Synchronous Vs Asynchronous

Synchronous : Every statement of code get executed one by one

So basically, a statement has to wait for earlier statement to get executed

Eg - console.log ("I");

console.log("eat");

console.log("ice-cream");

It will print I first,
then eat,
after that ice-cream

Asynchronous : It allows program to be executed immediately without blocking the code. Unlike the Synchronous method It doesn't wait for earlier statement to get executed first.

Each task execute completed Independently.

Eg- console.log ("I");

setTimeout (() => {
console.log ("eat"); }, 2000)

Console.log ("Ice Cream")

It will print
"I"
"Ice Cream" (will execute immediately)
"eat" (will print after 2s)

Asynchronous Functions

→ It contains async keyword.

How to use in Normal Function declaration

```
async function name (arg) {
}
```

How to use in an arrow function

```
Cons functionName = async (arg) => {
}
```

Asynchronous functions always return promises

It doesn't matter what you return. The returned value will always be promise.

Eg →

```
const getOne = async () => {
    return 1;
}

const promise = getOne();
console.log(promise)
```

The await keyword

The await keyword lets you wait for promise to resolve. Once promise is resolved it returns the parameter passed into then call.

Eg -

Eg →

```
const getOne = async _ => {
    return 1; }

getOne().then(value => {
                console.log(value.) }); //1
```

Now use of await keyword

```
const test = async _ => {
    const one = await getOne();
    Console.log (one);
}

test()
```

We can only use await when we have async.

Let's implement the fetch API code using asynct/await :

Date: / /

```
const FetchData = async () => {
const qoutes = await fetch ("http://--/quotes");
const response = await qoutes.json();

console.log(response);

}

  FetchData();
```

We can also handle errors in async/await by using try and catch.

```
const FetchData = async () => {
try {
  const quotes = await fetch ("http://__ ");
  const response = await quotes.json();
    console.log (response);
}
  catch (error) {
      Console.log(error);
  }

};
  FetchData();
```

Date: / /

Promises In Javascript

A promise is a javascript object that allows you to make asynchoonous calls.
It produces a value when async operation completes successfully or produces an error if it doesn't complete.

You can create promise using constructor

```
let promise = new Promise (Function(resolve, reject
{
});
```

↑ Executor function

Executor fn takes 2 arguments :-

→ resolve — indicate successful completion
→ reject — indicates an error

The Promise objects and states

The promise object should be capable of informing consumers when execution has been started, completed or returned with an error

1. State → pending - When execution fn starts

Fulfilled - When promise resolved successfully

rejected - When the promise rejects

2. result →

undefined - Initially when state value is pending

value - When promise is resolved

Error When the promise is rejected

A promise that is either resolved or rejected are settled

Handling Promises by Consumer

Three important handler methods

- then()
- catch()
- Finally

These methods helps us create a link between executor and consumer

The .then() Promise Handler

It is used to let consumer know outcome of promise. It accept 2 arguements
- result
- error.

```
Eg- promise.then (
    (result) => {
        console.log (result);
    },
    (error) => {
        console.log (error);
    }
);
```

The catch Promise Handler

To handle Error (rejections) from Promises. It's better syntax to handle Error than handling it with :then()

```
Eg→ promise.catch (function (Error) {
        console.log (Error);
    });
```

The .finally() Promise Handler

The finally() handler method performs cleanups like stopping a loader, closing a live connection and so on

Irrespective of whether promise resolve or rejects, the finally() method will run

Eg - promise.finally(() => {

    console.log("Promise settled");
}).then((result) => {
        console.log({result});
});

Imp point to note,
the finally() method passes through result or error to the next handler which can call a .then() or .catch() again.