What this chapter is about?

async await >> promise chains >> callback hell

Sync in JS

Synchronous

Synchronous means the code runs in a particular sequence of instructions given in the program. Each instruction waits for the previous instruction to complete its execution.

Asynchronous

Due to synchronous programming, sometimes imp instructions get blocked due to some previous instructions, which causes a delay in the UI. Asynchronous code execution allows to execute next instructions immediately and doesn't block the flow.

Callbacks

A callback is a function passed as an argument to another function.

Callback Hell

Callback Hell : Nested callbacks stacked below one another forming a pyramid structure.

(Pyramid of Doom)

This style of programming becomes difficult to understand & manage.

```
function hello() {
    console.log("Hello");
}

setTimeout(hello, 2000);
```

setTimeout(callback func(), timer in ms) : is an function that takes function as an argument(callback).

*Paste the code and understand the flow and control

```
function getData(dataId, getNextData) { //getNextData as callback
    console.log("Inside getData", dataId);
    setTimeout(() => {
        console.log("Inside setTimeOut");
        console.log("Data", dataId);
        console.log("value of getNextData:", getNextData);
        if (getNextData) {
            console.log("Inside if (getNextData)");
            getNextData();
        }
    }, 1000); //end of setTimeout
}

// getData(1, () => {
//     console.log("Preparing for next getData(2)");
//     getData(2);
// });

getData(1, () => {
    getData(2, () => {
        getData(3);
    });
});
```

=> Nested Callbacks
=> Becomes unable to understand and manage when
in a large scale.
=> Cretes structure like pyramid

Promises

Promise is for “eventual” completion of task. It is an object in JS.

It is a solution to callback hell.

$$\texttt{let promise = new Promise(}\underbrace{\texttt{(resolve, reject) => \{ .... \}}}_{\text{Function with 2 handlers}}\texttt{)}$$

```
let promise = new Promise((resolve, reject) => {
    console.log("Inside Promise");
    //Try one by one
    // resolve();
    // reject("your promise has some error");
});
```

*resolve & reject are callbacks provided by JS

Promises

for codes goto VS Code/Git Hub

```
=>execute data variable before and after:
=>Before: Pending and after: Fullfilled
function getData(dataId, nextData) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("Data", dataId);
            resolve("Data Generated Successfully");
            if (nextData) {
                nextData();
            }
        }, 5000);
    });
}
let data = getData(434);
```

A JavaScript Promise object can be:

- Pending : the result is undefined
- Resolved : the result is a value (fulfilled)
- Rejected : the result is an error object

resolve(result)
Resolve means successful

reject(error)

Reject means Not successful

*Promise has state (pending, fulfilled) & some result (result for resolve & error for reject).

Promises

.then( ) & .catch( )

promise.then((res) => { .... })

promise.catch((err)) => { .... })

```
function promise() {
    return new Promise((resolve, reject) => {
        console.log("Inside Promise");
        //Try one by one
        // resolve("Success");
        reject("error");
    });
}

let newPromise = promise();
newPromise.then(() => {
    console.log("Promise Fullfilled");
});
newPromise.catch(() => {
    console.log("Promise Rejected");
});
```

Async-Await

async function always returns a promise.

```
async function myFunc( ) { .... }
```

await pauses the execution of its surrounding async function until the promise is settled.

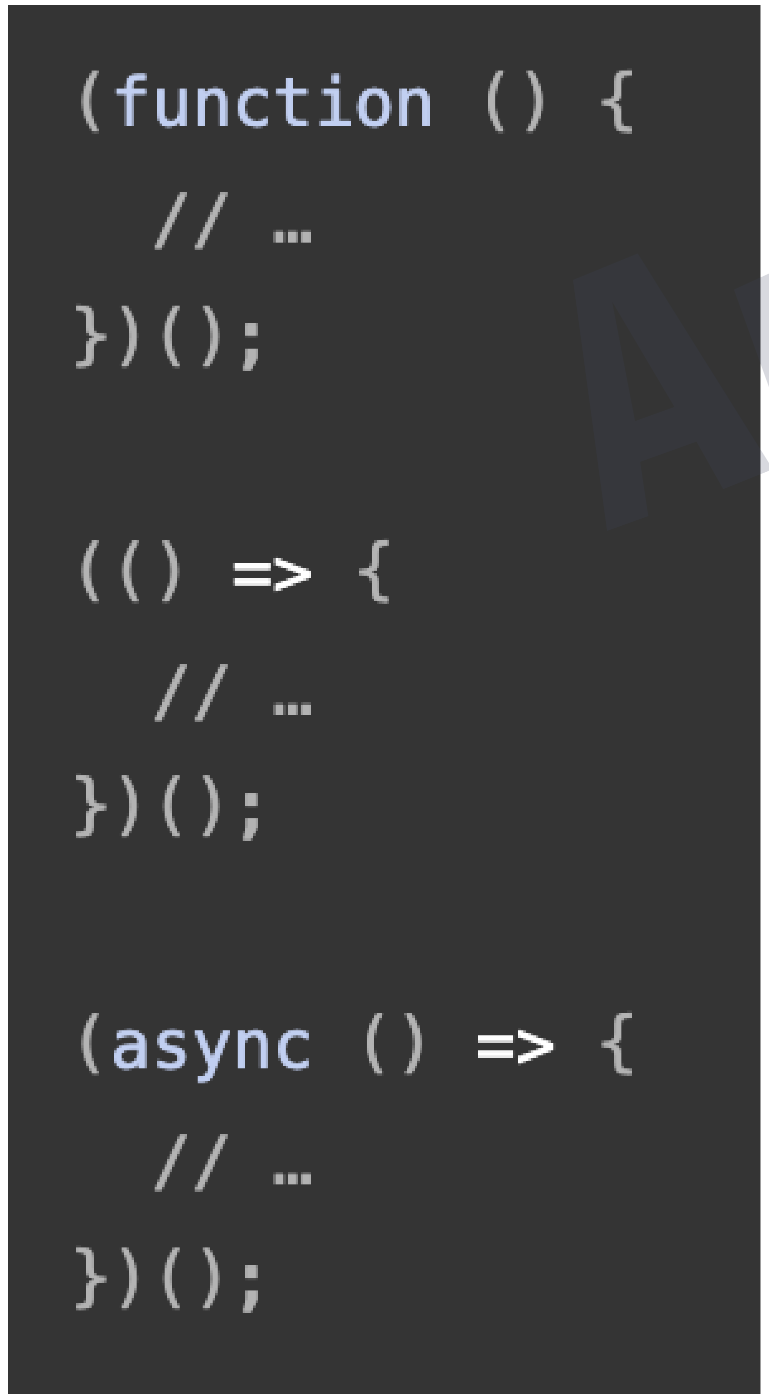
```
function Data(dataId) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("Data", dataId);
            resolve("Success");
        }, 2000);
    });
}

{//1st way with extra function name getData()
    async function getData() {
        //Each of the following waits for previous function.
        await Data(1);
        await Data(2); //Waits till Data(1) to execute and after it executes
        await Data(3); //Same for all the following
        await Data(4);
        await Data(5);
        await Data(6);
    }

    let gD = getData();
}
```

IIFE : Immediately Invoked Function Expression

IIFE is a function that is called immediately as soon as it is defined.



```
(function () {
  // …
})();

(() => {
  // …
})();

(async () => {
  // …
})();
```

```
function Data(dataId) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("Data", dataId);
            resolve("Success");
        }, 2000);
    });
}

{//2nd way without any function name using IIFE
    //Executes immediately (IIFE: Immediately Invoked Function Expression)

    (async function () {
        //Each of the following waits for previous function.
        await Data(1);
        await Data(2); //Waits till Data(1) to execute and after it executes
        await Data(3); //Same for all the following
        await Data(4);
        await Data(5);
        await Data(6);
    })();
}
```