**Asynchronous JS**

* **Callback hell**
  + This is a big issue caused by coding with complex nested callbacks. Here, each callback takes an argument that is a result of the previous callbacks. In this manner, the code structure looks like a pyramid, making it difficult to read and maintain. Also, if there is an error in one function, then all other functions get affected.
  + How to escape from a callback hell?
    - JavaScript provides an easy way of escaping from a callback hell. This is done by event queue and promises.
    - A promise is a returned object from any asynchronous function to which callback methods can be added based on the previous function’s result.
    - Promises use .then() method to call async callbacks. We can chain as many callbacks as we want, and the order is also strictly maintained.
    - Promises like .fetch() method fetches an object from the network. It also uses .catch() method to catch any exception when any block fails.
    - So, these promises are put in event queue so that they don’t block subsequent JS code. Also, once the results are returned, the event queue finishes its operations.
    - There are also other helpful keywords and methods like async, wait, setTimeout() to simplify and make better use of callbacks.
* **Asynchronous JavaScript**
  + It allows asynchronous functions to run in background.
  + Once the job is finished, we pass a callback function
  + **setTimeout ()**
    - It receives two parameters one is the function and second is the time in millisecond.
    - Sometimes it receives a third parameter which can become the input for the function we previously passed.
  + **Promises**
    - It the Object that keeps track about whether a certain event has happened already or not.
    - Determines what happens after the event has happened.
    - Implements the concept of a future value that we’re expecting.
    - When a new promise is made it is called PENDING, and when an event takes place, it is called SETTLED/RESOLVED which returns one of the two value which is either FULFILLED or REJECTED.
    - To create a promise, we use

const a = new Promise ((resolve, reject) =>{})

* Then to use the promise we use the then and catch

a

. then (//Function which receives data from the resolve promise //)

. catch (//Function which receives data from reject promise and on error//)

* + **Async/Await**
    - The async attribute is a boolean attribute.
    - If we define a function async it will always **return a Promise**.

const a = async () =>'hello'

a () // Promise {<fulfilled>: 'hello'}

* + - We can declare a function as async without using any await. In this case, the execution is not paused, and your code will be executed in a non-blocking manner.
    - When present, it specifies that the script will be executed asynchronously as soon as await is available.
    - The await expression causes async function execution to pause until a Promise is settled (that is, fulfilled or rejected), and to resume execution of the async function after fulfilment. When resumed, the value of the await expression is that of the fulfilled Promise
    - To handle errors here we use try/catch block
    - We can use await with any object that implements **a then function**.
      * Ex ->

Graphical user interface, text, application

Description automatically generated

* + **AJAX**
    - Diagram

      Description automatically generated
    - Here we will use different APIs to take data from the server and to the server
    - fetch() is used an it receives a URL and returns a promise
    - If no access is showing use website use crossorigin.me
    - It will return a promise which can be worked on with .then and .catch