Asynchronous Javascript

Javascript is a single threaded language, so it is understood that it can only execute one thing at a single point of time. Unlike languages like Java and Python, which are multi threaded, Javascript performs a sync task by the virtue of 'delegation'. This simply means that whenever the JS engine sees an asynchrony task, it delegates it to the browser, all while executing synchronous code.

The browser, written in C++, is able to achieve these asynchronous tasks (low level tasks) easily with its inbuilt methods and functionalities. There is always a communication between these two worlds of JS and the browser, via which each of them is aware of what is happening inside of the individual worlds.

And by this concept, JS is able to achieve a non blocking performance of asynchronous tasks with ease.

Synchronous means sequentially, while asynchronous means parallel. In the context of JS, sync code is that code which can be executed inside other world of JS, while async code is that code which can be executed inside the browser world.

List of synchronous code in JS

- variable declarations
- Function definitions and function expressions
- For loops, if-else, switch statements
- callback functions execution
- Sync file read (readFIle)

List of asynchronous code in JS

- setTimeout
- setInterval
- Promises / API calls made to the server
- Background tasks
- CRON jobs (server side)
- Asynchronous file read (readAsyncFile)

SetTImeout and setInterval are wrapper methods in JS, which refer to the C++ timer apis. The callbacks passed inside of these methods are themselves executed in the JS world

Another point to be noted here is that, the functions and callbacks which are delegated to the browser are not actually given to the browser world. Rather, a reference of these functions and methods are created in the browser world.

When the browser finishes executing the async task, it informs the JS world to execute the callback functions present in its memory.

Callback functions (revisit)

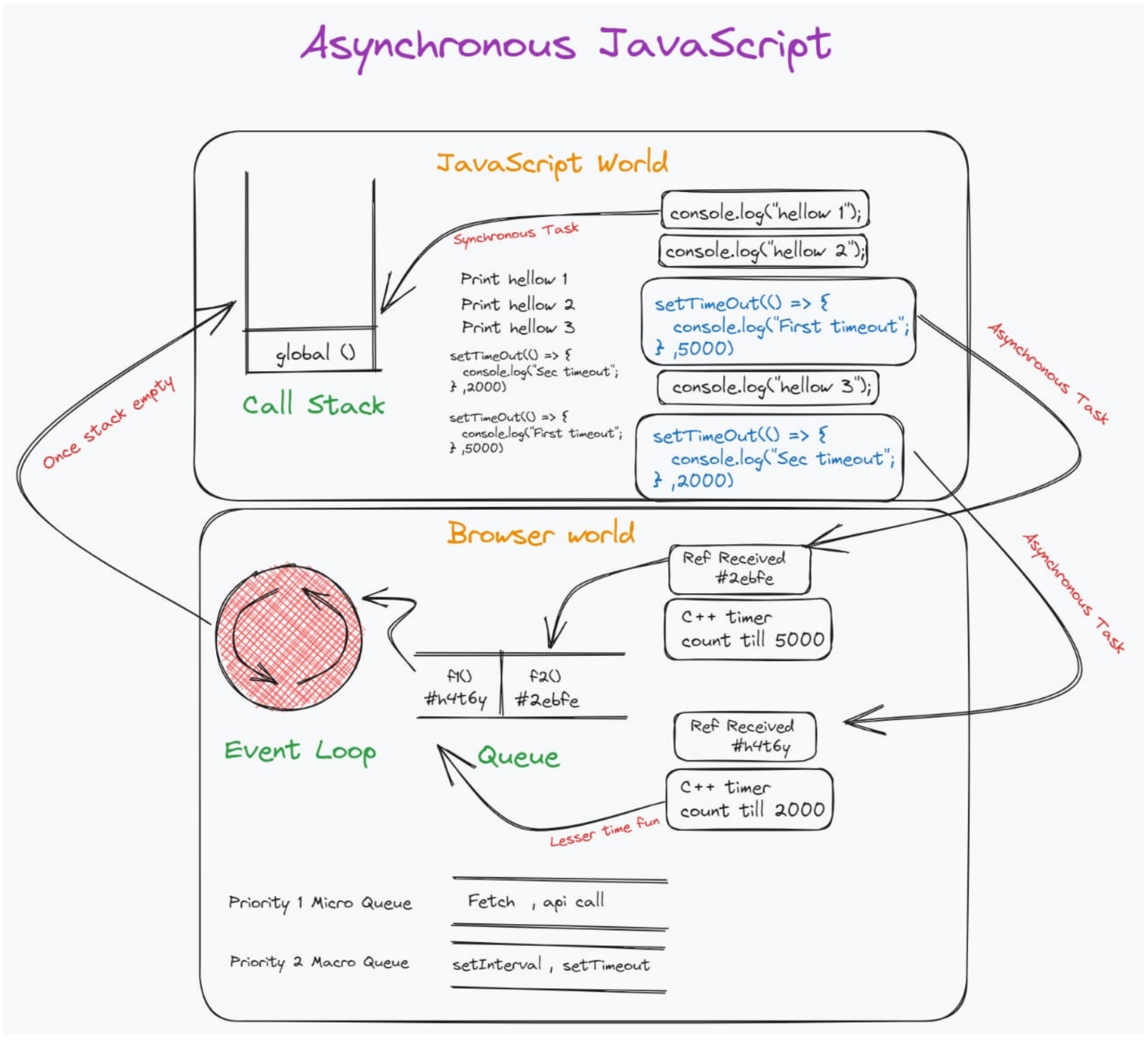
Callback functions are functions which are passed inside of another function/ method to be executed. Callback functions are an essential part of the async programming in JS, as these callback functions are executed inside the JS world with the data given by the browser world after a certain async task has been completed by it.

Callback functions are not just in the async code, but sync code as well. The array methods like .map(), .reduce(), .filter() also leverage the power of callback functions.

You can use both arrow functions as well as regular functions as callback functions.

How async code runs in Javascript / Working of an event loop

The following diagram depicts how the event loop works in JS.



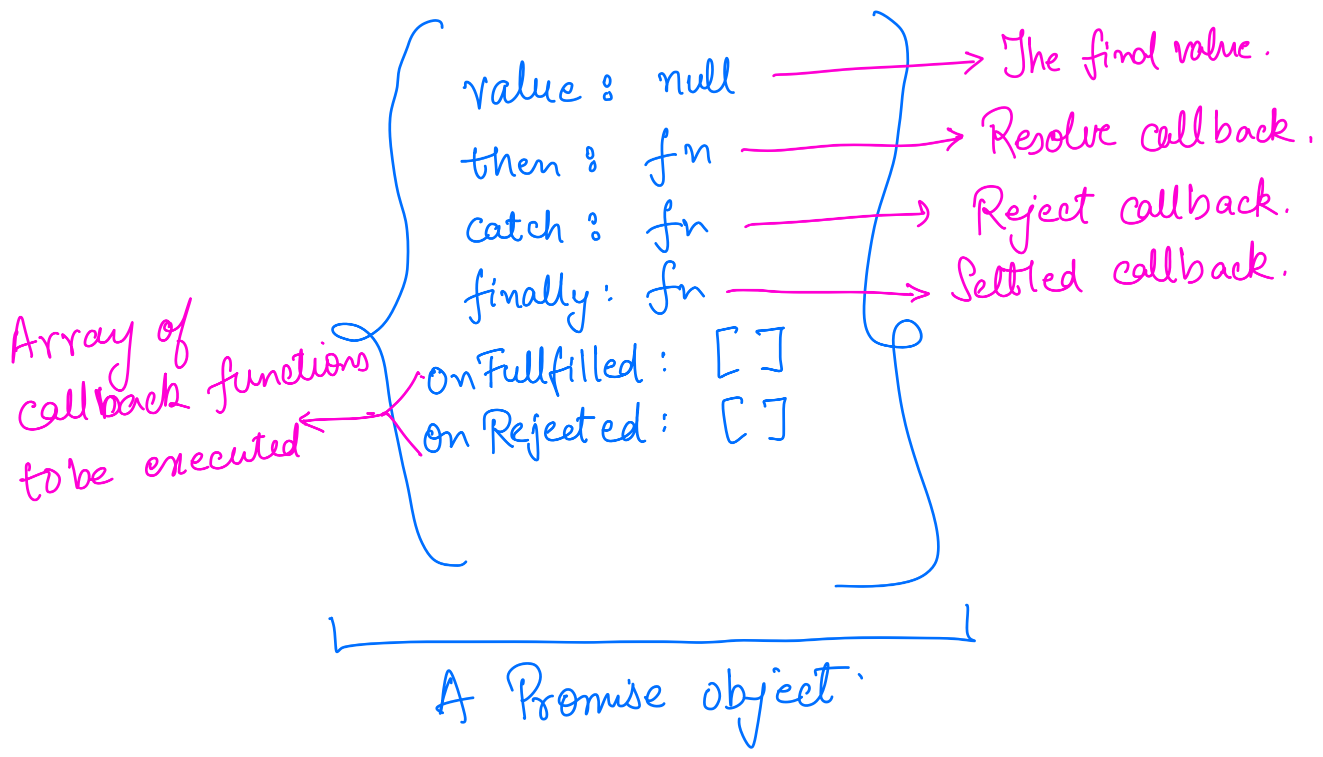
Picture Courtesy : Yash Jain

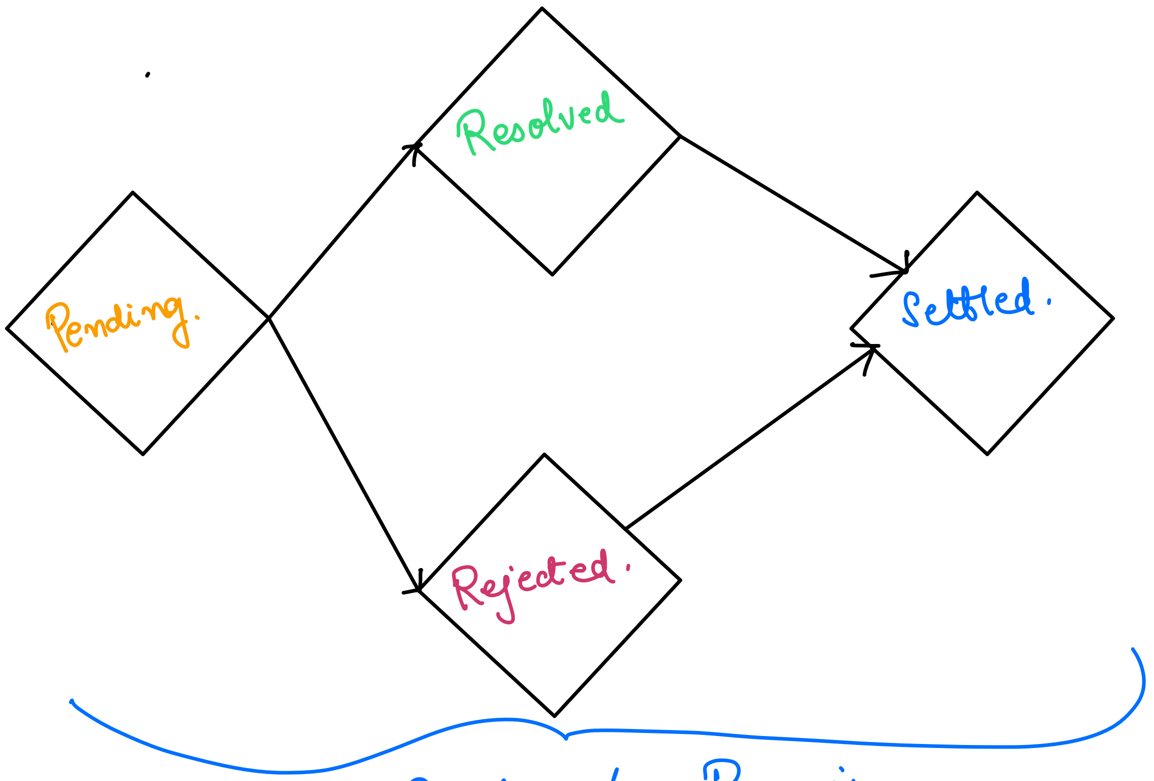
Important points to remember -:

- All the sync code takes utmost priority, no matter how many number of async code is present.
- Depending on the availability of the call stack, the event loop starts informing JS about executing the callback functions present in the memory.
- Micro tasks have a higher priority than macro tasks.

Promises

Promises are objects in Javascript which represent an eventual future value. Promises are guarantees that a browser provides JS that an async task will be completed and the resultant information be shared with Javascript itself.





States of a Promise

1. Pending $\longrightarrow$ Resolved $\longrightarrow$ Settled
2. Pending $\longrightarrow$ Rejected $\longrightarrow$ Settled.

Creation of a new promise in Javascript

We can create a new promise in Javascript by instantiating the Promise class present in Javascript. The Promise class takes a callback function with two arguments : resolve and reject, which are functions themselves.

```
let promise = new Promise((resolve, reject) => {
        let a = 2;
        let b = 3;
        if (a === b) {
            resolve('They are equal');
        } else {
            reject('They are not equal');
        }
})
```

The resolve and the reject function takes in an argument which is the data which is passed on to the callback functions when the promise either resolves or rejects.

To use this data in the callback, we use the .then() and .catch() methods respectively.

```
promise.then((data) => console.log(data)).catch(err => console.log(err));
```

If you don't resolve or reject the promise, a rejected promise throws a Javascript error.

We can also chain promises by returning another promise from inside of a callback of a promise, and use .then() method to resolve that returned promise. This enables us to create a chaining sequence of execution of promises. This is called Promise Chaining in Javascript.

```
Const p1 = new Promise((resolve,reject) => resolve('First promise'));
P1.then((data) => {
    const p2 = new Promise((resolve, reject) => resolve('second promise'));
    return p2;
}).then((data) => {
    const p3 = new Promise((resolve, reject) => resolve('third promise'));
    return p3;
})
```