JS

**Variables and Data Types**:

* JavaScript variables are declared using **var**, **let**, or **const**.
* Common data types include numbers, strings, booleans, arrays, and objects.

// Variable declaration

var x = 5;

let y = "Hello";

const PI = 3.14;

// Data types

let num = 10;

let str = "JavaScript";

let bool = true;

let arr = [1, 2, 3];

let obj = { name: "John", age: 30 };

**Functions**:

* Functions in JavaScript can be declared using the **function** keyword or as arrow functions **() => {}**.
* Functions can accept parameters and return values.

// Function declaration

function greet(name) {

return "Hello, " + name + "!";

}

// Arrow function

const add = (a, b) => {

return a + b;

};

// Traditional function expression

const double = function(x) {

return x \* 2;

};

// Arrow function equivalent

const double = x => x \* 2;

console.log(greet("World")); // Output: Hello, World!

console.log(add(3, 5)); // Output: 8

**Conditional Statements**:

* JavaScript supports **if**, **else if**, and **else** for conditional execution.
* Ternary operator (**condition ? expr1 : expr2**) can be used for concise conditional expressions.

let age = 18;

if (age >= 18) {

console.log("You are an adult.");

} else {

console.log("You are a minor.");

}

// Ternary operator

let result = (age >= 18) ? "Adult" : "Minor";

console.log(result); // Output: Adult

LOOPS:  
// For loop

for (let i = 0; i < 5; i++) {

console.log(i);

}

// While loop

let j = 0;

while (j < 5) {

console.log(j);

j++;

}

// Do-while loop

let k = 0;

do {

console.log(k);

k++;

} while (k < 5);

ARRAYS AND OBJECTS:  
// Array

let numbers = [1, 2, 3, 4, 5];

console.log(numbers[0]); // Output: 1

// Object

let person = { name: "John", age: 30 };

console.log(person.name); // Output: John

**DOM Manipulation**:

* JavaScript is often used to manipulate the Document Object Model (DOM) to interact with HTML elements on web pages.
* Common DOM methods include **getElementById()**, **querySelector()**, **addEventListener()**, etc.

<!-- HTML -->

<button id="myButton">Click me</button>

// JavaScript

document.getElementById("myButton").addEventListener("click", function() {

alert("Button clicked!");

});

PROMISE CHAINS

A promise chain in JavaScript is a sequence of asynchronous operations that are executed one after the other, where each operation depends on the result of the previous one. Promises are objects representing the eventual completion or failure of an asynchronous operation, and they allow you to handle asynchronous operations in a more organized and readable manner.

When you chain promises together, you use the **.then()** method to specify what should happen after the asynchronous operation represented by the promise is completed successfully. If there is an error, you can handle it using the **.catch()** method at the end of the chain.

asyncOperation1()

.then(result1 => {

// Do something with result1

return asyncOperation2(result1); // Return another promise

})

.then(result2 => {

// Do something with result2

return asyncOperation3(result2); // Return another promise

})

.then(result3 => {

// Do something with result3

})

.catch(error => {

// Handle any errors that occurred in the chain

console.error(error);

});

Asynchronous operations in programming are tasks that do not occur immediately or in a sequential order with the rest of the program. Instead, they are initiated, and the program continues to execute without waiting for them to complete. Once the asynchronous operation finishes, it notifies the program, typically by invoking a callback function, resolving a promise, or triggering an event.

**result1** is the value resolved by the promise returned by **asyncOperation1()**.

When you call **asyncOperation1()**, it starts an asynchronous operation, and the JavaScript runtime continues to execute other code while waiting for that operation to complete. Once the operation is finished, the promise returned by **asyncOperation1()** is either fulfilled with a value or rejected with an error.

When the operation completes successfully (i.e., the promise is fulfilled), the value it resolves to is passed as an argument to the **then()** method. In this case, **result1** is the value resolved by the promise returned by **asyncOperation1()**.

Inside the **then()** callback, you can perform operations with **result1** or return another promise (**asyncOperation2(result1)**), which allows you to chain asynchronous operations together. This chaining pattern is common in JavaScript for managing asynchronous code flow.

CLASSES IN JS

class Person {

constructor(name, age) {

this.name = name;

this.age = age;

}

greet() {

console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);

}

}

const john = new Person('John', 30);

john.greet(); // Output: Hello, my name is John and I am 30 years old.

IN JAVASCRIPT YOUC ANNOT DECLAR INSTANCE VARIABLES SO YOU HAVE TO USE THIS(before the constructor at least)

ASYNC FUNCTIONS

n JavaScript, **async** is a keyword used in asynchronous programming to define functions that return promises. When you mark a function with the **async** keyword, it means the function will always return a promise, even if you don't explicitly return one.

Here's a basic example of an async function:

async function fetchData() {

return 'Data fetched';

}

In the above example, the **fetchData** function is marked as **async**, indicating that it is asynchronous. Even though there is no explicit **return new Promise(...)** statement, the function implicitly returns a promise that resolves to **'Data fetched'**.

Async functions allow you to use the **await** keyword inside them to pause execution until a promise is resolved or rejected. This makes asynchronous code look and behave more like synchronous code, which can be easier to read and write.

Here's an example of using **await** inside an async function:

async function fetchData() {

let data = await fetch('https://api.example.com/data');

let json = await data.json();//await is necessary here also as we use await with data

return json;

}

In the above example, **await** is used to pause execution until the **fetch** function returns a promise that resolves with the response. Then, another **await** is used to pause execution until the **json()** method returns a promise that resolves with the parsed JSON data. Finally, the parsed JSON data is returned from the async function.

Async functions provide a cleaner and more concise syntax for working with asynchronous code compared to traditional promise-based code or using callbacks. They are widely used in modern JavaScript development, especially in web applications for handling asynchronous operations such as fetching data from APIs, reading files, or making network requests.

DESTRUCTURING OBJECT:

export const register= async (req,res)=>{

    //async as call to database is asynchronous

    try{

        const {firstName,

        lastName,

        email,

        password,

        picturePath,

        friends,

        location,

        occupation} = req.body;

    }

};

HERE req is an object, body is one of the properties of the object and the corresponding values are passed on to the attributes enclosed in the lhs object. Const is necessary

Knowing when to use **await** depends on whether the function or method you're calling returns a promise or not. In JavaScript, when you're dealing with asynchronous operations, such as fetching data from a database, making HTTP requests, or using libraries like bcrypt for hashing passwords, you'll typically encounter functions or methods that return promises.

Here are some guidelines to help you decide when to use **await**:

1. **Async Functions**: If you're calling an asynchronous function within an **async** function, you can use **await** to pause execution until the promise returned by the asynchronous function resolves or rejects.

async function myAsyncFunction() {

const result = await someAsyncOperation();

// Use result here

}

**Promises**: If you're calling a function or method that returns a promise (either explicitly or implicitly), you can use **await** to wait for that promise to resolve.

async function myAsyncFunction() {

const result = await somePromiseReturningFunction();

// Use result here

}

**Non-Promise Operations**: If you're calling a synchronous function or performing an operation that doesn't return a promise, you don't need to use **await**.

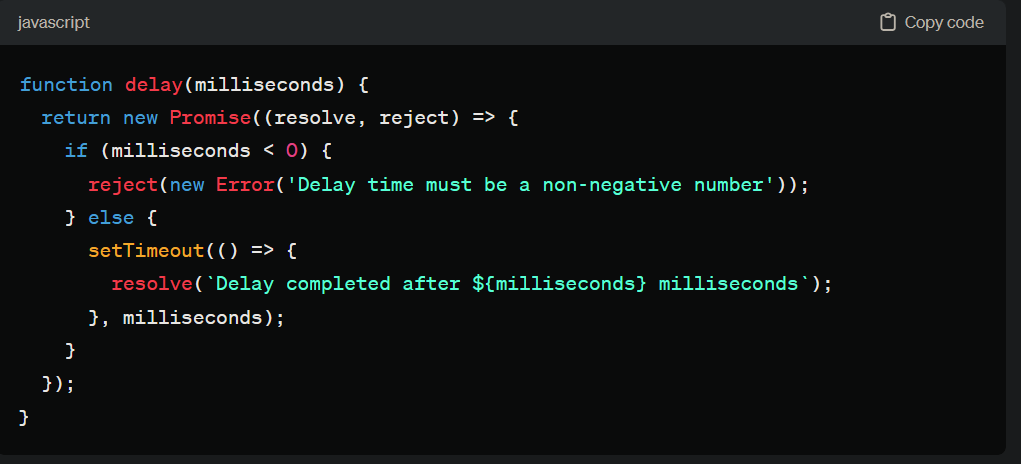
async function myAsyncFunction() {

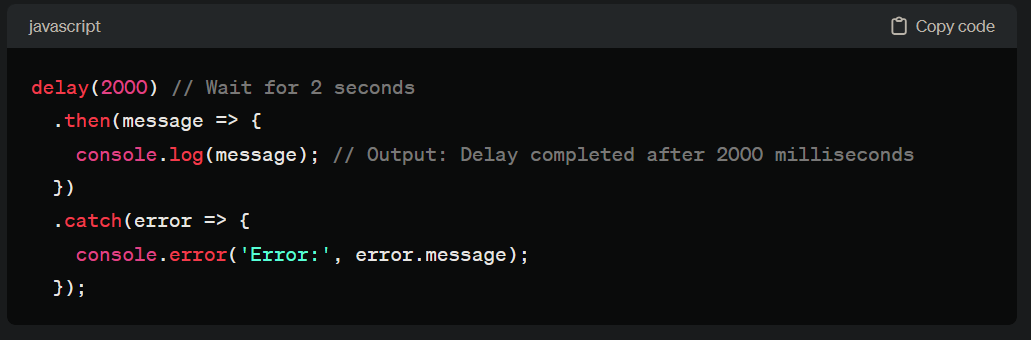
const result = someSynchronousOperation();

// Use result here

}

PROMISE FUNCTION EXAMPLE(FUNCTION RETURNS PROMISE)





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

// Asynchronous operation

setTimeout(() => {

resolve('Operation successful');

}, 2000);

});

1. For this particular code: **resolve(value):** This function is used to fulfill the Promise. When called with a value (which can be any JavaScript value, including undefined), it transitions the Promise from the pending state to the fulfilled state, indicating that the asynchronous operation represented by the Promise has been successfully completed. Any value passed to **resolve** will be available as the result of the Promise when it is consumed using **then()**.
2. **reject(reason):** This function is used to reject the Promise. When called with a reason (usually an Error object or a string describing the reason for the rejection), it transitions the Promise from the pending state to the rejected state, indicating that the asynchronous operation encountered an error or was unsuccessful. The reason passed to **reject** will be available as the error when the Promise is consumed using **catch()**.

In this code, **resolve('Operation successful')** is called inside the **setTimeout** callback function after a delay of 2000 milliseconds. This means that after 2000 milliseconds, the Promise will be fulfilled with the value **'Operation successful'**, transitioning it from the pending state to the fulfilled state.

Now, regarding **setTimeout**:

1. **setTimeout(callback, delay):** This is a built-in JavaScript function that is used to schedule the execution of a callback function after a specified delay (in milliseconds). It does not block the execution of subsequent code; instead, it sets a timer and returns immediately. After the specified delay, the callback function is added to the execution queue and will be executed when the call stack is empty.