Class 14 - async programming 3

Agenda

Introduction to Async/Await

The Basics of Async/Await

Error Handling with Async/Await

Practical Examples of Async/Await

Best Practices and Common Pitfalls

title: Introduction to Async/Await

1. What is Async/Await?
    a. async/await is a modern JavaScript syntax feature that simplifies writing asynchronous code, making it more readable and easier to debug. Introduced in ES2017, it builds upon promises, providing a cleaner, more elegant syntax for working with asynchronous operations.
2. Why Async/Await?
    a. Asynchronous operations are fundamental in JavaScript for non-blocking operations, such as fetching data from a server or reading files. While promises significantly improved asynchronous code readability, async/await further simplifies the syntax, allowing developers to write code that looks synchronous while executing asynchronously.

title: The Basics of Async/Await

Desc: how to declare async functions and use the await keyword to pause function execution until a promise settles.

1. Declaring an Async Function
    a. An async function returns a promise, automatically wrapping non-promise values in a promise.

```
async function fetchData() {
   return 'data';
 }
  const dataPromise = fetchData()
 console.log(dataPromise)
```

    b. Here in this function we have initalized it with using the async keyword ,Whenever a function is initialized with a async keyword it always returns a Promise , although you can see we are just returing a simple data value but it will be returned with a promise
    c. Output

Promise { 'data' }

Calling fetchData() returns a promise that can be resolved with 'data'.

Now how to resolve this promise?

We already know that to resolve a promise you can use the **then** method

So you can resolve it like

```
async function fetchData() {
 return "data";
}

const dataPromise = fetchData();
console.log(dataPromise);

dataPromise.then((res) => console.log(res));
```

Using Await

1. The combination of async and await is used to handle promises ,

2. I am creating a normal function (not async) and try to resolve a promise within the function

3. See the code snippet

```
const p = new Promise(function (resolve, reject) {
 resolve("Promise Resolved");
});

function fetchData() {
 p.then(function (res) {
   console.log(res);
 });
}

fetchData();
```

4. Now let's suppose I want to handle this promise within an async function

5. Using async / await

```
async function handlePromise(){
   const val = await p
   console.log(val)
}

handlePromise();
```

output

Promise Resolved

important things to note

6. Await can only be used inside an async function
7. You write the await keyword infornt of a Promise and it handles it

8. Let us add a time delay in async operation and then see
9.

```
const p = new Promise((resolve, reject) => {
 setTimeout(() => {
   resolve("Promise Resolved");
 }, 3000);
});

// async function handlePromise() {
//   const val = await p
//   console.log(val)
// }

// handlePromise() // This code to be used later

function fetchData() {
 // JS engine will not wait for the promise
 p.then((res) => console.log(res));
 console.log("Create Impact");
}

fetchData();
```

10. the promise will now wait in the event loop and after 3 seconds it will be resolved , The event loop will do it's job perfectly
11. Let us resolve the same with async / await

```
const p = new Promise((resolve, reject) => {
 setTimeout(() => {
   resolve("Promise Resolved");
 }, 10000);
});

async function handlePromise() {
 // JS engine waits for the promise to get resolved and
then moves forward
 const val = await p;
 console.log("Create Impact");
 console.log(val);
}

handlePromise();

// function fetchData() {
//    p.then((res)=> console.log(res))
//    console.log("Create Impact")
// }

//   fetchData()
```

12. Here the whole code inside the async function will now wait for the promise to get resolved and when it resolves after 3 seconds then only it will move forward to execute the next lines

13. Here, await pauses handlePromise until p is resolved. This approach simplifies asynchronous control flow, making it more akin to synchronous code in appearance and readability.

Case with two promises

Code

```
const p = new Promise((resolve, reject) => {
 setTimeout(() => {
   resolve("Promise Resolved");
 }, 3000);
});

async function handlePromise() {
 // JS engine waits for the promise to get resolved and then
moves forward

 console.log("Scaler");

 const val = await p;

 console.log("Create Impact 1");
 console.log(val);

 const val2 = await p;

 console.log("Create Impact 2");
 console.log(val2);
}

handlePromise();

// function fetchData() {
//   p.then((res)=> console.log(res))
//   console.log("Create Impact")
// }
//  fetchData()
```

Output

1. So try to figure out the output of the above code
2. Creating a Promise (p):
    a. A new Promise object is instantiated with an executor function that takes two arguments: resolve and reject.
    b. Inside the executor function, setTimeout is used to simulate an asynchronous operation that takes 3 seconds (3000 milliseconds) to complete. After 3 seconds, the resolve function is called with the string 'Promise Resolved' as its argument. This marks the promise (p) as fulfilled.
3. Defining an asynchronous function (handlePromise):
    a. An asynchronous function handlePromise is defined. This function will use the await keyword to wait for Promises to resolve before continuing with the next lines of code.
4. Execution within handlePromise:
    a. When handlePromise is called, it first executes console.log("Scaler"), immediately printing "Scaler" to the console.
    b. Next, the code encounters the await keyword before p, causing the JavaScript engine to pause execution within handlePromise until the promise p is resolved (i.e., 3 seconds later). No further code within handlePromise is executed during this waiting period.

c. Once p is resolved (after 3 seconds), its resolved value ('Promise Resolved') is assigned to val. Then, "Create Impact" is printed to the console followed by the resolved value of p ("Promise Resolved").

d. The next line also awaits the resolution of p. **However, since p has already been resolved**, JavaScript does not wait another 3 seconds. The resolved value of p is immediately assigned to val2, and then "Create Impact" and "Promise Resolved" are printed to the console again.

Code

1. create a diffrent promise and I try to resolve it so now my code will have two promises with different delays , So now I have two promises suppose p1 and p2

```
const p1 = new Promise((resolve , reject)=>{
   setTimeout(()=>{
       resolve('Promise Resolved')
   } , 10000)
 })
  const p2 = new Promise((resolve , reject)=>{
   setTimeout(()=>{
       resolve('Promise Resolved')
   } , 5000)
 })
  async function handlePromise() {
   // JS engine waits for the promise to get resolved and
then moves forward
```

```
    console.log("Scaler")
    const val = await p1
    console.log('Create Impact 1')
   console.log(val)
    const val2 = await p2
    console.log('Create Impact 2')
   console.log(val2)
 }
  handlePromise()
```

Ques - How will this code work now? there is a promise which resolves in 5 sec (p1) and there is promise which resolves in 10 sec (p2) , So will the 5 sec promise excute first and it will be printed will the 10 sec promise execute first and then the 5 sec promise will start doing it's job , will the total time be 15 sec to execute the whole function or will it be 10 sec where both the promise will resolve in a 10 sec timeframe?

ANs

1. Start Execution: When the script starts executing, the top-level code is run. The definition of p1 and p2 are encountered, and they are both initialized as promises. Each promise is set to resolve after a certain timeout (p1 after 10 seconds, and p2 after 5 seconds). The setTimeout function calls are handed off to the Web API environment, which will handle the timer.
2.

3. Call handlePromise: Next, the handlePromise function is called. This call is placed on the call stack, and the function starts executing.
4.
5. Log "Scaler": The first operation inside handlePromise is a console.log("Scaler"), which is executed immediately, logging "Scaler" to the console. This operation is synchronous and executed straight away.
6.
7. Await p1: The execution then reaches the await p1 statement. At this point, the JavaScript engine checks the status of p1. Since p1 is not yet resolved (it's set to resolve after 10 seconds), the await pauses the execution of handlePromise, allowing other operations to run (if there were any). The function's execution is essentially paused, waiting for p1 to resolve. The function is removed from the call stack, and control is returned to the event loop.
8. p2 Resolves First: After 5 seconds, p2 resolves ("Promise Resolved"). However, because handlePromise is currently awaiting the resolution of p1 and not p2, this resolution does not immediately affect the paused state of handlePromise. The resolved promise (p2) is ready, but its handling awaits the execution flow to reach the await p2 statement
9. p1 Resolves: After 10 seconds from the start, p1 resolves. Since handlePromise was awaiting p1, the resolution of p1

allows the paused execution of handlePromise to resume. The resolved value of p1 is assigned to val, and the execution continues to the next lines, logging "Create Impact 1" and the resolved value of p1 to the console.

10.

11. Log "Create Impact 2": The last part of the function logs "Create Impact 2" and the resolved value of p2 to the console. This concludes the execution of handlePromise.

12.

13. Conclusion: The event loop, during this process, continually checks if the call stack is empty and if there are any pending callbacks (from resolved promises or other asynchronous operations) that need to be executed. In this case, it ensures that once p1 and p2 are resolved, their results are properly handled according to the async/await logic within handlePromise.

```
function test1(){
console.log("test")
}
const test2 = new Promise((resolve, reject) => {
   setTimeout(() => {
       resolve("test2")
   }, 3000)
})

function test3(){
   console.log("test3")
}

async function getData(){
   const data = await test2
   console.log(data)
}
test1()
getData()
test3()
```

To summarize, the call stack executes synchronous operations immediately and utilizes the event loop and Web API environment to handle asynchronous operations like timeouts and promises. The await keyword pauses the execution of the async function until the awaited Promise is resolved, allowing other tasks to run in the meantime.

coffee Problem

Now as you know the how async/await works , We will take an example , first we will solve it by using promises and then we will use async/await to solve the same problem title: Coffee Shop Problem

Question

Here we have a problem that we are at a coffee shop and the now the coffee shop only has coffee we cannot order for any other drink , it will reject if any other drink is ordered and if coffee is ordered it will be accepted then it will be processed then it will be served and at the end we will recieve a bill. The task is to promisify this process.

Create resolve and reject states of Promise to place a order, then process the order and then generate a bill process.

1. Step - 1: Create a Promise method for placing/accepting the order.
   a. Create resolve state if order is placed

```
function placeOrder(drink) {
   return new Promise(function(resolve, reject) {
       if(drink === 'coffee') {
           resolve('Order for Coffee Placed.')
       }
       else {
           reject('Order can not be Placed.')
       }
   })
}

placeOrder('coffee').then(function(orderStatus) {
   console.log(orderStatus)
})
```

2. Placing order for tea

```
placeOrder('tea').then(function(orderStatus) {
   console.log(orderStatus)
}).catch(function(error) {
   console.log(error)
})
```

3. Step - 2: Create a Promise method for process the order.

```
function placeOrder(drink) {
 return new Promise(function (resolve, reject) {
   if (drink === "coffee") {
     resolve("Order for Coffee Placed.");
   } else {
     reject("Order can not be Placed.");
   }
 });
}

function processOrder(orderPlaced) {
 return new Promise(function (resolve) {
   resolve(`${orderPlaced} and Served.`);
 });
}

placeOrder("coffee")
 .then(function (orderStatus) {
   console.log(orderStatus);
   return processOrder(orderStatus);
 })
 .then(function (finalStatus) {
   console.log(finalStatus);
 });
```

4. Create a promise method to generate the bill

```
function generateBill(processedOrder) {
 return new Promise(function (resolve) {
   resolve(`${processedOrder} and Bill Generated with 200 Rs.`);
 });
}
```

```
placeOrder("coffee")
 .then(function (orderStatus) {
   console.log(orderStatus);
   return processOrder(orderStatus);
 })
 .then(function (finalStatus) {
   console.log(finalStatus)
   return finalStatus
 })
 .then(function (orderIsProcessed) {
   const billGenerated = generateBill(orderIsProcessed);
   return billGenerated;
 })
 .then(function (bill) {
   console.log(bill);
 });
```

5. Lets say bill generation takes some time

```
function generateBill(processedOrder) {
 return new Promise(function (resolve) {
   setTimeout(function () {
     console.log("Bill Generated");
     resolve(`${processedOrder} and Bill Generated with 200
Rs.`);
   }, 3000);
 });
}
```

```
placeOrder("coffee")
 .then(function (orderStatus) {
   console.log(orderStatus);
   return processOrder(orderStatus);
 })
 .then(function (finalStatus) {
   console.log(finalStatus)
   return finalStatus
 })
 .then(function (orderIsProcessed) {
   const billGenerated = generateBill(orderIsProcessed);
   console.log("fill the feedback till bill is generating")
   return billGenerated;
 })
 .then(function (bill) {
   console.log("bill poaid:",bill);
 });
```

Observation: this chaining of operations is not very intuitive and not very clean way of coding

Optimized Solution: Using Async & Await

```
function placeOrder(drink) {
 return new Promise(function (resolve, reject) {
   if (drink === "coffee") {
     resolve("Order for Coffee Placed");
   } else {
     reject("Order cannot be Placed");
   }
 });
}

function processOrder(orderPlaced) {
 return new Promise(function (resolve) {
   resolve(`${orderPlaced} and Served`);
 });
}

function genreateBill(processedOrder) {
 return new Promise(function (resolve) {
   resolve(`${processedOrder} and Bill generated with 200Rs`);
 });
}

// Async and Await
// to use async await you need to create Functions

async function serveOrder() {
 let orderstatus = await placeOrder("coffee");
 console.log(orderstatus);
 let processedOrder = await processOrder(orderstatus);
 console.log(processedOrder);
 let generatedBill = await genreateBill(processedOrder);
 console.log(generatedBill);
}

serveOrder();
```

How to handle error with async await

To Handle error we will use try and cacth method

```
function placeOrder(drink) {
 return new Promise(function (resolve, reject) {
   if (drink === "coffee") {
     resolve("Order for Coffee Placed");
   } else {
     reject("Order cannot be Placed");
   }
 });
}

function processOrder(orderPlaced) {
 return new Promise(function (resolve) {
   resolve(`${orderPlaced} and Served`);
 });
}
```

```
function genreateBill(processedOrder) {
 return new Promise(function (resolve) {
   resolve(`${processedOrder} and Bill generated with 200Rs`);
 });
}

// Async and Await
// to use async await you need to create Functions

async function serveOrder() {
 try {
   let orderstatus = await placeOrder("tea");
   console.log(orderstatus);
   let processedOrder = await processOrder(orderstatus);
   console.log(processedOrder);
   let generatedBill = await genreateBill(processedOrder);
   console.log(generatedBill);
 } catch (error) {
   console.log(error);
 }
}

serveOrder();
```

At each await statment the promise related to it will be resolved and in a sequntial manner everything will be executed!

EXTRA: can rewrite the promise using async function

```
async function placeOrder(drink) {
```

```
//   return new Promise(function (resolve, reject) {
   if (drink === "coffee") {
     return ("Order for Coffee Placed");
   } else {
     throw new Error("Order cannot be Placed");
   }
//   });
}
```

This is how you can use async/await in JS