**Topic: callback hell and promises**

**Callback hell**

Callback hell, also known as "**Pyramid of Doom,**" is a term used in JavaScript programming to describe a situation where multiple nested callbacks make the code difficult to read, understand, and maintain. This usually happens when dealing with asynchronous operations, such as making API requests or reading files.

**Syntax:**

step1(function() {

    step2(function() {

        step3(function() {

            console.log("All steps completed");

        });

    });

});

**Example:1**

      function first(callback) {

        console.log("first");

        callback();

      }

      function second(callback) {

        console.log("second");

        callback();

      }

      function third(callback) {

        console.log("third");

        callback();

      }

      function fourth(callback) {

        console.log("fourth");

      }

      first(() => {

        second(() => {

          third(() => {

            fourth();

          });

        });

      });

**Example:2**

      function add(val, callback) {

        callback(val + 10);

      }

      function sub(val, callback) {

        callback(val - 5);

      }

      function mul(val, callback) {

        callback(val \* 2);

      }

      function div(val, callback) {

        callback(val / 5);

      }

      add(10, (address) => {

        sub(addres, (subres) => {

          mul(subres, (mulres) => {

            div(mulres, (finalres) => {

               console.log(finalres);

            });

          });

        });

      });

**Example:3**

function wakeUp(callback) {

    setTimeout(() => {

        console.log("1. Woke up");

        callback();

    }, 1000);

}

function eatBreakfast(callback) {

    setTimeout(() => {

        console.log("2. Ate breakfast");

        callback();

    }, 1000);

}

function study(callback) {

    setTimeout(() => {

        console.log("3. Studied");

        callback();

    }, 1000);

}

function goToSleep(callback) {

    setTimeout(() => {

        console.log("4. Went to sleep");

        callback();

    }, 1000);

}

// The callback hell part starts here

wakeUp(() => {

    eatBreakfast(() => {

        study(() => {

            goToSleep(() => {

                console.log("Finished all tasks!");

            });

        });

    });

});

To mitigate callback hell, several approaches have been developed:

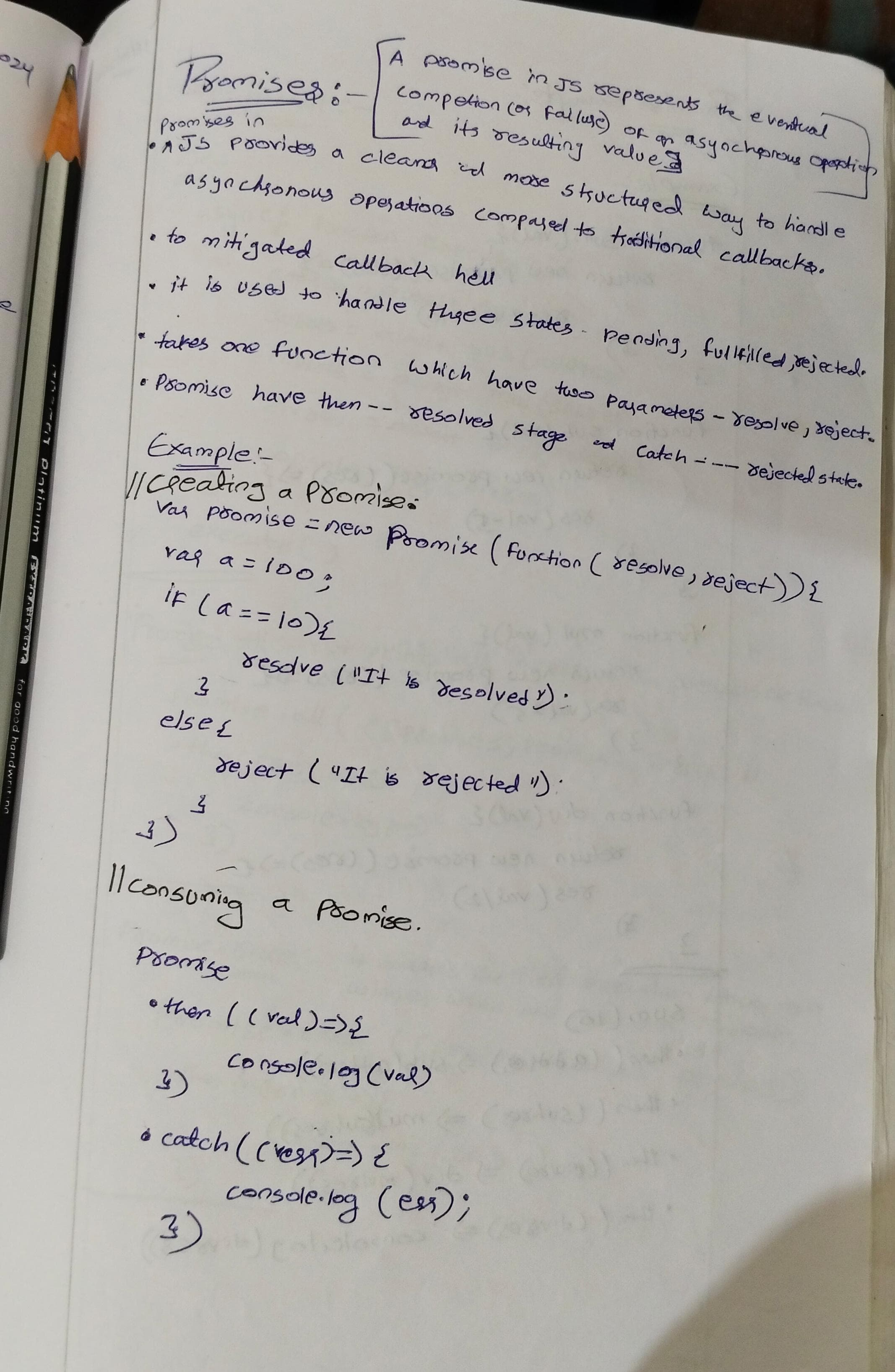
Named functions

Promises

Async/await

**Promises**

Promises in JavaScript provide a cleaner and more structured way to handle asynchronous operations compared to traditional callbacks.It has three states: pending, fulfilled, or rejected.



**Creating a Promise(producing)** : You create a new Promise object using the Promise constructor. This constructor takes a function as an argument, which in turn takes two parameters: resolve and reject. Inside this function, you perform your asynchronous operation, and when it's completed, you call resolve with the result or reject with an error if it fails.

      //promises creation

      var promises=new Promise(function (resolve,reject){

      var a=100;

      if(a==10){

      resolve("a is 10")

      }

      else{

      reject("a is not 10")

      }

      });

**Consuming a Promise:** You consume a promise using the then method, which takes two optional parameters: a callback function to handle the resolved value, and a callback function to handle any errors.

      //print the response

      promises.then((val)=>{

      console.log(val)

      }).catch((err)=>{

      console.log(err)

      })

Promises also forming chain method which inturns make code readability  difficult in order to avoid this

      let add = (val) =>

        new Promise((resolve, reject) => {

          resolve(val + 10);

        });

      let sub = (val) =>

        new Promise((resolve, reject) => {

          resolve(val - 10);

        });

      let mul = (val) =>

        new Promise((resolve, reject) => {

          resolve(val \* 5);

        });

      let div = (val) =>

        new Promise((resolve, reject) => {

          resolve(val / 2);

        });

      add(10)

        .then((addres) => sub(addres))

        .then((subres) => mul(subres))

        .then((mulres) => div(mulres))

        .then((divres) => console.log(divres))

        .catch((error) => console.error(error));

**Promises aysnchronous**

        let promise1= new Promise((resolve,reject)=>{

          console.log("promise 1");

          setTimeout(resolve, 2000, "promise 1 success")

        })

        let promise2= new Promise((resolve,reject)=>{

          console.log("promise 2");

          setTimeout(resolve, 1500, "promise 2 success")

        })

        let promise3= new Promise((resolve,reject)=>{

          console.log("promise 3");

          setTimeout(resolve, 1800, "promise 3 success")

        })

        let promise4= new Promise((resolve,reject)=>{

          console.log("promise 4");

          setTimeout(resolve, 500, "promise 4 success")

        })

        promise1.then((resolve)=>{console.log(resolve)})

        promise2.then((resolve)=>{console.log(resolve)})

        promise3.then((resolve)=>{console.log(resolve)})

        promise4.then((resolve)=>{console.log(resolve)})

//convert synchronous to aynchronous

      promise1

        .then((result) => {

          console.log(result);

          return promise2;

        })

        .then((result) => {

          console.log(result);

          return promise3;

        })

        .then((result) => {

          console.log(result);

          return promise4;

        })

        .then((result) => {

          console.log(result);

        });

**Async/Await**  
Async/await is a modern feature in JavaScript that simplifies working with asynchronous code, especially when dealing with Promises. It allows you to write asynchronous code in a synchronous-like manner, making it easier to read, write, and maintain.

1. **Async Functions**: An async function is a function that operates asynchronously via the event loop. You declare an async function by prefixing the function declaration with the **async** keyword.

      async function myAsyncFunction() {

        // Asynchronous code here

      }

1. **Await Keyword:** The await keyword is used inside an async function to pause the execution of the function until a Promise is settled (resolved or rejected). It allows you to write code that looks synchronous but behaves asynchronously.

      async function myAsyncFunction() {

        const result = await somePromise;

        // Code here executes after somePromise is resolved

      }

Example

//promise is created

      function apromise() {

        return new Promise(function (res, rej) {

          var a = 20;

          if (a % 2 == 0) {

            res("num is even");

          } else {

            rej("num is odd");

          }

        });

      }

//resolving the promise value using async/await

      async function asyncfun() {

        var v = await apromise();

        console.log(v);

      }

      asyncfun();

//callback hell

        async function executor(){

          var addres=await add(10);

          var subres=await sub(addres);

          var mulres=await mul(subres);

          var divres=await div(mulres);

          console.log(divres);

        }

        executor()

//asynchronous

      async function executor() {

        let result1 = await promise1;

        console.log(result1);

        let result2 = await promise2;

        console.log(result2);

        let result3 = await promise3;

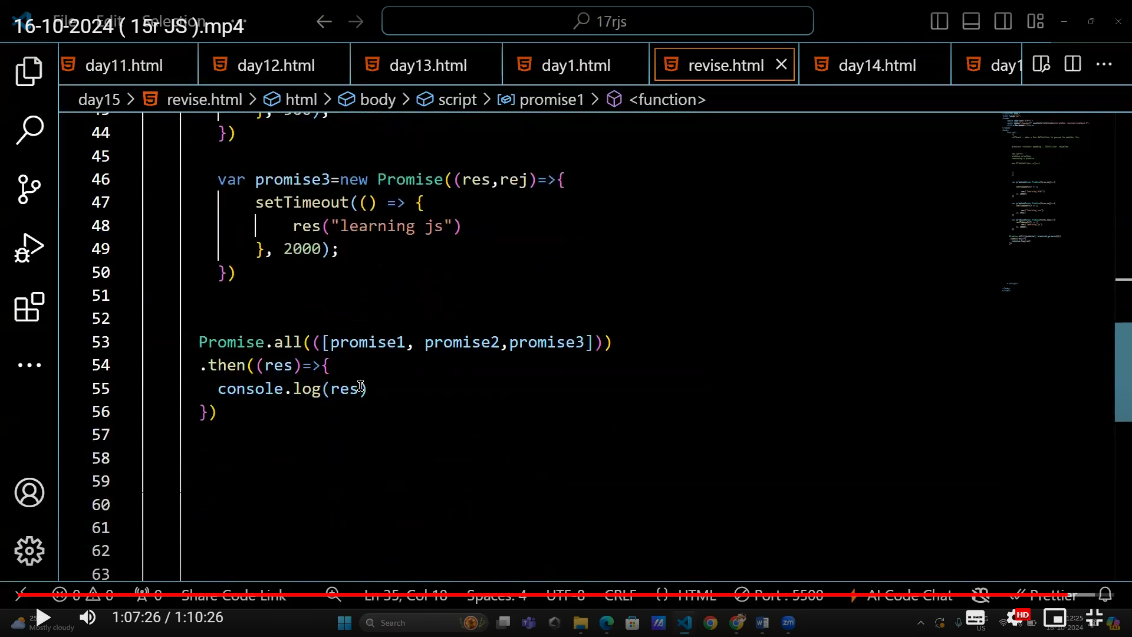
        console.log(result3);

        let result4 = await promise4;

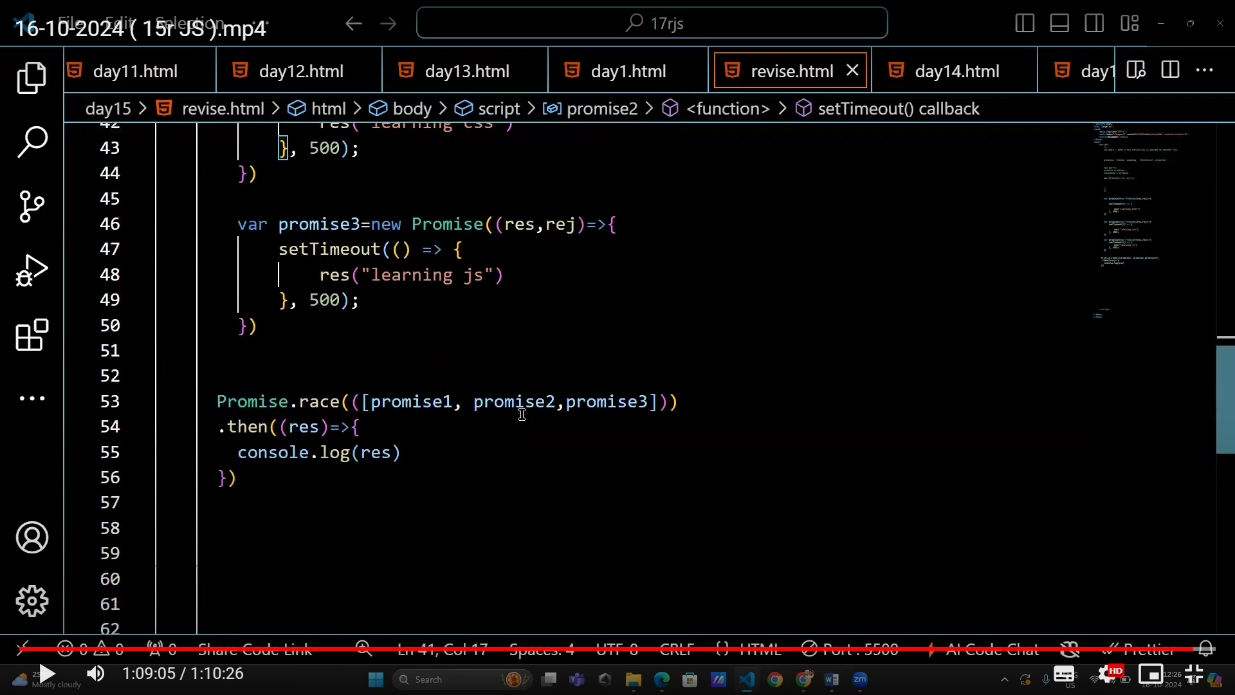
        console.log(result4);

      }

      executor();



// promise.all --> print all



//race – it declares the winner