call BackFunctions:

A JavaScript callback is a function which is to be executed after another function has finished execution.

=>Any function that is passed as an argument to another function so that it can be executed in that other function is called as a callback function.

why we need of callback function:

=>We need callback functions because many JavaScript actions are asynchronous, which means they don't really stop the program (or a function)
 from running until they're completed, as you're probably used to.
==>Instead, it will execute in the background while the rest of the code runs.
==>A callback's primary purpose is to execute code in response to an event.
==> These events might be user-initiated, such as mouse clicks or typing.
==>With a callback, you may instruct your application to "execute this code every time the user clicks a key on the keyboard."

ex:

```
const button = document.getElementById('button');
function callback(){
console.log("Hello world");
}

button.addEventListener('click',callback);
```

Above code :
==>in the above code, we add addEventListener as a function and we are passing another function callback as an argument.
 ==>And when a click event is triggered the addEventListener registers the callback function.

Using Promises:
==========
==>We must generate a new promise for each callback in order to convert them to promises.
==>When the callback is successful, we may resolve the promise.
==>If the callback fails, we can reject the promise.

ex:

```
function getUserPromise {
  const newUser = getUserFromApi(user)
  return new Promise((resolve, reject) => {
    if (user) {
      resolve(user)
    } else {
      reject(new Error('No more new users!'))
    }
  })
}
```

By using Async-Await:
==========
=>We have already seen how to create an asynchronous callback in the above section.

=>We can write asynchronous functions as if they are synchronous (executed sequentially) with the use of await as it stops the
=> execution until the promise is resolved (function execution is successful).

ex:

```
const userProfile = async () => {
  const user = await fetchUsers(1) // argument indicated number of users to fetch
  const updatedAddress = await updateAddress(user)
  const pincode = await getPincode()
  const updateUser = await updateUser(user, updatedAdress, pincode)
  return user
}

// fetch and update user profile
userProfile()
```

Uses of Callback Function:

==>Callback functions are needed because many JavaScript actions are asynchronous.
 ===>Instead, it executes in the background while the rest of the code runs.
 ==>A callback's purpose is to execute code in response to an event.
 ==>These events can be like mouse clicks; with callback, we can add text at the end of each statement like"execute this code every
 ==>time the user clicks a key on the keyboard."

==========Promises::==========
=======

**Imagine a function waiting for a resource or performing a network request. This

takes time, and meanwhile, the entire execution freezes
**To overcome this drawback, asynchronous programming comes into play.
  =>With the help of features like Callbacks, Promises, and Async/Await,
  => you can perform time-consuming actions without affecting the main execution thread

what is Promises:

**A promise is an asynchronous action that may complete at some point in the future and produce a value.
**This value is not necessarily known at the time of its creation. Once the value is produced, it notifies the use

**Promises provide a robust way to wrap the result of asynchronous work, overcoming the problem of deeply nested callbacks.

syntax:
========

```
let some_action = new Promise(function(resolve, reject)

{

  // Perform some work

})
```

**The Promise object takes a callback function as a parameter, which, in turn, takes two parameters, resolve and reject.
**The promise is either fulfilled or rejected.

There are three states governing promises:

Pending - The promise is pending when it's created. This means that the promise is yet to be fulfilled and the underlying operation is not yet performed.
Fulfilled - This state indicates that the operation has finished and the promise is fulfilled with a value.
Rejected - This means that an error occurred during the process and the promise has been denied.

It’s known that if a promise is fulfilled, then some piece of code must be executed.

And if it's rejected, then error handling must be performed. So for this,

 there are two methods:

1. then(callback) is invoked to attach a callback when the promise is resolved/fulfilled.

```
.this(function(result)
{

// handle success
 }
```

2. catch(callback) method is invoked to attach a callback when the promise is rejected.

```
.catch(function(error)
{

//handle error
}
```

*Once a promise is settled (either fulfilled or rejected), it's said to be immutable and cannot change its state.

exa:
=====>

```
<script>

  let car = new Promise(function(resolve,reject){

fuel_fullTank = true;

           if(fuel_fullTank)

           resolve()

     else

                reject()

             });

  car.then(function(){

       document.write("The fuel tank is full. Happy Driving!!")

}).catch(function(){

             document.write("The fuel tank is not empty.")
```

```
})

</script>
```

==========Async-Await==========

*Here, we will talk about one of its key features—JavaScript async/await, which are asynchronous programming constructs.
* Async and await build on top of promises and generators to express asynchronous actions inline

what is Synchrnous programming:

Synchronous programming includes the sequential execution of functions and processes.
If a particular function needs a resource, execution stops until the currently executing function fetches the resource and finishes its process.

==>it’s time-consuming and doesn’t use the system to its full potential.
Asynchronous programming solves these issues.

Asynchronous Programming?:

**Asynchronous programming, on the other hand, ensures that the execution does not stop if a function is performing other operations.
**Instead, the execution continues normally until the function is called back again.

**To facilitate this, concepts like Callbacks, Promises, and Async/await are used.

JavaScript Async Functions:

**Async and await are built on promises. The keyword “async” accompanies the function, indicating that it returns a promise.
*Within this function, the await keyword is applied to the promise being returned.
* The await keyword ensures that the function waits for the promise to resolve. On the surface, the execution looks synchronous, but it is asynchronous.
*The function’s execution is blocked at the await keyword’s placement in the coding.

**Async functions make the code more readable and are easier to handle than promises.

**The use case includes two functions that return promises to verify if you have passed or failed a hypothetical exam and the grade that you have received on that exam.

ex:

```
let result = function(score){

                return new Promise(function(resolve,reject)

                {

                console.log("Gathering results...");

                if(score>50)

                    resolve("Congratulations! You have Passed" )

                else

                    reject("You have failed")

                })

            }
```

==>The function is defined as a value with the name result. It uses a parameter called score that the user provides.
The function returns a promise, which resolves or rejects based on the score.

```
 let grade = function(response){

                return new Promise(function(resolve,reject)

                {

                console.log("Calculating your grade..");

                resolve("Your grade is A. " + response )

                })

            }
```

==>The next function indicates the score secured. This function uses a parameter called response. The promise gets resolved with a message.

```
result(80).then(response =>{
                console.log("Results received")
                    return grade(response)
            }).then(finalgrade => {
                console.log(finalgrade)
            }).catch(err => {
                    console.log(err)
            })
```

==>The code above calls the functions.
  The result function is called with a value 80.
     This is done to show how the code executes when the promise is resolved.
           The .then method catches the response and indicates that the results are received.
                  To call the grade function, the parameter response is passed as an argument.
                     Since the grade function returns a promise, we can use the .then method over it and catch the result of this promise in finalgrade.
==>Finally, to check for errors, we have the .catch method, which executes if the promise is rejected.

=>Using async and await helps with code readability, and can help users avoid complicated coding outputs.

The following is how async and await are implemented:

```
async function calculateResults(){
                try {
                    const response = await result(80)
                    console.log("Results received")
                    const finalgrade = await grade(response)
```

```
                    console.log(finalgrade)

                }

            catch(err){

                    console.log(err)

                }

            }


        calculateResults()
```

**We've created an async function called calculateResults().
The function result is called with a preceding await keyword.
The result of this function is stored in the variable response.
Consecutively, the function grade is called with the argument response.
The result of this function is stored in the variable finalgrade.
This set of code is enclosed within the try block and in case the promise is rejected, the control enters the catch block to display the error message

**Finally, the async function calculateResults() is called.