Node.js - Callbacks Concept

- A callback function is simply a function that is passed as an argument into another function and it is called once the execution of the function(within which you defined this callback function) is finished.
- This is commonly seen in asynchronous APIs, Where you do not want to wait for a function to complete before doing some other task.
- Callback function can have the following arguments : **data object, result object** and (or) **error object** containing information regarding the task.

- The simplest example I can think of in JavaScript is the setTimeout() function. It's a global function that accepts two arguments. The first argument is the callback function and the second argument is a delay in milliseconds. The function is designed to wait the appropriate amount of time, then invoke your callback function.

```
setTimeout(function () {
  console.log("10 seconds later...");
 },  10000);
```

- We could rewrite the code above to make it more obvious.

```
var callback = function () {
console.log("10 seconds later...");
}; setTimeout(callback, 10000);
```

- Callbacks are used all over the place in Node because Node is built from the ground up to be asynchronous in everything that it does.
- For example, a function to read a file may start reading file and return the control to the execution environment immediately so that the next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as a parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results.

- Blocking Code Example
- Create a text file named **input.txt** with the following content −
- *This is just a demo of callbacks*
- Create a js file named **main.js** with the following code −

```
var fs = require("fs");
var data = fs.readFileSync('input.txt');
console.log(data.toString());
console.log("Program Ended");
```

(The fs module exposes two unique API functions: readFile and readFileSync.

The readFile function is asynchronous while readFileSync is obviously not.

)

- ***Output –***

```
This is just a demo of callbacks
Program Ended
```

- **Non-Blocking Code Example –**
- Update main.js to have the following code −

```
var fs = require("fs");
fs.readFile('input.txt', function (err, data) {
if (err)
return console.error(err);
console.log(data.toString());
});
console.log("Program Ended");
```

- ***Output –***

```
Program Ended
This is just a demo of callbacks
```

- **These two examples explain the concept of blocking and non-blocking calls.**

1 - The first example shows that the program blocks until it reads the file and then only it proceeds to end the program.

2 - The second example shows that the program does not wait for file reading and proceeds to print "Program Ended" and at the same time, the program without blocking continues reading the file.

- Thus, a blocking program executes very much in sequence. From the programming point of view, it is easier to implement the logic but non-blocking programs do not execute in sequence. In case a program needs to use any data to be processed, it should be kept within the same block to make it sequential execution.