Agenda:

- Concurrency in Javascript
- CallBack Function in Javascript
- Callback Hell or Pyramid of Doom
- Promises in Javascript
- Creating a Promise
- then () & Catch() method in Javascript
- Chaining multiple promises
- Promise methods in Javascript
- Async/Await keywords in Javascript

In this session, we are going to learn about the asynchronous behaviour in Javascript, In order to understand it let's first see that what doe Asynchronicity means ?

An *asynchronous operation* is one that allows the computer to "move on" to other tasks while waiting for the asynchronous operation to complet Asynchronous programming means that time-consuming operations don't have to bring everything else in our programs to a halt.

There are countless examples of asynchronicity in our everyday lives. Cleaning our house, for example, involves asynchronous operations such as dishwasher washing our dishes or a washing machine washing our clothes. While we wait on the completion of those operations, we're free to do oth chores.

Similarly, web development makes use of asynchronous operations. Operations like making a network request or querying a database can be tim consuming, but JavaScript allows us to execute other tasks while awaiting their completion.

Now, let's understand it better by learning about concurrency in javascript:

Concurrency in Javascript:

As you should know by now, JavaScript runs on a single thread. This thread is event-based and responds to events when they occur. So how does it n block other functions from executing? Well, the answer is simple, it does. JS functions are hoisted, as pointed out by my friend Featherweight, functior declared in variables will not execute if called before being defined as the variable doesn't exist, while normal functions do execute anyways as they' hoisted.

```
sayHi()
function sayHi() {
 console.log("Hello")
}//this will worksayHello()
let sayHello = () => {
 console.log("hello")
}//this will not work
```

So, this means JavaScript code is parsed sequentially. Which is the direct opposite of what we're trying to achieve. Fortunately, JavaScript comes wi three features that allow you to run code in concurrently. **Callbacks, Promises**and **Async/Await**.

So, understanding it all , let's now begin to understand or learn about the three features that Javascript provides to run code in concurrently, starting wi Callbacks:

CallBacks in Javascript:

In JavaScript, functions are first-class citizens. Therefore, you can pass a function to another function as an argument.

By definition, a callback is a function that you pass into another function as an argument for executing later.

Let's try to understand above given statements with help of examples:

As now you all know about the high order filter() methods in Javascript, let's take example around it to understand callbacks:

Consider the example shown below, the given example defines a `filter()` function that accepts an array of numbers and returns a new array of oc numbers:

```
function filter(numbers) {
  let results = [];
  for (const number of numbers) {
    if (number % 2 != 0) {
      results.push(number);
    }
```

```
    }
    return results;
}
let numbers = [1, 2, 4, 7, 3, 5, 6];
console.log(filter(numbers));
```

How it works.

- First, define the `filter()` function that accepts an array of numbers and returns a new array of the odd numbers.
- Second, define the `numbers` array that has both odd and even numbers.
- Third, call the `filter()` function to get the odd numbers out of the numbers array and output the result.

If you want to return an array that contains even numbers, you need to modify the `filter()` function. To make the `filter()` function more gener and reusable, you can:

- First, extract the logic in the `if` block and wrap it in a separate function.
- Second, pass the function to the filter() function as an argument.

Here's the updated code:

```
function isOdd(number) {
  return number % 2 != 0;
}

function filter(numbers, fn) {
  let results = [];
  for (const number of numbers) {
    if (fn(number)) {
      results.push(number);
    }
  }
  return results;
}
let numbers = [1, 2, 4, 7, 3, 5, 6];
console.log(filter(numbers, isOdd));
```

The result is the same. However, you can pass any function that accepts an argument and returns a boolean value to the second argument the `filter()` function.

By definition, the `isOdd` is a callback function or callback. Because the `filter()` function accepts a function as an argument, it's called a *high-ord function*.

There are two types of callbacks: synchronous and asynchronous callbacks

Synchronous Callbacks:

A synchronous callback is executed during the execution of the high-order function that uses the callback. The `isOdd` is an example of synchronou callbacks because they execute during the execution of the `filter()` function.

Asynchronous Callbacks:

An asynchronous callback is executed after the execution of the high-order function that uses the callback.

Asynchronicity means that if JavaScript has to wait for an operation to complete, it will execute the rest of the code while waiting.

Note that JavaScript is a single-threaded programming language. It carries asynchronous operations via the callback queue and event loop.

Let's consider an example, suppose that you need to develop a script that downloads a picture from a remote server and process it after the downloa completes:

```
function download(url) {
// ...
}

function process(picture) {
// ...
}
```

```
download(url);
process(picture);
```

However, downloading a picture from a remote server takes time depending on the network speed and the size of the picture.

The following `download()` function uses the `setTimeout()` function to simulate the network request:

```
function download(url) {
    setTimeout(() => {
// script to download the picture hereconsole.log(`Downloading ${url} ...`);
    },1000);
}
```

And this code emulates the `process()` function:

```
function process(picture) {
    console.log(`Processing ${picture}`);
}
```

When you execute the following code:

```
let url = 'https://www.almabetter.net/pic.jpg';

download(url);
process(url);
```

you will get the following output:

```
Processing https://almabetter.net/pic.jpg
Downloading https://almabetter.net/pic.jpg ...
```

This is not what you expected because the `process()` function executes before the `download()` function. The correct sequence should be:

- Download the picture and wait for the download completes.
- Process the picture.

To resolve this issue, you can pass the `process()` function to the `download()` function and execute the `process()` function inside the `download()` function once the download completes, like this:

```
function download(url, callback) {
    setTimeout(() => {
// script to download the picture hereconsole.log(`Downloading ${url} ...`);

// process the picture once it is completed
        callback(url);
    }, 1000);
}

function process(picture) {
    console.log(`Processing ${picture}`);
}

let url = 'https://wwww.almabetter.net/pic.jpg';
download(url, process);
```

Output:

```
Downloading https://www.almabetter.net/pic.jpg ...
Processing https://www.almabetter.net/pic.jpg
```

Now, it works as expected.

In this example, the `process()` is a callback passed into an asynchronous function.

When you use a callback to continue code execution after an asynchronous operation, the callback is called an asynchronous callback.

Callback Hell or Pyramid of Doom:

Nesting many asynchronous functions inside callbacks is known as the **pyramid of doom** or the **callback hell.**

However, this callback strategy does not scale well when the complexity grows significantly.

Consider the example shown below:

```
asyncFunction(function(){
    asyncFunction(function(){
        asyncFunction(function(){
            asyncFunction(function(){
                asyncFunction(function(){
                    ....
                });
            });
        });
    });
});
```

To avoid the pyramid of doom, you use promises or async/await functions.

As we have the problem of callback hell that is where Promises in Javascript come into existence,

So, let's learn about Promises:

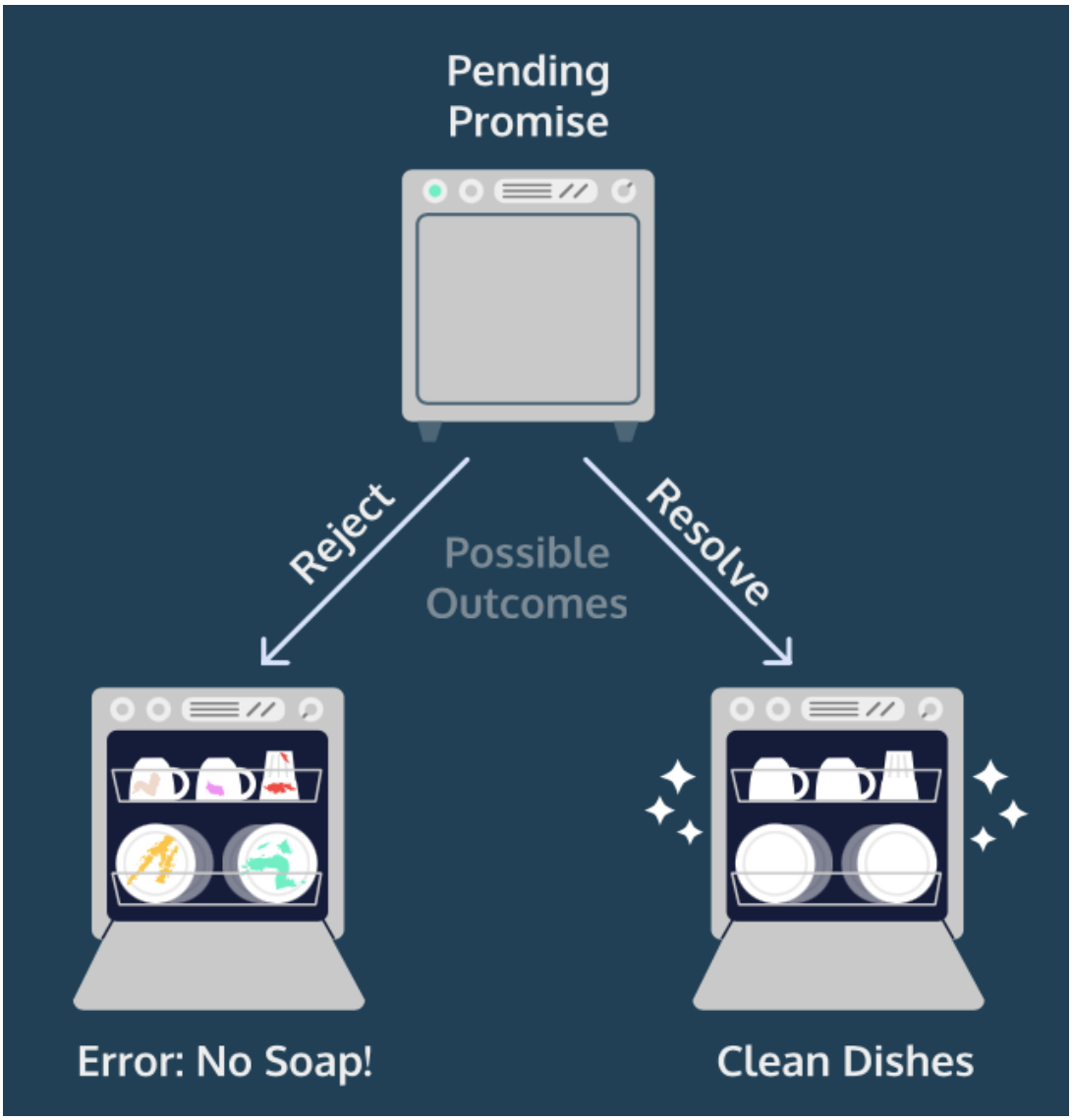
Promises in Javascript:

Promises are objects that represent the eventual outcome of an asynchronous operation. A `Promise` object can be in one of three states:

- **Pending**: The initial state— the operation has not completed yet.
- **Fulfilled**: The operation has completed successfully and the promise now has a *resolved value*. For example, a request's promise might resolve with a JSON object as its value.
- **Rejected**: The operation has failed and the promise has a reason for the failure. This reason is usually an `Error` of some kind.

We refer to a promise as *settled* if it is no longer pending— it is either fulfilled or rejected. Let's think of a dishwasher as having the states of a promise:

- **Pending**: The dishwasher is running but has not completed the washing cycle.
- **Fulfilled**: The dishwasher has completed the washing cycle and is full of clean dishes.
- **Rejected**: The dishwasher encountered a problem (it didn't receive soap!) and returns unclean dishes.



If our dishwashing promise is fulfilled, we'll be able to perform related tasks, such as unloading the clean dishes from the dishwasher. If it's rejected, w can take alternate steps, such as running it again with soap or washing the dishes by hand.

Creating a Promise:

Let's construct a promise! To create a new `Promise` object, we use the `new` keyword and the `Promise` constructor method:

```
const executorFunction = (resolve, reject) => { };
const myFirstPromise = new Promise(executorFunction);
```

The `Promise` constructor method takes a function parameter called the *executor function* which runs automatically when the constructor is called. Th executor function generally starts an asynchronous operation and dictates how the promise should be settled.

The executor function has two function parameters, usually referred to as the `resolve()` and `reject()` function The `resolve()` and `reject()` functions aren't defined by the programmer. When the `Promise` constructor runs, JavaScript will pass i **own** `resolve()` and `reject()` functions into the executor function.

- `resolve` is a function with one argument. Under the hood, if invoked, `resolve()` will change the promise's status from `pending` to `fulfilled`, and the promise's resolved value will be set to the argument passed into `resolve()`.
- `reject` is a function that takes a reason or error as an argument. Under the hood, if invoked, `reject()` will change the promise's status from `pending` to `rejected`, and the promise's rejection reason will be set to the argument passed into `reject()`.

Consider the example shown below:

```
const executorFunction = (resolve, reject) => {
  if (someCondition) {
      resolve('I resolved!');
  } else {
```

```
        reject('I rejected!');
    }
}
const myFirstPromise = new Promise(executorFunction);
```

Let's break down what's happening above:

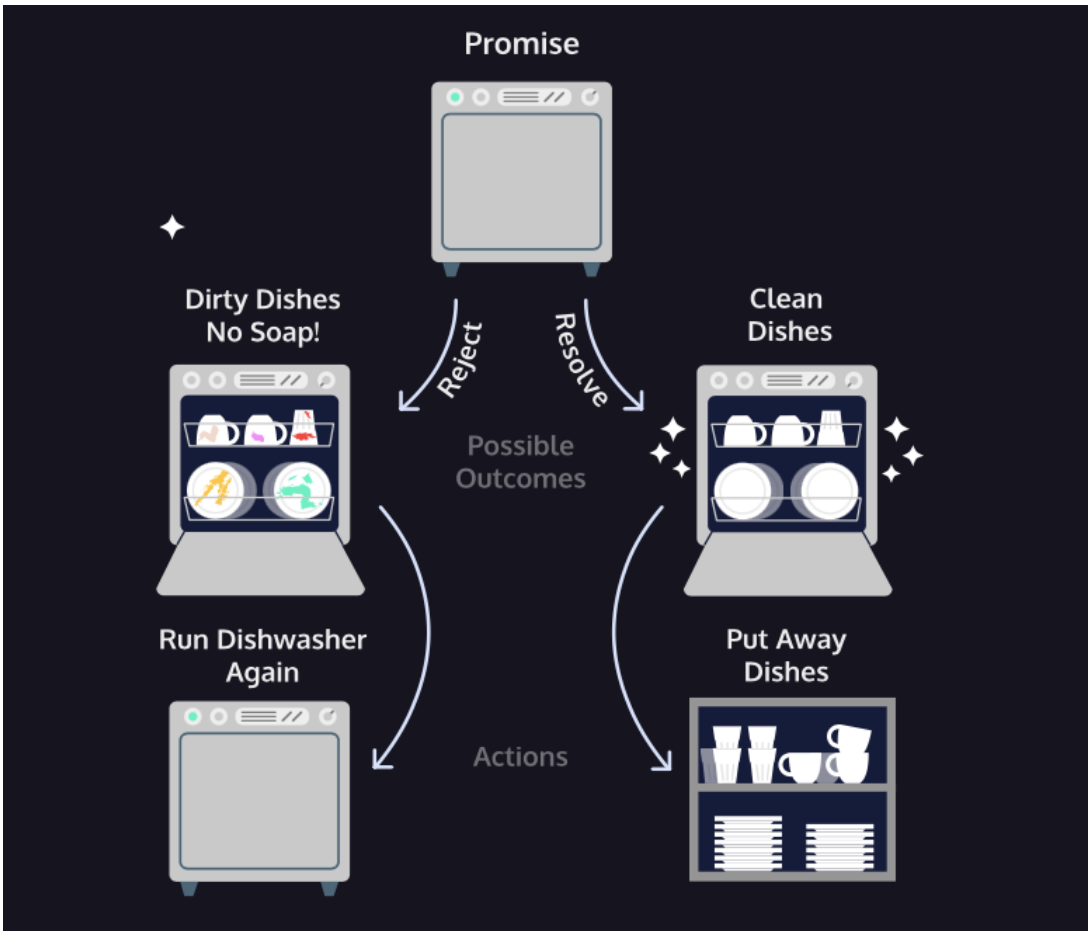
- We declare a variable `myFirstPromise`
- `myFirstPromise` is constructed using `new Promise()` which is the `Promise` constructor method.
- `executorFunction()` is passed to the constructor and has two functions as parameters: `resolve` and `reject` .
- If `someCondition` evaluates to `true` , we invoke `resolve()` with the string `'I resolved!'`
- If not, we invoke `reject()` with the string `'I rejected!'`

Consuming Promises:

The initial state of an asynchronous promise is `pending` , but we have a guarantee that it will settle. How do we tell the computer what should happe then? Promise objects come with an aptly named `.then()` method. It allows us to say, "I have a promise, when it settles, **then** here's what I want happen…"

In the case of our dishwasher promise, the dishwasher will run **then**:

- If our promise rejects, this means we have dirty dishes, and we'll add soap and run the dishwasher again.
- If our promise fulfills, this means we have clean dishes, and we'll put the dishes away.



`.then()` is a higher-order function— it takes two callback functions as arguments. We refer to these callbacks as *handlers*. When the promise settle the appropriate handler will be invoked with that settled value.

- The first handler, sometimes called `onFulfilled` , is a *success handler*, and it should contain the logic for the promise resolving.
- The second handler, sometimes called `onRejected` , is a *failure handler*, and it should contain the logic for the promise rejecting.

We can invoke `.then()` with one, both, or neither handler! This allows for flexibility, but it can also make for tricky debugging. If the appropriate handl is not provided, instead of throwing an error, `.then()` will just return a promise with the same settled value as the promise it was called on. Or important feature of `.then()` is that it always returns a promise.

Note: We can invoke `.then()` with one, both, or neither handler! This allows for flexibility, but it can also make for tricky debugging. If the appropria handler is not provided, instead of throwing an error, `.then()` will just return a promise with the same settled value as the promise it was called o One important feature of `.then()` is that it always returns a promise

To handle a "successful" promise, or a promise that resolved, we invoke `.then()` on the promise, passing in a success handler callback functio Consider the example shown below:

```
const prom = new Promise((resolve, reject) => {
  resolve('Yay!');
});

const handleSuccess = (resolvedValue) => {
  console.log(resolvedValue);
};

prom.then(handleSuccess); // Prints: 'Yay!'
```

Let's break down what's happening in the example code:

- `prom` is a promise which will resolve to `'Yay!'` .
- We define a function, `handleSuccess()` , which prints the argument passed to it.
- We invoke `prom` 's `.then()` function passing in our `handleSuccess()` function.
- Since `prom` resolves, `handleSuccess()` is invoked with `prom` 's resolved value, `'Yay'` , so `'Yay'` is logged to the console.

Catch() method with Promises:

Remember, `.then()` will return a promise with the same settled value as the promise it was called on if no appropriate handler was provided. Th implementation allows us to separate our resolved logic from our rejected logic. Instead of passing both handlers into one `.then()` , we can chain second `.then()` with a failure handler to a first `.then()` with a success handler and both cases will be handled.

```
prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
  .then(null, (rejectionReason) => {
    console.log(rejectionReason);
  });
```

Since JavaScript doesn't mind whitespace, we follow a common convention of putting each part of this chain on a new line to make it easier to read. T create even more readable code, we can use a different promise function: `.catch()` .

The `.catch()` function takes only one argument, `onRejected` . In the case of a rejected promise, this failure handler will be invoked with the reasc for rejection. Using `.catch()` accomplishes the same thing as using a `.then()` with only a failure handler.

Let's look at an example using `.catch()`

```
prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });
```

Let's break down what's happening in the example code:

- `prom` is a promise which randomly either resolves with `'Yay!'` or rejects with `'Ohhh noooo!'` .
- We pass a success handler to `.then()` and a failure handler to `.catch()` .
- If the promise resolves, `.then()` 's success handler will be invoked with `'Yay!'` .
- If the promise rejects, `.then()` will return a promise with the same rejection reason as the original promise and `.catch()` 's failure handler will be invoked with that rejection reason.

Chaining multiple Promises:

Let's illustrate this with another cleaning example, washing clothes:

We take our dirty clothes and put them in the washing machine. If the clothes are cleaned, **then** we'll want to put them in the dryer. After the dryer runs, the clothes are dry, **then** we can fold them and put them away.

This process of chaining promises together is called *composition*. Promises are designed with composition in mind! Here's a simple promise chain code:

```
firstPromiseFunction()
.then((firstResolveVal) => {
  return secondPromiseFunction(firstResolveVal);
})
.then((secondResolveVal) => {
  console.log(secondResolveVal);
});
```

Let's break down what's happening in the example:

- We invoke a function `firstPromiseFunction()` which returns a promise.
- We invoke `.then()` with an anonymous function as the success handler.
- Inside the success handler we **return** a new promise— the result of invoking a second function, `secondPromiseFunction()` with the first promise's resolved value.
- We invoke a second `.then()` to handle the logic for the second promise settling.
- Inside that `.then()` , we have a success handler which will log the second promise's resolved value to the console.

Note: In order for our chain to work properly, we had to `return` the promise `secondPromiseFunction(firstResolveVal)` . This ensured that th return value of the first `.then()` was our second promise rather than the default return of a new promise with the same settled value as the initial.

Promise.all() method:

Let's think in terms of cleaning again,

For us to consider our house clean, we need our clothes to dry, our trash bins emptied, and the dishwasher to run. We need **all** of these tasks complete but not in any particular order. Furthermore, since they're all getting done asynchronously, they should really all be happening at the same tim

To maximize efficiency we should use *concurrency*, multiple asynchronous operations happening together. With promises, we can do this with th function `Promise.all()` .

`Promise.all()` accepts an array of promises as its argument and returns a single promise. That single promise will settle in one of two ways:

- If every promise in the argument array resolves, the single promise returned from `Promise.all()` will resolve with an array containing the resolve value from each promise in the argument array.
- If any promise from the argument array rejects, the single promise returned from `Promise.all()` will immediately reject with the reason that promise rejected. This behavior is sometimes referred to as *failing fast*.

Consider the example shown below:

```
let myPromises = Promise.all([returnsPromOne(), returnsPromTwo(), returnsPromThree()]);

myPromises
  .then((arrayOfValues) => {
    console.log(arrayOfValues);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });
```

Let's break down what's happening:

- We declare `myPromises` assigned to invoking `Promise.all()` .
- We invoke `Promise.all()` with an array of three promises— the returned values from functions.
- We invoke `.then()` with a success handler which will print the array of resolved values if each promise resolves successfully.
- We invoke `.catch()` with a failure handler which will print the first rejection message if any promise rejects.

Async/Await Keywords in Javascript:

The Async keyword:

The `async` keyword is used to write functions that handle asynchronous actions. We wrap our asynchronous logic inside a function prepended wi the `async` keyword. Then, we invoke that function.

```
async function myFunc() {
  // Function body here
};

myFunc();
```

We'll be using `async` function declarations throughout this lesson, but we can also create `async` function expressions:

```
const myFunc = async () => {
  // Function body here
};

myFunc();
```

`async` functions always return a promise. This means we can use traditional promise syntax, like `.then()` and `.catch` with our `async` function An `async` function will return in one of three ways:

- If there's nothing returned from the function, it will return a promise with a resolved value of `undefined` .
- If there's a non-promise value returned from the function, it will return a promise resolved to that value.
- If a promise is returned from the function, it will simply return that promise

Consider the example shown below:

```
async function fivePromise() {
  return 5;
}

fivePromise()
.then(resolvedValue => {
    console.log(resolvedValue);
  })  // Prints 5
```

In the example above, even though we return `5` inside the function body, what's actually returned when we invoke `fivePromise()` is a promise wi a resolved value of `5` .

The Await Operator:

The `await` keyword can only be used inside an `async` function. `await` is an operator: it returns the resolved value of a promise. Since promise resolve in an indeterminate amount of time, `await` halts, or pauses, the execution of our `async` function until a given promise is resolved.

In most situations, We can `await` the resolution of the promise it returns inside an `async` function.

Consider the example shown below:

```
async function asyncFuncExample(){
  let resolvedValue = await myPromise();
  console.log(resolvedValue);
}

asyncFuncExample(); // Prints: I am resolved now!
```

In the example above, `myPromise()` is a function that returns a promise which will resolve to the string `"I am resolved now!"` . With our `async` function, `asyncFuncExample()` , we use `await` to halt our execution until `myPromise()` is resolved and assign its resolved value the variable `resolvedValue` . Then we log `resolvedValue` to the console. We're able to handle the logic for a promise in a way that reads lik synchronous code.

Handling Dependent Promises:

The true beauty of `async...await` is when we have a series of asynchronous actions which depend on one another. For example, we may make network request based on a query to a database. In that case, we would need to wait to make the network request until we had the results from th database. With native promise syntax, we use a chain of `.then()` functions making sure to return correctly each one. For this consider the examp given below:

```
function nativePromiseVersion() {
  returnsFirstPromise()
    .then((firstValue) => {
      console.log(firstValue);
      return returnsSecondPromise(firstValue);
    })
   .then((secondValue) => {
      console.log(secondValue);
    });
}
```

Let's break down what's happening in the `nativePromiseVersion()` function:

- Within our function we use two functions which return promises: `returnsFirstPromise()` and `returnsSecondPromise()` .
- We invoke `returnsFirstPromise()` and ensure that the first promise resolved by using `.then()` .
- In the callback of our first `.then()` , we log the resolved value of the first promise, `firstValue` , and then return `returnsSecondPromise(firstValue)` .
- We use another `.then()` to print the second promise's resolved value to the console.

Now, let's see this using async … await method:

```
async function asyncAwaitVersion() {
  let firstValue = await returnsFirstPromise();
  console.log(firstValue);
  let secondValue = await returnsSecondPromise(firstValue);
  console.log(secondValue);
}
```

Let's break down what's happening in our `asyncAwaitVersion()` function:

- We mark our function as `async` .
- Inside our function, we create a variable `firstValue` assigned `await returnsFirstPromise()` . This means `firstValue` is assigned the resolved value of the awaited promise.
- Next, we log `firstValue` to the console.
- Then, we create a variable `secondValue` assigned to `await returnsSecondPromise(firstValue)` . Therefore, `secondValue` is assigned this promise's resolved value.
- Finally, we log `secondValue` to the console.

Though using the `async...await` syntax can save us some typing, the length reduction isn't the main point. Given the two versions of the functio the `async...await` version more closely resembles synchronous code, which helps developers maintain and debug their cod The `async...await` syntax also makes it easy to store and refer to resolved values from promises further back in our chain which is a much mo difficult task with native promise syntax.

Handling Errors:

When `.catch()` is used with a long promise chain, there is no indication of where in the chain the error was thrown. This can make debuggir challenging.

With `async...await` , we use `try...catch` statements for error handling. By using this syntax, not only are we able to handle errors in the sam way we do with synchronous code, but we can also catch both synchronous and asynchronous errors. This makes for easier debugging!

Consider the example shown below:

```
async function usingTryCatch() {
 try {
   let resolveValue = await asyncFunction('thing that will fail');
   let secondValue = await secondAsyncFunction(resolveValue);
 } catch (err) {
```

```
    // Catches any errors in the try block
    console.log(err);
  }
}

usingTryCatch();
```

In the above example, we are using try…catch block in the async function, In the try block we are awaiting for the result of our request and if an err occurs then the flow moves into the catch block where the error is handled.

Await Promise.all() method:

Another way to take advantage of concurrency when we have multiple promises which can be executed simultaneously to `await` a `Promise.all()` .

We can pass an array of promises as the argument to `Promise.all()` , and it will return a single promise. This promise will resolve when all of th promises in the argument array have resolved. This promise's resolve value will be an array containing the resolved values of each promise from th argument array.

Consider the example shown below:

```
async function asyncPromAll() {
  const resultArray = await Promise.all([asyncTask1(), asyncTask2(), asyncTask3(), asyncTask4()]);
  for (let i = 0; i<resultArray.length; i++){
    console.log(resultArray[i]);
  }
}
```

In our above example, we `await` the resolution of a `Promise.all()` . This `Promise.all()` was invoked with an argument array containing fo promises (returned from required-in functions). Next, we loop through our `resultArray` , and log each item to the console. The first eleme in `resultArray` is the resolved value of the `asyncTask1()` promise, the second is the value of the `asyncTask2()` promise, and so on.

Note: `Promise.all()` allows us to take advantage of asynchronicity— each of the four asynchronous tasks can proces concurrently. `Promise.all()` also has the benefit of *failing fast*, meaning it won't wait for the rest of the asynchronous actions to complete once ar one has rejected. As soon as the first promise in the array rejects, the promise returned from `Promise.all()` will reject with that reason. As it wa when working with native promises, `Promise.all()` is a good choice if multiple asynchronous tasks are all required, but none must wait for any oth before executing.

Conclusion:

In this session we have learned about:

- Asynchronicity in Javascript
- Callbacks and callback hells
- Promises and their handling
- And at last about async/await way of handling promises.

So, now after learning about asynchronous communication in Javascript from the next session we will learn about HTTP calls and AJAX.

Interview Questions

> What is your understanding of the Event Loop concept in JavaScript?

The Event Loop is a mechanism used by JavaScript to handle asynchronous events. It is a continuous loop that checks for events and then processe them accordingly. This allows JavaScript to handle multiple events at the same time and makes it possible for things like animations and user input to b processed without blocking the main thread of execution.

> Why are callbacks not recommended for most applications?

Thank You !