9. Asynchronous JavaScript - callbacks, promise and async-await

9.1 Introduction:

- **Asynchronous vs synchronous code (revision)**:
  - **Synchronous code**: In synchronous code, every statement of the code gets executed one by one. Meaning, until the current code finishes executing, the following code will be stopped from being executed because the thread, where these actions are taking place, is blocked. This may lead to the delaying of the UI.
  - **Asynchronous code**: The word 'asynchronous' means 'takes some time' or 'happens in the future, not right now' An asynchronous model allows multiple things to happen at the same time. In asynchronous code, you can move to another task before the previous one finishes. Meaning, asynchronous code lets you execute a block of code without stopping (or blocking) the following code. This way, asynchronous programming increases the number of tasks that can be executed concurrently without having to block the thread, thus completing more tasks in a much shorter period of time
- **JavaScript is synchronous by default:** By default, every line in a function executes sequentially, one line at a time. The same is true when you invoke multiple functions in your code. This is because JavaScript is a single-threaded programming language where its engine can only process one task at a time using its single thread. This nature of JavaScript can sometimes cause unnecessary delay and hence slowness of the application
- **When do we want JavaScript code to execute asynchronously?** Executing things in sequence works perfectly fine in JavaScript. However, there are times when you want to fetch data from the server or execute a function with a delay. In such instances, you want the JavaScript engine to stop the sequential execution of code and want it to execute the code asynchronously
- **JavaScript offers three different methods of asynchronous code implementation**:
  - 1. Callback function with setTimeout and setInterval methods (powered by browser API)
  - 2. Async-await
  - 3. Promise

9.2 Callback functions

- **What is a callback?** A callback is a regular JavaScript function passed as a parameter to another JavaScript function. Such a function is called callback because it is executed after another function has finished executing, hence the name callback.
    - **Synchronous callback**: If we have a callback function "C" that is passed as a parameter into another function, called "X", function "X" executes the code under function "C" at some point. If the execution of function "C" is immediate, then the execution is synchronous.
    - **Asynchronous callback**: However, if the execution of the above function "C" happens later (example, 10 seconds later), it is an asynchronous callback.
    - **Example of synchronous callback**: Let us say that we want a function that takes two numbers, adds them and returns the result. We also want to display the result somewhere. In such a case, we create an adder function first and save the sum result (adderFunction). We will also call another function to display the result (displayerFunction). Let us look at it below

      // index.html

```
<body>
    <div id="display"> </div>
    <script src="script.js"></script>
</body>
```

// script.js

```
function displayerFunction(someParameter) {
    document.getElementById("display").innerHTML=someParameter;
}
function adderFunction(num1, num2) {
    let sum = num1 + num2;
    displayerFunction(sum);
}

adderFunction(5, 10); // 15 shows on browser
```

- **Why do we need callback functions**: Like stated above, we will not see the use of callback to be huge in synchronous callback. However, callbacks become very important when we want to have better control over when to execute a function. Callbacks really become important in asynchronous functions, where one function must wait for another function (like waiting for a file to be fetched from a database before displaying it on the browser).
- **Callback functions with setTimeout/setInterval methods to allow JavaScript engine to run asynchronous code**: One question to ask here is "if JavaScript is single threaded and can only handle one task at a time, how is it possible to asynchronous code without blocking that single thread using JavaScript?". Even if JavaScript can only do one thing at a time by its nature, the browser can execute concurrent tasks asynchronously through its APIs. Meaning, web browsers have built-in APIs (example: methods like "setTimeout", event handlers like "click") that we can call in our JavaScript code that allow asynchronous code to execute. Because the execution of such asynchronous code is handled by the browser itself, the single thread will not be blocked, or it will not block the call stack

- **API (Application Programming Interface)**: API is a tool that lets two independent applications exchange data. API is part of the server that serves as a middleman by receiving data request from one application and sending a response data to another application. **Note**: API is nothing but a set of functions/methods and properties that developers use to access and integrate the functionality of another application into their application. **Example**: If you use Google maps in your website or embedding YouTube videos into your website
- **Browser/web API**: Web APIs are the APIs made available to front-end developers, by the browser (these APIs are built-in the browser). Their main task is to connect the browser (an application itself) to other applications so that we can add more functionality to the browser. **For example**: DOM API connects web pages to browsers using JavaScript to access/manipulate your website's HTML and CSS. Visit this MDN's website here to see more examples of web APIs

- **Diagram showing how browsers and JS engine work together to handle synchronous and asynchronous code**: Look at this diagram that shows how JavaScript engine handles synchronous code. The diagram also shows how JavaScript engine and the browser interact to execute asynchronous code without blocking the normal thread of synchronous execution. Let us explain the terms in the diagram before explaining browser and JavaScript handling of asynchronous code:
  - **Memory/JavaScript heap**: The JavaScript heap is where objects are stored when we define functions or variables. Since memory heap does not affect the call stack and the event loop, we will not be discussing it under this section
  - **Call stack**: This is the synchronous part of JavaScript engine component that tracks the current code being executed and lets JavaScript engine know what code to execute next. **Note**: Everything that happens inside the call stack is sequential as this is the synchronous part of JavaScript. JavaScript's main thread makes sure that it takes care of everything in the stack before it starts looking into anything in the callback queue

- **Callback queue**: This is the asynchronous part of a JavaScript engine that maintains a queue of callback functions (asynchronous code). Once the JavaScript engine finishes executing the synchronous tasks on the call stack, it continues to execute the asynchronous code found on the callback queue. **Note**: Unlike the call stack, the callback queue follows the FIFO order (First In, First Out), meaning the callback function that first gets into the queue has the opportunity to go out first. Questions to ask:
    - **When does the JavaScript engine put asynchronous code in the call back queue?** The JavaScript engine keeps executing synchronous code in the call stack. No callback function is pushed into the call stack, therefore, there is no code blocking the execution of synchronous code in the call stack. Calling setTimeout(), setInterval() or other browser API methods, triggers a browser API. Then the browser API adds the callback in the setTimeout() to the callback queue. Because the callback is not in the call stack, the synchronous code in the call stack continues to execute
    - **When does the JavaScript engine take it out of the queue?** JavaScript engine creates a loop, called the event loop, that looks into the call back queue periodically to see if there is any callback function to pull from the callback queue. If there is a callback function in the callback queue and if the call stack is empty, the event loop takes this callback out of the callback queue and pushes it to the stack. Now the callback function executes generally as any other function in the stack. If there are more callbacks in the queue, the loop continues to pull them from the queue and pushes them to the stack for execution

- **The "Even Loop"**: it is a system to track tasks being executed at the external processor. Once the browser finishes processing the tasks under the synchronous stack (or call stack), event loop will pass the tasks under call back queue to the browser to be processed. (**Example**: While the events at the call stack are synchronously being processed, tasks like bringing data from the server, will be taken to the call back queue where they will be processed on the side. Once browser finishes the code at the call stack (the synchronous code), the event loop pops the asynchronous code from the callback queue and pushes them onto the call stack to be executed. **Note**: One simple thing you can notice in here is that all the asynchronous code gets executed after the synchronous ones are executed

- **What happens when we create JavaScript functions and execute them?**
  - 1. The JavaScript engine maintains a stack data structure called call stack which tracks the current function in execution
  - 2. When engine executes function, it adds it to the stack, and execution starts
  - 3. If the currently executed function calls another function, the engine adds the other function to the stack and starts executing it
  - 4. Then the engine will resume the execution of the first function from the point it left it last time
  - 5. Once execution of the first function is over, the engine takes it out of the stack
  - 6. Continue the same way until there is nothing to put into the stack

- **Explaining the above 6 steps with examples**:
  - **Example1: Synchronous code**

```
// script.js

function f1() {
        console.log("Hi");
}

function f2() {
        console.log("Hi again");
}

function f3() {
        console.log("Bye");
}

f1();

f2();

f3();
```

    - In the above code snippet, first, f1() goes into the call stack, executes, and pops out. Then f2() does the same, and finally f3(). After that, the stack is empty, with nothing else to execute. So, the codes work in a sequence. Every line of code waits for its previous one to get executed first and then it gets executed
    - **Note**: JavaScript functions are executed in the sequence they are called, not in the sequence they are defined

- **Example 2: Synchronous code (functions that call other functions)**:

// Script.js

```
function f1() {
        console.log("Hi");
}
function f2() {
        f1();
}
function f3() {
        f2();
}
f3();
```

- In the above code snippet, f3() gets into the stack first. Because f3() calls f2() in it, f2() also gets added to the stack while f3() remains in the stack. Because f2() itself invokes f1(), f1() goes to the call stack with both f2() and f3() in the stack. Because f1() does not invoke any function in it, it gets executed first and comes out of the stack first. Right after that f2() finishes, and finally f3()

- **Example 3**: Asynchronous code

```
function printMe() {
console.log("print me");
}

function test() {
console.log("test");
}

setTimeout(printMe, 2000);
test();
```

- In the above code, JavaScript engine WILL NOT wait for 2000ms and print "print me" first and then go to print "test". Rather, it will manage to keep the callback function of setTimeout aside and continue its other executions. That's where the asynchronous mechanism kicks in. Therefore, “test” prints first, then, after 200ms, “printMe” prints in the console

- **Example 4:** Asynchronous code

```
// Script.js
function f1() {
        console.log("f1");
}
 function f2() {
        console.log("f2");
}
function main() {
        console.log("main");

        setTimeout(f1, 1000);

        f2();
}

main();
```

- In the above code snippet, "main" prints first, then callback in setTimeout gets set aside to the callback queue. This is because calling setTimeout triggers the execution of the browserAPI, which adds the callback function to the callback queue. Secondly, "f2" prints. Once the call stack is empty, the callback gets removed from the callback queue and gets added in the call stack. After 1000ms, "f1" prints

- **Callback hell and other issues when using asynchronous callbacks**: If your app is not full of complex logic, a few callbacks seem harmless. But once your project requirements start to swell, you will quickly find yourself piling layers of nested callbacks. This is what we call callback hell. Refer this diagram to see what a callback hell looks like. From this diagram, you can see that each callback takes parameters that have been obtained as a result of previous callbacks
    - We saw earlier that can use the setTimeOut function to decouple a statement from the main synchronous code and run it asynchronously. The setTimeOut function allows us to run a function once after a given interval of time. The setTimeOut function mainly takes two arguments. Since the setTimeOut function runs asynchronously, we can pass the function that we want to run asynchronously as a callback function to it. This would make our code asynchronous. This is the traditional way of writing asynchronous code
    - **Example**: Let us assume you have three paragraphs and a button. When the button is clicked, assume you want to wait for 5000ms and change the first paragraph to red. Once you see the red paragraph, you want to wait for 500ms and change the second paragraph to blue. Once you see the blue paragraph, you want to wait for 2000ms and change the third paragraph to green

```
// index.html

<p id="first">Hello: Red</p>

<p id="second">Hello: Blue</p>

<p id="third">Hello: Green</p>

<button>Click</button>
```

// script.js

```
var myFirst = document.getElementById("first");
var mySecond = document.getElementById("second");
var mythird = document.getElementById("third");
var btn = document.querySelector("button");

btn.addEventListener("click", function () {
    setTimeout(function () {
        myFirst.style.color = "red";
    }, 5000);
    setTimeout(function () {
        mySecond.style.color = "Blue";
    }, 500);
    setTimeout(function () {
        mythird.style.color = "Green";
    }, 2000);
});
```

- **Issue with the above example code**: The above snippet of code basically will change the 2nd paragraph to blue first because the time we set for it to change its color is 500ms, which is less than the first paragraph that takes 5000ms. It is the 3rd paragraph that will change to green and lastly the first paragraph changes to red. This was not the sequence we wanted to run. We wanted the first one to turn to red (after 5000ms of the click), then want to wait for 500ms and change the 2nd paragraph to turn to blue and want to wait for 2000ms and turn the 3rd paragraph into green

- **How to fix the code**: One way to fix is hard coding the time for each setTimeout function to keep the sequence (like chaning the time for 2nd paragraph to 6000ms and the 3rd paragraph to 8000ms). However, such a hard coding solution is not always feasible. Let us see how to fix this without hard coding
- **Callback hell/nesting a callback in a callback**: To fix the above issue and guarantee that the first paragraph gets changed before the second, we can nest the second setTimeout function within the first one and nest the third one in the second one. That is what we call a callback hell. Notice: We did not need to change the time set for each paragraph to change color, we just need to call each setTimeout function within another

```
btn.addEventListener("click", function () {
    setTimeout(function () {
        first.style.color = "red";
    setTimeout(function () {
        second.style.color = "Blue";
    setTimeout(function () {
        third.style.color = "Green";
            }, 2000);
        }, 500);
    }, 5000);
});
```

- **Problems of nesting callback in a callback (callback hell)**: Look at the above code, because we have to call callbacks inside callbacks, we get a deeply nested function. This is much harder to read, debug and handle errors. For these reasons, most modern asynchronous APIs don't use callbacks. A better/newer way of creating asynchronous code is using Promises

9.3 Promise (creating a promise and states of a promise)

- **Why do we need JavaScript Promise?** Prior to JavaScript promises, callback functions were used to handle concurrent tasks without blocking in JavaScript. However, callbacks have limited functionalities and multiple callbacks create unmanageable code (callback hell). Promises are the ideal choice for handling asynchronous operations in the simplest manner. They can handle multiple asynchronous operations easily and provide better error handling than callbacks and events. In short, Promise lets you write an asynchronously executed but a predictable code
- **What is a Promise in JavaScript?** A Promise is a JavaScript object that serves as a representation/placeholder for an asynchronous task which is yet to be completed. Meaning, a promise produces a value after an asynchronous operation completes successfully, or after an error (if it does not complete successfully due to time out, network error, and so on)
- **Real life promise example**: Promises work in a similar way a real life promise works. Imagine that you are interviewing Abebe, a job seeker for a position at your company. When his interview session is about to begin, Abebe realizes that he has forgotten his résumé. He asks you to text his wife, Tirunesh, to help him find his résumé. Whether she finds Abebe's resume or not, Tirunesh PROMISES to text him back to let him know about the status of his résumé in their house. Assume Tirunesh would text him "successful" if she found the resume or "Unsuccessful" if she did not. While waiting for Tirunesh's text, the interviewer continues to interview Abebe holding on to the PROMISE of getting the resume, not holding on for the actual the resume. At this time, you set the status of the résumé delivery to PENDING. Abebe correctly answers all the questions asked, but his employment still depends on the FINAL STATUS of his résumé. Finally, his wife finally texts back. If her text indicates that she could not find the résumé, Abebe will share this failure with you. At this moment, the interview will end and Abebe will be REJECTED from the hope of getting the job. On the other hand, if Tirunesh finds the résumé, Abebe will happily tell you the succuss. At this moment, Abebe will go ahead and FULFILL his hopes of getting the job. The JavaScript Promises also work in a similar way

- **Creating a Promise in JavaScript**: The first thing we need to do to use Promises in JavaScript is to actually create the Promise object. Like it is mentioned in the definition above, Promises are just JavaScript objects, thus, we create them in a similar way we create any other object using Promise constructor and the "new" keyword
    - **Syntax:**

```
let myPromise = new Promise (function (resolve, reject) {

    // You write your executor code in here

});
```

- **Executor function (producing function)**: As you can see it above, executor function is the function that is passed to a new Promise as a parameter. This function runs automatically when the promise is constructed, and to the created Promise uses this function to perform a job (mostly asynchronously). When the function finishes attempting to perform the job, the executor function produces a success result (if "resolve" function is called) or a failure result (if "reject" function is called). **Note**: It is the executor function that moves Promise from "pending "to either "Fulfilled" or "Rejected" state
    - **Producing code**: Executor function contains the producing code which eventually produces the result
    - **Consuming code**: This is the code that uses the result of the "producing code" as an input
    - **The resolve and reject functions**: The executor function takes two takes two callback functions as parameters, the resolve and reject functions. **Note**: These functions are pre-defined by the JavaScript engine, so we do not need to create them. We should only call one of them when ready
        - **resolve**: If everything goes well, use the resolve function to fulfil the promise
        - **reject**: If desired operations do not go well then you will call "reject"

- **Example of executor responding to a successful promise**

```
let myPromise = new Promise(function (resolve, reject) {
        resolve("Succuss");
        });
});
```

- **Example of the executor rejecting the promise with an error**

```
let myPromise = new Promise(function (resolve, reject) {
        reject("error"), 1000;
});
```

- **Promise.resolve() vs. new Promise()**: Promise.resolve() is a built-in function that returns the Promise object resolved with the given value. Promise.resolve(x); is basically the same as new Promise(function(resolve){ resolve(x); });

```
function myDisplay() {
return Promise.resolve(300);
// The above returend promise is the same as the below
// return new Promise(function (resolve, reject) {
//      resolve(300);
// });
}
 myDisplay().then(function (value) {
console.log(value);// returns 300
});
```

- **Properties created by the Promise object**: When you create a new Promise following the above syntax, the Promise constructor returns an object with some unique properties. If you console.log this new object and try to observe what it looks like, you can see that there are two properties named "PromiseState" and "PromiseResult"
    - **PromiseState property**: This property holds the current status of the Promise and can have the following values:
        - **Pending**: The promise object has a "pending" value when the executor function initially starts the execution
        - **Fulfilled or Rejected**: It then then changes to either "fulfilled" when promise is resolved (when resolve is called) or it can change to "rejected" when the Promise is rejected (when reject function is called)
        - **Settled promise**: A promise that is either fulfilled or rejected is called a settled promise
        - **Note**: A Promise executor should call only one resolve or one reject. Any state change is final. All further calls of resolve and reject are ignored
    - **PromiseResult property**: It holds the result when Promise is fulfilled and can have the following values:
        - **Undefined**: When the executor function initially starts execution and state of promise is pending, result will have an "undefined" value
        - **value**: It then changes to "value" when resolve(value) function is called
        - **error**:It will change to "error" if the reject(error) function is called
    - **1. Example showing initial state of a Promise (when it is created)**

```
let myPromise = new Promise(function (resolve, reject) {

});

console.log(myPromise);
```

        - Your console will show you that the PromiseState as "pending" and the PromiseResult as "underfined"

- **2. Example showing fulfilled state of a Promise**

```
let myPromise = new Promise(function (resolve, reject) {\
        resolve("YesSuccess");
});
console.log(myPromise);
```

  - Your console will show you that the PromiseState as “fulfilled” and the PromiseResult value as “YesSuccess”

- **3. Rejected State of a Promise**

```
let myPromise = new Promise(function (resolve, reject) {\
        reject(“NoSuccess”);
});
console.log(myPromise);
```

  - Your console will show you that the PromiseState as “Rejected” and the PromiseResult value as “NoSuccess”

- **4. Promise with both reject and resolve functions (the latter one will be ignored)**

```
let myPromise = new Promise(function (resolve, reject) {
resolve("YesSuccess");
        reject("NoSuccess");
});
console.log(myPromise);
```

  - Your console will show you that the PromiseState as “fulfilled” and ignore the reject

9.4 Promise (consuming/handling a promise)

- **Consuming/handling a Promise**: The main reason we use Promises is to get notified when asynchronous code in the executor function completes a task and is ready to pass the returned result to the consumer function.
    - **Consumer function**: is a function that uses an outcome of the promise.
- **Applying dot notation on then() and catch() methods to consume the outcome of a Promise:** Once Promises are created, they can be consumed by applying the dot notation on the "then" and "catch" methods provided under the Promise object. **Note**: The .then() and .catch() methods create the link between the executor and the consumer functions so that they can be in sync when a promise resolves or rejects
- **Using .then() to consume the outcome of a Promise**: The then() function is the main method that we use to receive successful operations or failed operations of the Promise to use it in our consumer function. The .then() method should be called on the promise object to handle a result (resolve) or an error (reject).
    - **The .then() function takes 2 OPTIONAL callback parameters**: The .then() functions' two parameters, let us call them, the onFulfilled() and onRejected() functions will receive the result and the error respectively from the executor/producer function
        - **onFulfilled()**: We have seen that a promise always starts out in the pending state. If the promise transitions to the fulfilled state, JavaScript calls the onFulfilled() function
        - **onRejected()**: JavaScript will call this function if the underlying async operation failed
    - **Syntax**:

```
myPromise.then(
        function(result) { console.log("handle a successful result") },
        function(error) { console.log("handle an error")}
);
```

- **Handling ONLY successful outcomes with .then()**: If you are interested only in successful outcomes, you can pass only one argument like below

```
let myPromise = new Promise(function (resolve, reject) {
        resolve("YesSuccess");
});
myPromise.then(
        function(result) {
                console.log("successful result")
        }
);
```

- **Handling ONLY error outcomes with .then()**: If you are interested only in error outcomes, you can pass null for the first parameter

```
let myPromise = new Promise(function (resolve, reject) {
reject("NoSuccess");
        });
myPromise.then(null,
        function(result) { console.log("handle error") }
 );
```

  - **Note**: Errors can be handled better usign the .catch() function. The .catch() function is just a shorthand of .then(null, function), we will see that in the next section

- **Examples for .then() handling Promises**: Below is a Promise, called myPromise. If the Promise is resolved and the resolved result is received by the the .then() method, the first parameter of .then() method runs. Meaning, if the setTimeout finishes counting 2000ms and the .then() method receives that outcome, the first parameter of the .then() function runs and prints the promise's result, i.e., it prints "Successful!!"
  - **Example 1**:

```
let myPromise = new Promise(function (resolve, reject) {
        setTimeout(function () {
                resolve(" Unsuccessful!!");
        }, 2000);
});
myPromise.then(
        function (result) {
                console.log(result);
        },
        function (error) {
                console.log(error);
        }
);
```

- **Example 2**: Below if the Promise is rejected and the rejection is received by the the .then() method, the 2nd parameter of .then() method runs. Meaning, if the setTimeout finishes counting 2000ms and the .then() method receives rejection outcome, the 2nd parameter of the .then() function runs and prints “Successful!!”

```
let myPromise = new Promise(function (resolve, reject) {
        setTimeout(function () {
                reject("Unsuccessful!!");
        }, 2000);
});
 myPromise.then(
        function (result) {
                console.log(result);
        },
        function (error) {
                console.log(error);
        }
);
```

- **Promisifying/ Promisification**: Promisification means transformation. It is a conversion of a function (that accepts a callback) into a function that returns a promise. It is a technique whereby we use a classic JavaScript function that takes a callback and returns a promise. Assume you want to create a function that returns a promise to let you know if the file exists. If the promise's result is that the file exists, then you will read and print from the file. Assume you want to make sure the file exists before reading and printing its text. If the file does not exist, then you will print "Unsuccessful!!". Assume only "abebe.txt" exists, thus, the text from "abebe.txt" will be read. However, if you provide "alem.txt" when there is no file called as such, then the error message "unsuccessful" will print in the console

```
const fs = require("fs");
const getFileFunc = (fileName) => {
    return new Promise(function (resolve, reject) {
        fs.readFile(fileName, "utf8", function (err, textOnFile) {
            if (err) {
                reject("unsuccessful" );
                return;
            } else {
                resolve(textOnFile);
            }
        });
    });
};
```

```
getFileFunc("./abebe.txt").then(
        function (textOnFile) {
                console.log(textOnFile);
        },
        function (err) {
                console.error(err);
        }
);
```

- **Chaining promises**: When you want one asynchronous task to be performed after the completion of another asynchronous code, where the 2nd asynchronous operation starts with the results from the previous asynchronous operation, you will use promise chaining. Technically speaking, Promise chaining occurs when the callback function in the .then() method returns a promise. In reality, a promise chaining can be adding up a couple of numbers, waiting for two seconds, and fulfilling the promise with the sum.
    - **Syntax**:

```
myPromise
        .then(function(resultFromPromise){})
        .then(function(resultFromPromise){})
```

- **Steps to creating promise chaining**: Let us assume that we want to create a Promise that waits for 3 seconds and resolves “10” or gives 10 as a result. If the promise resolves and gives us the result 10, then we want to use that result, multiply it by 3 and return the multiplied value. Using the multiplied result value, we then want to return a new value divided by 2. In short, we want to take result “10” from our original Promise, then multiply that value by 3 and then divide that value by 2. This should give us 15 at the end
  - **1. Create a new Promise**: **Note:** The setTimeout() function below is to simulate an asynchronous operation.

```
let myPromise = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve(10);
    }, 3000);
});
```

  - 2. **Then, invoke the then() method of the Promise you just created**: Please note that the callback passed to the then() method below executes once the Promise is resolved. In the callback, we use the result of the Promise and return a new value (resultFromPromise * 3)

```
myPromise.then(function (resultFromPromise) {
    return resultFromPromise * 3;
})
```

  - 3. **To the second .then() method, pass the returned value from the first then() method**: Please note that, the first then() method returns a new Promise with a resolved value. In the callback for the second .then() method, use the result of the previous .then() method and return a new value (resultFromPromise / 2). Here is how chain your Promise:

```
myPromise.then(function (resultFromPromise) {
        console.log(resultFromPromise); // prints 10
        return resultFromPromise * 3;
}).then(function (resultFromPromise) {
        console.log(resultFromPromise / 2); // prints 15


});
```

- **Returning a Promise in the .then() method**: When you return a value in the then() method, you are basically asking the then() method to return a new Promise that immediately resolves to the return value. In short, the following 2 example code give the same result. Please take a look at the first .then() code for both examples. Both of the first .then() methods return a new Promise, but the syntax is different.
    - **Example of returning value in the .then() method**: Returns a new Promise

```
let myPromise = new Promise(function (resolve, reject) {
        setTimeout(function () {
        resolve(10);
        }, 3000);
});
myPromise.then(function (resultFromPromise) {
        return resultFromPromise * 3;
}).then(function (resultFromPromise) {
        console.log(resultFromPromise / 2);
});
```

- **Example of returning a new Promise in the .then() method**: Returns a new Promise

```
let myPromise = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve(10);
    }, 3000);
});
myPromise.then(function (resultFromPromise) {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            resolve(resultFromPromise * 3);
        }, 3000);
    });
}).then(function (resultFromPromise) {
    console.log(resultFromPromise / 2);
});
```

- **Multiple handlers of a promise**: Please note that calling the .then() method multiple times on an original Promise is not promise chaining. In the below example, we have multiple handlers for one promise. These handlers have no relationships. Also, they execute independently and don't pass the result from one to another like the promise chain above.

```
let myPromise = new Promise(function (resolve, reject) {
        setTimeout(function () {
                resolve(10);
        }, 3000);
});
myPromise.then(function (resultFromPromise) {
        console.log(resultFromPromise * 3); // prints 30
});
myPromise.then(function (resultFromPromise) {
        console.log(resultFromPromise / 2); // prints 5
});
myPromise.then(function (resultFromPromise) {
        console.log(resultFromPromise - 2);// prints 8
});
```

9.5 Promise (using .catch() to handle errors in Promise)

- **Handling errors in a Promise**: Asynchronous actions may sometimes fail. When there is an error, the corresponding promise becomes rejected. For instance, fetch fails if the remote server, from which we are planning to grab data from, is not available. In such cases, the most known way of handling errors (rejection) is using the .catch() method. The followings are the different ways of handling rejections.
- **Handling errors by passing null as the first argument to the .then()**: If we are interested only in errors, then we can use null as the first argument to our .then) function. See the example below:

```
let myPromise = new Promise(function (resolve, reject) {
        setTimeout(function () {
                reject("Unsuccessful!!");
        }, 2000);
});
 myPromise.then(null, function (error) {
        console.log(error);
});
```

  - However, this is not a great way to handle errors. The .catch() method below is preferred to handle Promise rejections as .catch() method catches errors that occurred anywhere in a promise chain.

- **Handling errors using the .catch() method**: If the async operation failed, JavaScript will reject the promise. When JavaScript rejects a Promise, the callback function in .catch() method is executed. The value passed by reject() is passed as a parameter to this callback.
  - **Syntax**:

```
let myPromise = new Promise(function (resolve, reject) {

});

myPromise.catch(function (errorResult) {

    console.log(errorResult);

});
```

  - **Example**:

```
let myPromise = new Promise(function (resolve, reject) {

    setTimeout(function () {

        reject(10);

    }, 3000);

});

myPromise.catch(function (rejectedResult) {

    console.log(rejectedResult - 1);

});
```

- **Note1: If the callback in .catch() returns a value, .catch() will return a new promise with resolved status**: If the handler function in catch() returns a value, it means that .catch() is returning a new Promise with a resolved value. Please note that this new Promise is resolved and has the returned value as its value. **Note:** Promises created when a .cathc() method returns a value will never be handled by .cathc(), but by .then(). **Example**:

```
let myPromise = new Promise(function (resolve, reject) {
    setTimeout(function () {
        reject(10);
    }, 3000);
});
myPromise.catch(function (result) {
    return result * 2;
    // code below is same as the returned value above
    // return new Promise(function (resolve, reject) {
        // resolve(result * 2);
    // });
})
.catch(function (returnedReulst) {
console.log(returnedReulst); // prints nothing
})
.then(function (returnedReulst) {
    console.log(returnedReulst); // Prints 20
});
```

- **Note 2: If the callback in .catch() throws an error, .catch() will return a new promise with reject()**: If the handler function in catch() throws an error, it means that .catch() is returning a new Promise with reject(). If reject() is the status of a promise, it can only by .catch(). **Example**:

```
let myPromise = new Promise(function (resolve, reject) {
    setTimeout(function () {
        reject(10);
    }, 3000);
});
myPromise
    .catch(function (result) {
        throw new Error(result * 3);
        // new Promise below is same as the above thrown value
        // return new Promise(function (resolve, reject) {
        // reject(result * 3);
        // });
})
.then(function (result) {
    console.log(result); //nothing prints here
})
.catch(function (result) {
    console.log(result); // prints 30
});
```

9.6 Promise (async-await)

- **Why Async/Await**: The "Async" function and the "Await" key words that are added on ECMAScript 2017 JavaScript edition just to simplify the way we use Promises. In other words, Async-Await operators/syntax are added on JavaScript to make asynchronous code writing easier and clearer.
    - **Async-Await in short**: Async makes a function return a Promise and await makes a function wait for the returned Promise
- **Creating promises using Promise vs. creating promises using Async/Await**: Promises can be created using Promise as well as using Async/Await
    - **Does Async/Await make Promises out of date?** No, not at all. When working with Async/Await we are still using Promises under the hood. That is why we need to understand Promises as there are even use cases where Async/Await doesn't cut it and we have to go back to Promises for help. One such scenario is when we need to make multiple independent asynchronous calls and wait for all of them to finish.
- **Creating promises using Promise**: This is basically what we have seen previously when we talked about creating promises using Promise. Look at the example below:

```
const job = function () {
        return new Promise(function (resolve, reject) {
                resolve("Success 1");
        });
}
job().then(function (result) {
        console.log(result); // prints "Success 1"
});
```

- **Creating promises using async**: To create a promise using async, all you need to write is the word “async” before any regular.
- **Async**: declares an asynchronous function by automatically transforming a regular function into a Promise. When called async functions resolve with whatever is returned in their body.

```
const job = async function () {
    return "Success 1";
};
```

  - **Note**: The above function returns a resolved promise with the result of "Success 1". That means, we can use the returned promise like below:

```
job()
    .then(function (result) {
        console.log(result);
    })
    .catch(function (result) {
        console.log(result); // returns nothing as async
        function   creates a promise with resolve value
    });
```

- **Await**: The keyword "await" is just a syntax that makes JavaScript wait until that promise settles and returns its result. Await pauses the execution of async functions. Meaning, when await is placed in front of a Promise call, it forces the rest of the code to wait until that Promise finishes and returns a result. **Note**: Only async functions enable the use of await. Meaning, await works only with Promises, it does not work with callbacks. Await can only be used inside async functions.
    - **Example**: Lets write a function that creates a Promise and consumes it in the same function. Notice that we have created a function myDisplay which has async keyword on it. This keyword makes it asynchronous, which means when this function is called, a Promise is returned, and normal code execution will commence as usual. Because we are using the await keyword before the Promise that just got returned, it forces the remaining code to wait until that Promise finishes and returns a result. It is only after waiting for the Promise to return the 300 value that any code after the await keyword starts to execute.

```
async function myDisplay() {
    let myPromise = new Promise(function (resolve, reject) {
        resolve(300);
    });
    return await myPromise;
myDisplay().then(function (value) {
    console.log(value);// returns 300
});
```

- **Note (Can't use await in regular functions)**: If we try to use await in a non-async function, there would be a syntax error:

```
function myDisplay() {
let myPromise = new Promise(function (resolve, reject) {
resolve(300);
});
        return await myPromise;//  SyntaxError: await is only valid with async
        }
```

- **Handling Errors in Async/Await by wrapping our cold with "try/catch" block**: Async function creates a promise with resolve value. If something goes wrong and we want to catch the error, we cannot just use the .catch() method like we did for ordinary Promise.

```
async function myDisplay() {
let myPromise = new Promise(function (resolve, reject) {
resolve(300);
});
try {
return await myPromise;
} catch (err) {
console.error(err);
}
        }
        myDisplay().then(function (value) {
console.log(value);
        });
```

- **Here are a couple of nice links to practice your understanding of Promises**
  - https://www.codingame.com/playgrounds/347/javascript-promises-mastering-the-asynchronous/its-quiz-time
  - https://www.codingame.com/get-a-job