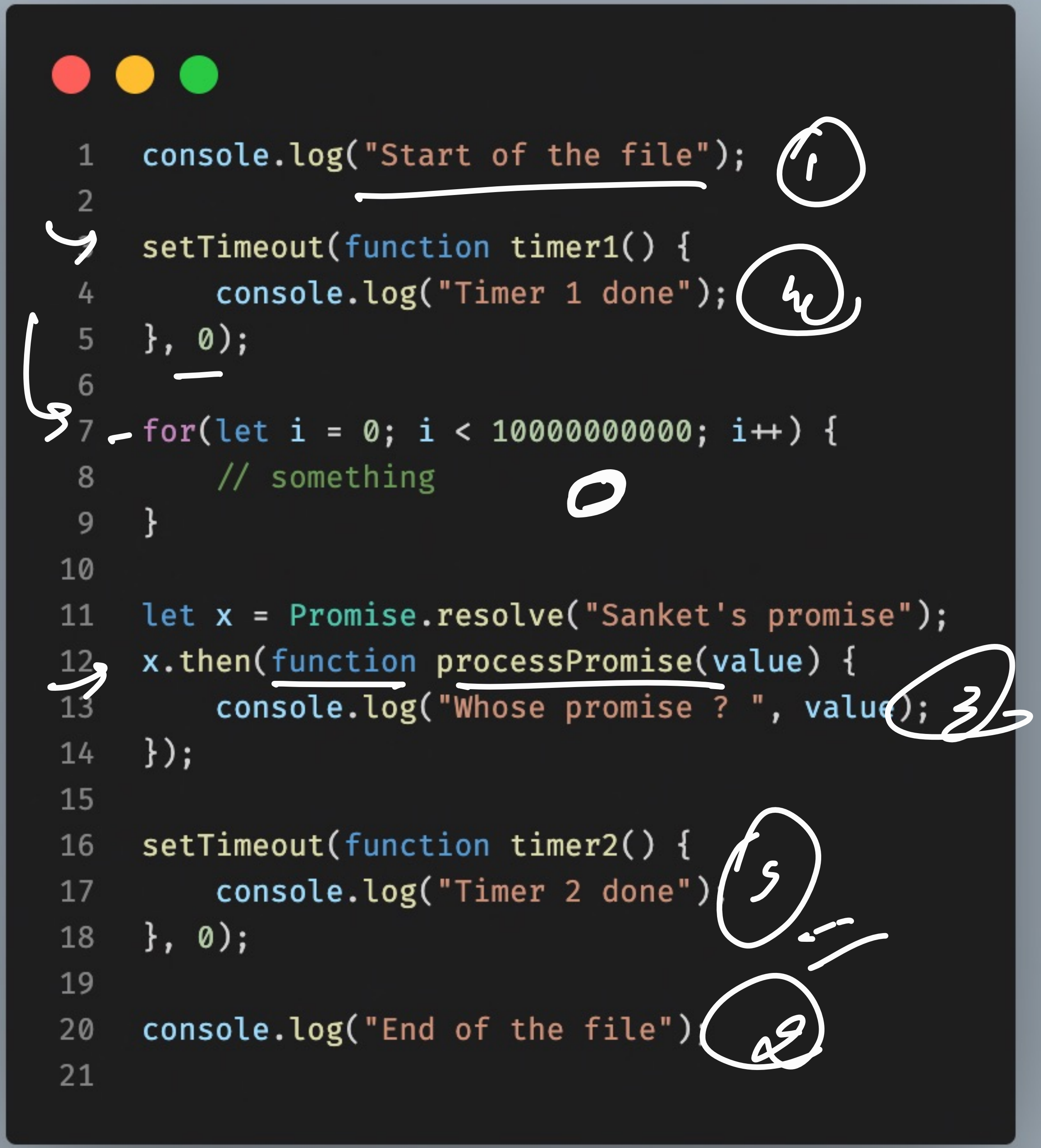


Start
Started download from
end
download completed
value is dummy data

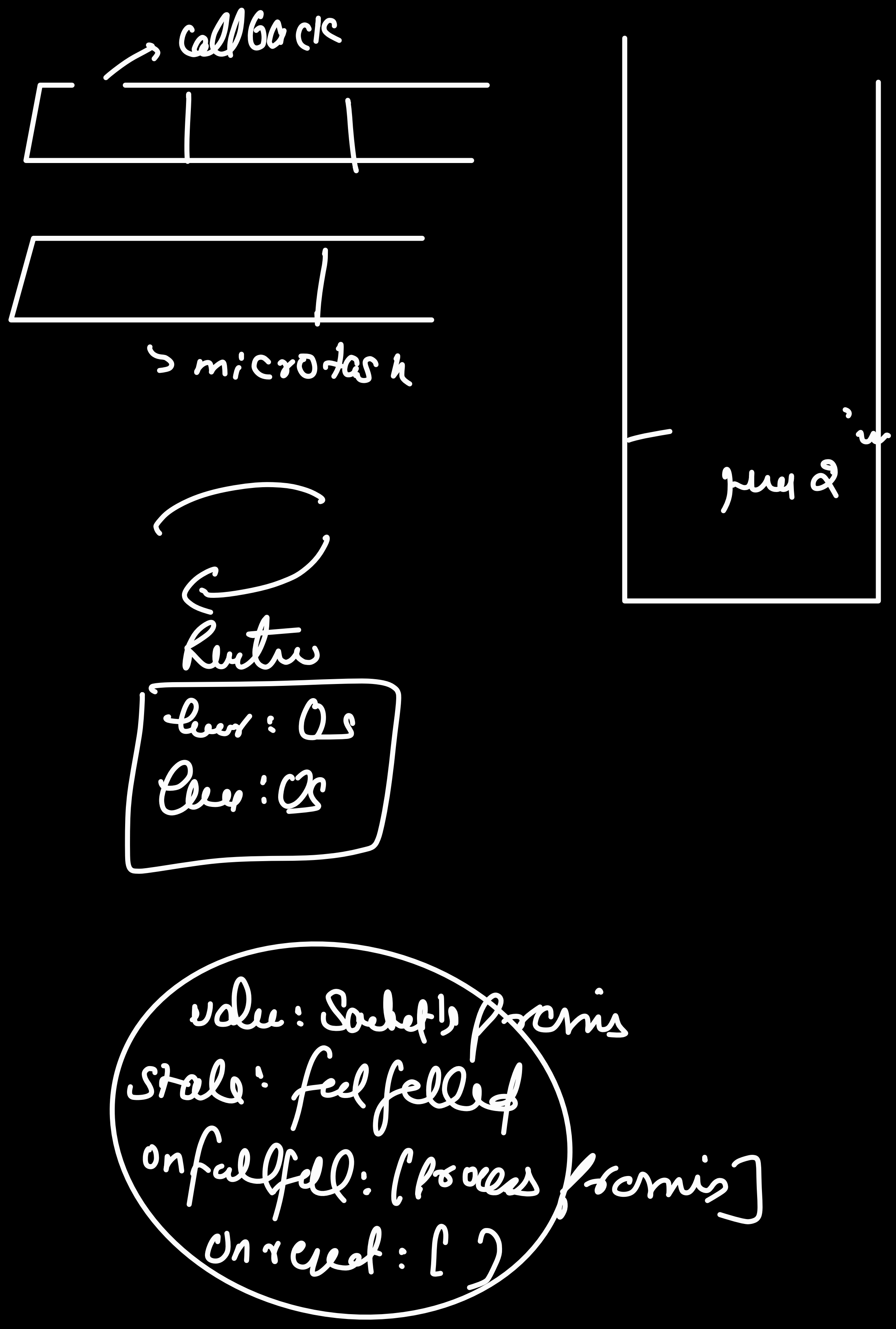
value: dummy data
State: fullfilled
onfulfill: []
onreject: []

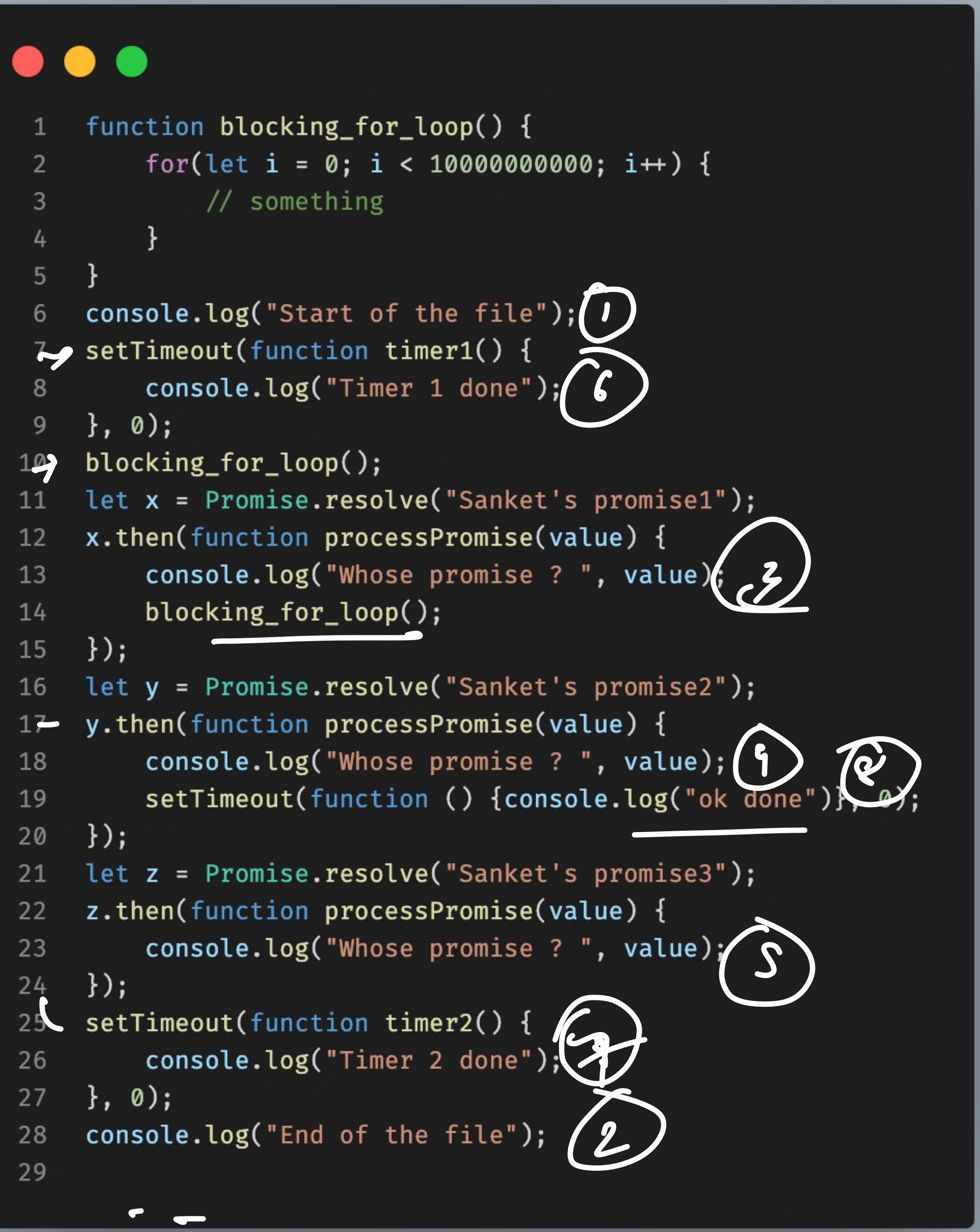
$$\begin{tikzcd}
\underline{\underline{\text{call stack}}} \arrow[d] & > & \text{micro task} & > & \text{Callback/} \\
\text{global code} & & & & \underline{\underline{\text{macrotask}}}
\end{tikzcd}$$

$$\longrightarrow$$

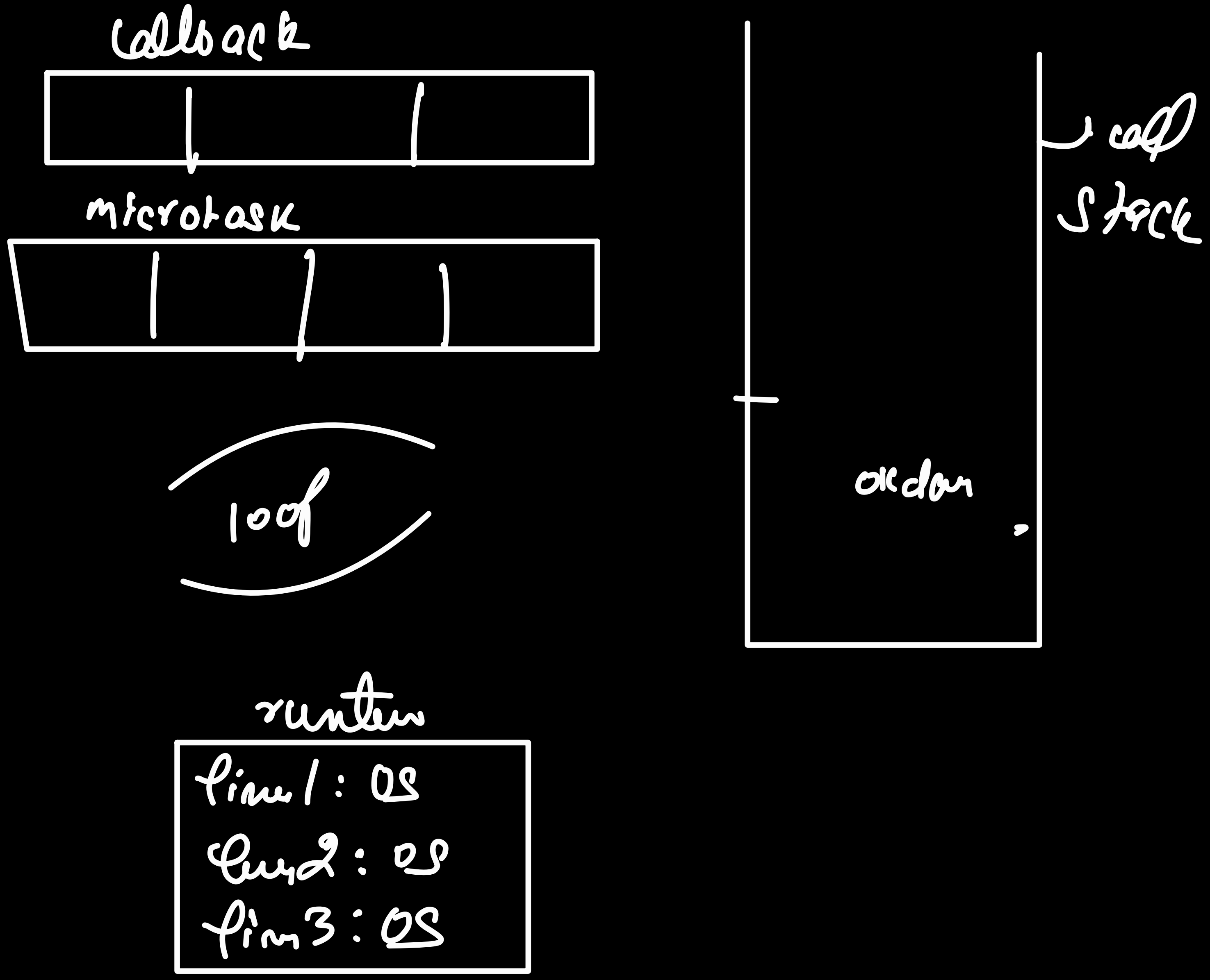


```javascript
console.log("Start of the file");

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");
```





```javascript
function blocking_for_loop() {
    for(let i = 0; i < 10000000000; i++) {
        // something
    }
}
console.log("Start of the file");
setTimeout(function timer1() {
    console.log("Timer 1 done");
}, 0);
blocking_for_loop();
let x = Promise.resolve("Sanket's promise1");
x.then(function processPromise(value) {
    console.log("Whose promise ? ", value);
    blocking_for_loop();
});
let y = Promise.resolve("Sanket's promise2");
y.then(function processPromise(value) {
    console.log("Whose promise ? ", value);
    setTimeout(function () {console.log("ok done")}, 0);
});
let z = Promise.resolve("Sanket's promise3");
z.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 → { value : Sanket's Promise1
state : fulfilled
onfulfill : [ processPromise ]
onreject : [ ] }

y → { value : Sanket's Promise2
state : fulfilled
onfulfill : [ processPromise ]
onreject : [ ] }

z → { value : Sanket's Promise1
state : fulfilled
onfulfill : [ processPromise ]
onreject : [ ] }

The .then function returns a new promise object. It immediately returns a new Promise object