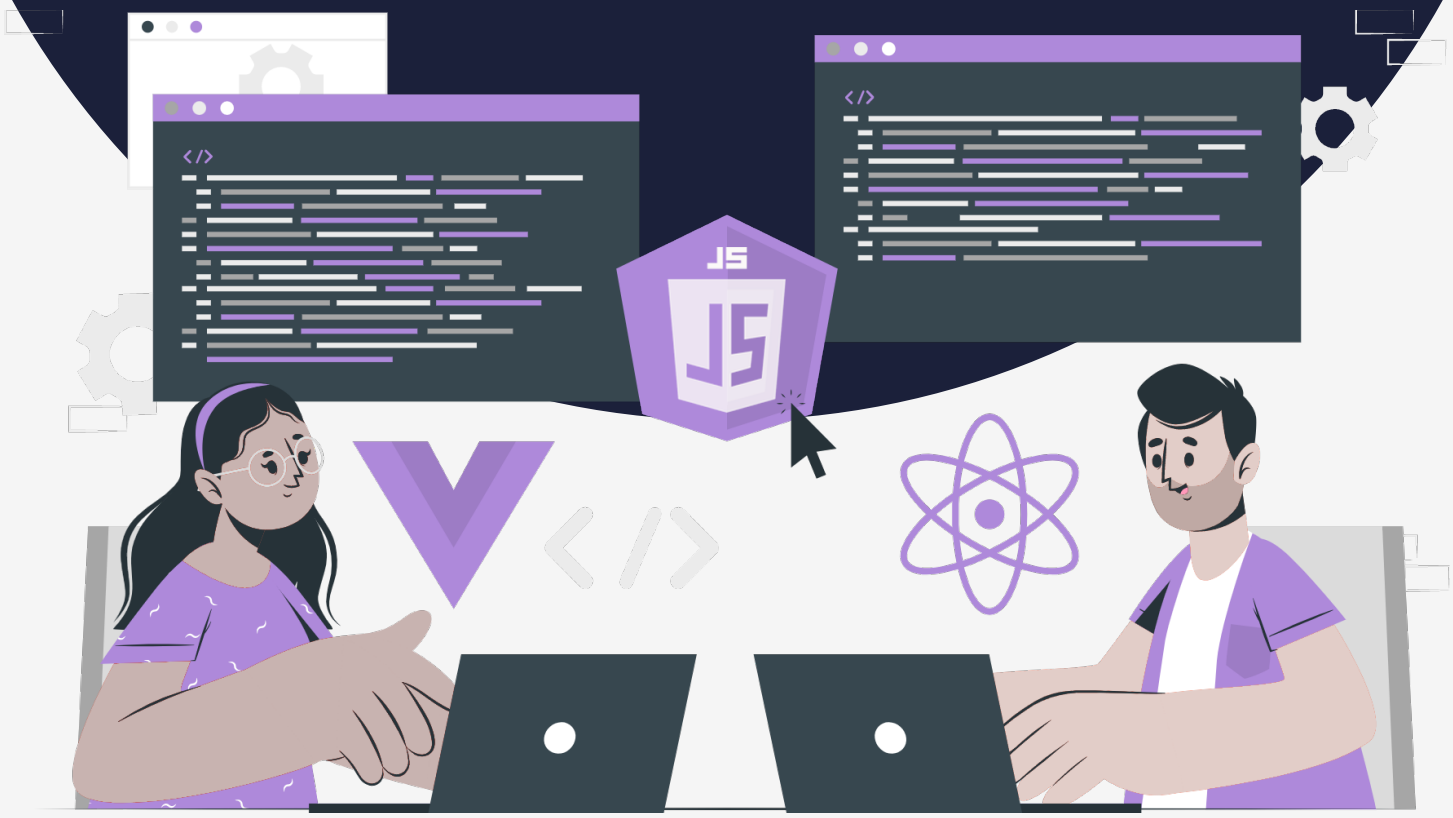
Lesson:

Callback



Topics Covered:

1. Introduction.
2. What is the need for a callback?
3. Synchronous callbacks.
4. Asynchronous callbacks.
5. Nested callbacks.
6. Callback Hell.

A callback is a function that is passed as an argument to another function and is executed once the main function has finished executing. The purpose of a callback is to allow a program to perform actions asynchronously. This is especially useful when dealing with tasks that take a long time to complete or when you want to execute multiple tasks at the same time.

Callbacks can be used in many different scenarios in programming, such as event handling, network requests, and user interactions. They provide a flexible and efficient way to execute code after a certain task has been completed, and can be a powerful tool for writing clean and efficient code. However, it's important to use callbacks carefully, as they can quickly become difficult to manage and lead to what is known as "callback hell."

We will be looking into all of these concepts in this lecture.

To put callback more simply, it is a function that will be called(executed) later after its parent function(the function in which it is passed as an argument) is done executing.

```
// Function to find the square of a number.

function squareOfANumber(num) {
  console.log(num * num);
}
function operation(num, func) {
  func(num);
}

// Calling the function.

operation(5, squareOfANumber);

// OUTPUT: 25
```

In the above code we have 2 functions.

The first function, squareOfANumber, takes a single parameter num and logs the square of that number to the console using the formula num * num.

The second function, operation, takes two parameters: num, which is the number to be squared, and func, which is a function that will be called and passed the num parameter. This function is designed to execute any arbitrary function that is passed to it as the func parameter.

Finally, the operation function is called with the arguments 5 and squareOfANumber. This means that the squareOfANumber function will be passed as the func parameter to the operation function, which will execute it with argument 5.

When the program is run, it will log the result of calling the squareOfANumber function with argument 5, which is 25. Therefore, the output of the program will be 25.

What is the need for a callback?

Javascript is a single-threaded language, which means, it executes the code sequentially one line after another. However, there are some cases where multiple tasks or operations can be executed concurrently without waiting for each other to complete them. This is asynchronous programming.

- Handling Asynchronous Operations: When we are dealing with an asynchronous operation, we don't want to wait for the operation to complete. Instead, you can register a callback function to handle the response when to execute.
- Event Handling: When you want to handle user events such as clicking a button, you can register a callback function to be executed when the event occurs.
- When a function fetches some data from an API that takes some time to get downloaded. We use the callback function here so that the function that needs the data will only get executed when all the required data is downloaded. By this, we can prevent running into errors because of the non-availability of some data that are needed by some other function.

Callback Functions can be used in two scenarios:

1. Synchronous operations.
2. Asynchronous operations.

In synchronous programming, tasks or operations are executed one at a time, in a sequential manner, and each task must complete before the next one can begin. In other words, the program waits for each task to finish before moving on to the next one.

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

function function1() {
  console.log("Executing Function 01");
}

function function2(callback) {
  callback();
  console.log("Executing Function 02");
}
```

```
function2(function1);

console.log("End");

/*
OUTPUT:
Start
Executing Function 01
Executing Function 02
End
*/
```

The above code demonstrates how a callback function can be passed as an argument to another function and then executed within that function. In this way, the program can execute different functions in a specific order, and each function can be designed to do a specific task. The use of callbacks allows for greater flexibility and modularity in the code, making it easier to maintain and reuse.

Here is the step-by-step execution of the code:

1. The program starts by logging the message "Start" to the console.
2. The function function1 is defined, which simply logs the message "Executing Function 01" to the console.
3. The function function2 is defined, which takes a single argument called callback. This argument is expected to be a function and is executed by calling it with parentheses () within the function2 function. After that, the message "Executing Function 02" is logged to the console.
4. The function2 function is called with function1 as its argument. This means that function1 is passed as the callback argument to function2.
5. The function2 function is executed, which calls the callback argument (which is function1 in this case). This causes the message "Executing Function 01" to be logged into the console.
6. After function1 is executed, the function2 function continues executing and logs the message "Executing Function 02" to the console.
7. Finally, the program logs the message "End" to the console.

Let’s now look at the asynchronous callback.

setTimeout is a function that is used to delay the execution of a piece of code for a specified amount of time.

When you call the setTimeout function, you provide it with arguments: it takes a callback function as its first argument, and it can also take additional arguments for the delay and any parameters to pass to the callback function.

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

setTimeout(() ⇒ {
  console.log("Set Timeout is being executed");
}, 2000);

console.log("End");

/*

OUTPUT:
StartEndSet Timeout is being executed

*/
```

Here, as we can see the code is not running sequentially. The console.log("End"); statement gets executed before the setTimeout() function. This is called asynchronous programming.

The browser or the nodeJS engine will not wait for the setTimeout() function to execute its callback function. In the meantime, it'll move to the next statement and execute that, in our case, that is console.log("End");.

However, after the 2-second delay has passed, the callback function is executed and logs the message "Set Timeout is being executed" to the console.

Nested Callbacks

Nested callbacks are a common pattern in asynchronous programming, where a callback function is called inside another callback function.

This pattern is used when you need to execute a series of asynchronous tasks, where each task depends on the output of the previous task.

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

setTimeout(function () {
  console.log("Executing Function 01");
  setTimeout(function () {
    console.log("Executing Function 02");
    setTimeout(function () {
      console.log("Executing Function 03");
    }, 1000);
  }, 1000);
}, 1000);

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

```
/*
OUTPUT:

Start
End
Executing Function 01
Executing Function 02
Executing Function 03

*/
```

The above code uses nested callbacks with setTimeout to delay the execution of each console log statement. When the code is executed, the Start and End are printed first, then the second console.log statement is executed after a delay of 1 second, and then the third console.log statement is executed after another delay of 1 second. Finally, the last console.log statement is executed after a total delay of 2 seconds.

Callback Hell.

Callback hell is a situation in asynchronous programming where multiple levels of nested callbacks make the code difficult to read, understand, and maintain. This can occur when you have to execute a series of asynchronous tasks, where each task depends on the output of the previous task, and you need to pass the results of each task to the next one.

When you use a lot of nested callbacks, the code can become difficult to read and maintain. Additionally, if there are any errors in the code, it can be difficult to pinpoint where the errors are occurring.

To avoid callback hell, there are several techniques you can use in JavaScript, such as Promises and async/await functions. We will be looking into them in further lectures. These techniques can help you write cleaner, more maintainable code when dealing with complex asynchronous operations.