Class 13 - Async programming 2

Agenda

What are Promises

Asynchronous Programming with Promises

Chaining of Promises

Async Tasks in Concurrent Order

Async Task in Serial Order

Order of execution of asynchronous operations

1. In the fs.js example, we are interacting with two files. If i have to ensure that file 2 is written with the data from file1, how do I make that possible
2. We are moving the writing of file 2 inside callback of file1.

```
// Step 1: Read a file
fs.readFile('f1.txt', function(err, data) {
    if (err) {
        console.error("Failed to read file:", err);
        return;
    }

    // Step 2: Process the data
    const processedData = data.toUpperCase();

    // Step 3: Write the processed data to a new file
    fs.writeFile('f2.txt', processedData, function(err) {
        if (err) {
            console.error("Failed to write file:", err);
            return;
        }

        console.log("File processed and written successfully.");
```

```
      // More nested operations could go here, deepening the "pyramid"
  });
});
```

3. Now if there are 10 such async operations and we for some reason need to ensure their order then based on what we have seen, we will start placing the next operation inside the callback of the first.
   Example fetch a user data -> then get his / her posts -> then get his/ her comments

```
function fetchUser(userId) {
 fetchUserData(userId, (err, userData) => {
   if (err) handleError("fetching user data", err);
   else {
     log("User data fetched", userData);
     fetchUserPosts(userData.id, (err, posts) => {
       if (err) handleError("fetching user posts", err);
       else {
         log("Posts fetched", posts);
         posts.forEach((post) => {
           fetchCommentsByPost(post.id, (err, comments) =>
{
             if (err) handleError("fetching comments for
post", err, post.id);
             else log("Comments for post", post.id,
comments);
             // Potential for more nested callbacks here
           });
         });
```

```
        }
      });
    }
  });
}

// Auxiliary functions to streamline pseudo-code
function handleError(action, err, postId = "") {
  console.error(`Error ${action}:`, postId, err);
}

function log(message, data) {
  console.log(message, data);
}
```

4. This leads to a situation of nested callbacks and we call this as **callback hell / pyramid of doom**
5. Callback hell, often referred to as "Pyramid of Doom," is a pattern in JavaScript that arises **when multiple asynchronous operations need to be performed in sequence**. Each operation waits for the previous one to complete and then proceeds, leading to deeply nested callbacks. This can make the code difficult to read, maintain, and debug

title: What are Promises

1. In JavaScript, a promise is an object representing the eventual completion or failure of an asynchronous operation.

2. It provides a way to handle asynchronous code more cleanly and manage the results or errors that may occur when the operation completes.
3. Promises have three states: pending, resolved (fulfilled), or rejected, and they allow you to attach callback functions to handle these different outcomes.

To show to you how the representation might change

```
getData(function(a){
   getMoreData(a, function(b){
       getEvenMoreData(b, function(c){
           console.log(c); // Callback Hell
       });
   });
});

getData()
   .then(getMoreData)
   .then(getEvenMoreData)
   .then(console.log); // Cleaner and easier to read
```

title: Asynchronous Programming with Promises

Lets look at different states of Promise

1. Pending

Explanation: This is the initial state of a Promise after it has been created but before it has been resolved or rejected. At this stage, the Promise is waiting for an operation to complete or fail.

Analogy: Think of it as ordering something online. You've placed the order, but it hasn't shipped yet.
2. Fulfilled (Resolved)
Explanation: A Promise enters this state if the operation completes successfully and returns a value. In the context of promises, "fulfilled" and "resolved" are often used interchangeably,
Analogy: Your online order has been shipped and delivered to you. You now have what you were promised.
3. Rejected
Explanation: This state is reached if the operation fails and the Promise cannot be fulfilled, usually due to an error or some other problem. The Promise provides a reason for the rejection.
Analogy: There was a problem with your order (maybe the item was out of stock), and it can't be delivered to you. You are informed why it couldn't be fulfilled.

Promises in JavaScript are a powerful way to handle asynchronous operations. They are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value.

Creating a Promise

1. A promise is created using the Promise constructor, which takes a function called the "executor" as its argument. The executor function is executed immediately by the Promise

implementation, and it receives two functions as parameters: resolve and reject.

2. resolve(value) — If the operation is successful, this function is called with the result.
3. reject(error) — If the operation fails, this function is called with the error.
4. **in JavaScript, you create a "promise" to do something. This is done using something called a "Promise constructor." It's like telling JavaScript, "I promise to give you a result later."**
5. let promise = new Promise(function(resolve, reject) {

// executor (the producing code)

});

6. This function has two main jobs, represented by two parameters: resolve and reject.
7. resolve: You use resolve when the job is completed successfully.
8. reject: You use reject when something goes wrong.

A Simple Example

Let's create a simple promise that simulates a coin toss:

```
const coinTossPromise = new Promise(function (resolve, reject) {
 // executr code - async operation
 setTimeout(function () {
   const isHeads = Math.random() > 0.5;
   if (isHeads) {
     // success
```

```
      resolve("Heads");
    } else {
      // failure
      reject(
        "Tails- Coin toss resulted in tails, considered as a fail
for this example"
      );
    }
  }, 1000);
});
```

Consuming a Promise

1. To use a promise, you attach callbacks to it using the .then(), .catch(), and .finally() methods.

2. .then() takes two arguments:
    a. The first argument is a callback function that is called when the promise is fulfilled, meaning the operation completed successfully
    b. The second argument is an optional callback function that is called when the promise is rejected, meaning the operation failed.
    c. However, it's more common to handle failures using the .catch() method for improved readability and separation of concerns.

3. .catch() is used to handle any errors or rejections from the promise.
4. .finally() allows you to execute logic regardless of the promise's outcome.

```
coinTossPromise
.then(function(result){
   console.log("success msg:",result); // on success / resolve
})
.catch(function(err){
   console.log("Err msgL:",err); // on error / reject
})
.finally(function(){
   console.log("Promise settled"); // always executed
   return "I will always run";
})
```

Explanation:

1. imagine you're flipping a coin, and you're waiting to see if it lands on heads or tails. In the world of JS, we use something called a "promise" to handle tasks that take some time to complete, like waiting for the coin to land. A promise can end in two ways: it can either be successful (resolved) or unsuccessful (rejected).

2. In the code we see, coinTossPromise is like flipping the coin. After the coin is flipped, one of three things can happen:

    a. If the coin lands on heads (Promise is resolved):

b. The code inside .then runs. Here, it's like saying, "Yay, it's heads!" and then showing the result with console.log(result);.
c. If the flip fails for some reason (Promise is rejected):

d. The code inside .catch runs. This is like saying, "Oh no, something went wrong!" and then showing the error message with console.error(error);.
e. No matter what happens (resolved or rejected):

f. The code inside .finally runs at the end. It's like saying, "The coin toss is done, no matter the outcome." and then marking the completion with console.log("Coin toss completed.");.

title: Chaining of Promises

1. We saw one promise and how to handle its success or failure
2. But the real strength of promises is their ability to be chained. The .then() method returns a new promise, **which allows for sequential execution of asynchronous operations**.

Let's see how chaining works with using the some example

```
const cleanRoom = function () {
 return new Promise(function (resolve, reject) {
   resolve("I Cleaned The Room");
   // reject("I did not clean the room")
 });
};

const eatFood = function (message) {
 return new Promise(function (resolve, reject) {
   resolve(message + " I ate my food");
 });
};

let getCandy = function (message) {
 return new Promise(function (resolve, reject) {
   resolve(message + " now you get Candy");
 });
};

cleanRoom()
 .then(function (result) {
   return eatFood(result);
 })
 .then(function (result) {
   return getCandy(result);
 })
 .then(function (result) {
   console.log("finished " + result); // Logs the final message after
all promises are resolved
 })
```

Code explanation:

1. The code we see demonstrates the concept of chaining asynchronous operations in JavaScript using Promises. It follows a logical flow for executing tasks one after another, where each task starts only after the previous one has finished.
2. Defining the tasks
    a. cleanRoom function: Represents the task of cleaning the room. Once the room is cleaned, it resolves the promise with a message saying "I Cleaned The Room".
    b. eatFood function: Represents eating food. It takes a message as input and appends " I ate my food" to it, then resolves the promise with this new message.
    c. getCandy function: Represents getting candy as a reward. It also takes a message as input, appends " now you get Candy" to it, and then resolves the promise with this new message.
3. Chain the tasks
    a. We start by calling cleanRoom(). Since cleaning the room is the first task, there's no input message; it simply resolves with "I Cleaned The Room".
    b. Once the room is cleaned (cleanRoom resolves), we move to the next task by using .then(). The result from cleanRoom is passed to eatFood.
    c. eatFood(result) takes the message from cleaning the room, adds " I ate my food" to it, and resolves. We chain another .then() to handle the resolution of eatFood.

d. The result from eatFood is then passed to getCandy, which adds " now you get Candy" to the message and resolves.
e. Finally, we have one last .then() that takes the result from getCandy and logs "finished " followed by the entire message.
f.

In this way you can mainatain your Promises in Sequence!

Error Handling in Chains

1. When chaining promises, a rejection in any promise will skip all the subsequent .then() methods until it finds a .catch(). This behavior simplifies error handling in a chain of asynchronous operations.
2. To handle rejection more explicitly in our candy example, let's adjust the functions to include scenarios where they might fail. We'll also incorporate explicit rejection handling to demonstrate how you can manage both success and failure outcomes in a promise chain.
3. Suppose now there is a 50% chance that function will reject

```
const cleanRoom = function () {
 return new Promise(function (resolve, reject) {
   if (Math.random() < 0.5) {
     resolve("Cleaned The Room");
   } else {
     // 50% chance of failure
```

```
    reject("not in mood to clean the room");
  }
  // reject("I did not clean the room")
 });
};

const eatFood = function (message) {
 return new Promise(function (resolve, reject) {
  if (Math.random() < 0.5) {
    resolve(message + " ate my food");
  } else {
    // 50% chance of failure
    reject("Dont like this food");
  }
 });
};

let getCandy = function (message) {
 return new Promise(function (resolve, reject) {
  resolve(message + " now you get Candy");
 });
};

cleanRoom()
 .then(function (result) {
  console.log(result);
  return eatFood(result);
 })
 .then(function (result) {
  console.log(result);
  return getCandy(result);
 })
```

```
 .then(function (result) {
   console.log(result);
   console.log("finished " + result); // Logs the final message
after all promises are resolved
 })
 .catch(function (error) {
   console.log("Error: " + error + "No candies for you"); //
Logs an error message if any of the promises are rejected
 });
```

In this setup, if any of the promises are rejected, the execution jumps directly to the .catch() method, skipping any remaining .then() methods. This is useful for centralizing error handling logic.

Conclusion

By introducing the possibility of failure and handling these failures explicitly, we can build robust asynchronous workflows that can gracefully handle both success and failure scenarios. This flexibility is one of the key strengths of using promises in JavaScript, allowing developers to write more reliable and maintainable code when dealing with asynchronous operations.

title: Read File Using Promises (concurrent)

1. How to read a file using Promises?
2. First of we will see how to read file using callback only , this we have seen in our previous class

```
const fs = require("fs");

fs.readFile("f1.txt", cb);

function cb(err, data) {
 if (err) {
   console.log(err);
 } else {
   console.log("This is File 1 data -> " + data);
 }
}
```

3. The fs.promises API is an object that provides asynchronous file system methods that return promises. This allows us to use modern asynchronous programming patterns with the file system operations, such as async/await or chaining operations with .then() and .catch().

4. check the output for below

```
const fs = require('fs')
const allPromises = fs.promises
console.log(allPromises)
```

5. Now update to read using then and catch

```
const fs = require("fs");

let promiseReadFile = fs.promises.readFile("f1.txt");
```

```
promiseReadFile.then(function (data) {
 console.log("This is file data -> " + data);
});

promiseReadFile.catch(function (err) {
 console.log("This is Your Error -> " + err);
});
```

title: Read multiple File Using Promises

```
const fs = require('fs')

let promiseReadFile1 = fs.promises.readFile('f1.txt')
let promiseReadFile2 = fs.promises.readFile('f2.txt')
let promiseReadFile3 = fs.promises.readFile('f3.txt')

// For File 1
promiseReadFile1.then(function(data) {
   console.log('This is file 1 data -> ' + data)
}).catch(function(err) {
   console.log('This is Your Error -> ' + err)
})

// For File 2
promiseReadFile2.then(function(data) {
   console.log('This is file 2 data -> ' + data)
}).catch(function(err) {
   console.log('This is Your Error -> ' + err)
})

// For File 3
promiseReadFile3.then(function(data) {
```

```
   console.log('This is file 3 data -> ' + data)
}).catch(function(err) {
   console.log('This is Your Error -> ' + err)
})
```

Since we are using promises , so promises also follow the event loop Mechanism , So you will see random order here Files will be read in a concurrent fashion

Cleaning the code

```
const fs = require("fs");

let f1p = fs.promises.readFile("f1.txt");
let f2p = fs.promises.readFile("f2.txt");
let f3p = fs.promises.readFile("f3.txt");

function readFileCallback(data) {
 console.log("This is the data -> " + data);
}

function handleError(err) {
 console.log("This is my error -> " + err);
}

f1p.then(readFileCallback);
f2p.then(readFileCallback);
f3p.then(readFileCallback);
```

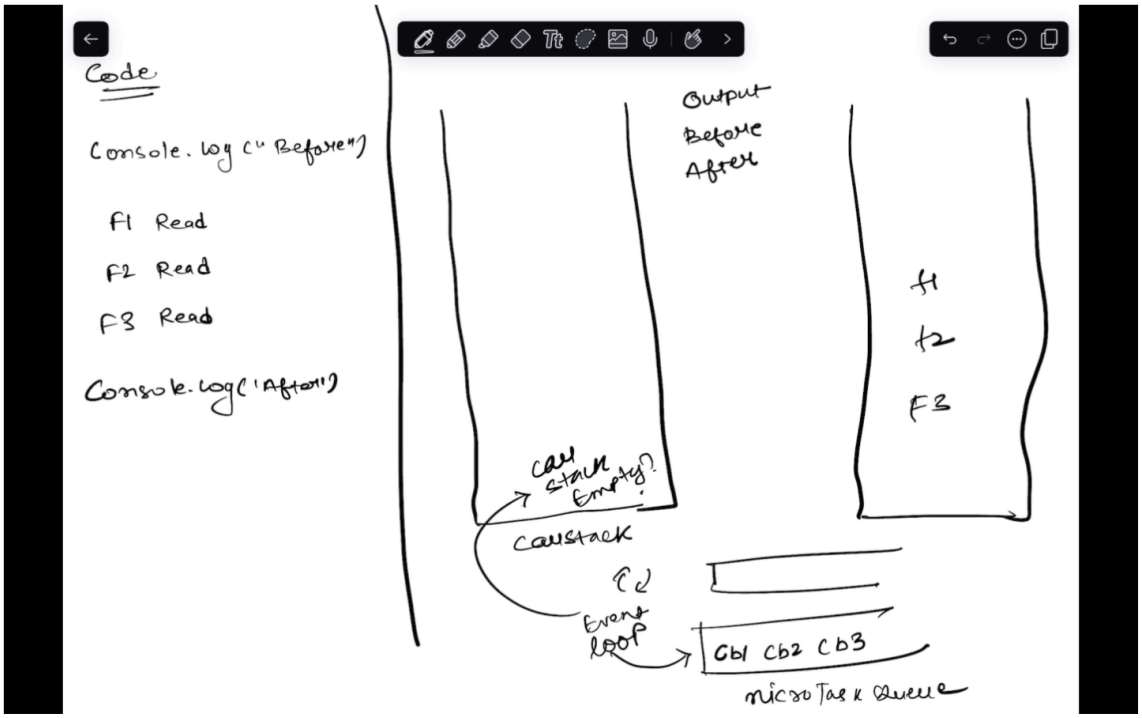
```
f1p.catch(handleError);
f2p.catch(handleError);
f3p.catch(handleError);
```

Event Loop

1. The event loop is a mechanism that allows Node.js / JS engines to perform non-blocking I/O operations, despite JavaScript being single-threaded. Here's how the event loop handles the promises:
2. The event loop continually checks if there's any work to be done and decides what to execute next.
3. It prioritizes executing the call stack (synchronous code) first and then looks to various task queues to see if there are any tasks to process.
4. These task queues include:
    a. Macro Task Queue: Contains tasks like setTimeout, setInterval, and I/O callbacks.
    b. Micro Task Queue: Contains tasks like promise callbacks (then, catch, finally handlers).
    c. When a promise is resolved or rejected, any associated .then(), .catch(), or .finally() callbacks are not executed immediately. Instead, they are placed into the micro task queue.
    d. The event loop will only execute these callbacks after the call stack is empty, ensuring that all synchronous code has finished executing.

e. **Here's the crucial part: the event loop gives higher priority to tasks in the micro task queue over tasks in the macro task queue.**
f. This means promise callbacks are executed before timers and I/O callbacks if both are queued.

Javascript visualizer

1. 

Microtasks Queue

1. Promise Callbacks Execution: Promise callbacks unlike regular callbacks (then and catch) are considered microtasks in Node.js

and are executed immediately after the current phase of the event loop completes and before moving to the next phase.

2. This means that as soon as the promise settles (either fulfilled or rejected), the corresponding .then() or .catch() callbacks are placed in the microtasks queue and executed in the order they were scheduled.
3. The callbacks for reading f1.txt, f2.txt, and f3.txt are executed as soon as their respective read operations complete and the event loop gets to processing microtasks.
4. The order of logs (This is file 1 data ->, This is file 2 data ->, This is file 3 data ->) to the console depends on the order in which the files are read successfully. It's not guaranteed to be in the order f1.txt, f2.txt, f3.txt as you already know

Conclusion

1. The event loop facilitates the non-blocking I/O operations by allowing the file reads to happen in the background. Once these operations are complete, the event loop ensures the registered callbacks are executed at the appropriate time.
2. The promises and the microtask queue allow handling asynchronous operations' outcomes in a clear and manageable way, executing promise callbacks as soon as possible after their asynchronous operations complete.

To show the compariosn between microtask and callback queue (task queue) You can see it this tool
https://www.jsv9000.app/

that which queue takes priority and how promise callbacks are given the firdt priority

Example code:

```
function logA() { console.log('A') }
function logB() { console.log('B') }
function logC() { console.log('C') }
function logD() { console.log('D') }


logA();
setTimeout(logB, 0);
Promise.resolve().then(logC);
logD();
```

title: How Serial Operation Works

Promise Chaining:

1. Suppose Now , we Want all this files to be read in a serial order , You already know Promise chaining that how you can maintain a squence of Promises.
2. It involves linking multiple asynchronous operations together, ensuring that one operation starts only after the previous one has completed successfully.
3. This is typically achieved using the .then() method to handle the result of a Promise and return another Promise, allowing you to chain multiple operations together in a clean and sequential manner.

```
const fs = require("fs");

console.log("Before");

let f1p = fs.promises.readFile("f1.txt");

f1p
 .then(function (data) {
   console.log("This is File 1 Data -> " + data);
   return fs.promises.readFile("f2.txt");
 })
 .then(function (data) {
   console.log("This is File 2 Data -> " + data);
   return fs.promises.readFile("f3.txt");
 })
 .then(function (data) {
   console.log("This is File 3 Data -> " + data);
 });

console.log("After");
```

This code demonstrates asynchronous file reading in Node.js, using Promises to handle the data from each file in sequence.

It shows how to chain asynchronous operations in a clean and readable manner, with each step waiting for the previous one to

complete before proceeding, without blocking the execution of subsequent code like the "After" log message.

this is how you can achieve Serial Operations with Promises