20 - Async JS

Async JavaScript (Async JS) is a powerful concept that allows for non-blocking operations, enabling JavaScript code to run asynchronously. This is particularly useful for tasks like fetching data from a server, reading files, or performing other operations that might take time, without freezing the main thread. Below is a detailed explanation of key concepts and mechanisms used in Async JS.

1. The Event Loop

The event loop is the core mechanism that enables JavaScript's non-blocking behavior. JavaScript is single-threaded, meaning it can only execute one piece of code at a time. The event loop allows JavaScript to perform asynchronous tasks by handling them outside the main execution stack and processing their results when the main thread is free.

- **Call Stack:** Executes your code line by line.
- **Web APIs:** Browser-provided APIs (like `setTimeout`, `fetch`, etc.) that handle async operations.
- **Callback Queue:** Holds callbacks from async operations, waiting to be executed.
- **Event Loop:** Continuously checks if the call stack is empty and pushes callbacks from the queue onto the stack for execution.

2. Callbacks

A callback is a function passed as an argument to another function that gets executed after the async operation is completed.

Example:

```javascript
function fetchData(callback) {
    setTimeout(() => {
        callback("Data fetched");
    }, 1000);
}

fetchData((data) => {
    console.log(data); // "Data fetched" after 1 second
});
```

Pros: Simple and easy to use for small operations.

Cons: Can lead to "callback hell" where multiple nested callbacks make the code difficult to read and maintain.

3. Promises

Promises are an improvement over callbacks, providing a more structured way to handle asynchronous operations.

- **States of a Promise:**
  - **Pending:** The initial state.
  - **Fulfilled:** Operation was completed successfully.
  - **Rejected:** Operation failed.
- **Creating and Using Promises:**

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("Data fetched");
    }, 1000);
});

promise.then((data) => {
    console.log(data); // "Data fetched"
}).catch((error) => {
    console.error(error);
});
```

Pros: Improves code readability and error handling.

Cons: Still can get messy with multiple chained operations.

4. Async/Await

`async` and `await` provide a way to write asynchronous code that looks and behaves more like synchronous code. `async` functions return a promise, and `await` pauses the execution until the promise is resolved.

- **Example:**

```
async function fetchData() {
    try {
        let data = await new Promise((resolve, reject) =>
{
            setTimeout(() => {
                resolve("Data fetched");
            }, 1000);
        });
        console.log(data); // "Data fetched"
    } catch (error) {
        console.error(error);
    }
}
```

```
fetchData();
```

Pros:

- Makes asynchronous code look cleaner and more manageable.
- Easier to debug and maintain compared to callbacks and promises.

Cons: Requires understanding of promises, as `await` is used to handle them.

5. Error Handling

Handling errors in asynchronous code is crucial, especially when using Promises or async/await.

- **Promises:** Use `.catch()` to handle errors.

```
promise.catch((error) => {
    console.error(error);
});
```

- **Async/Await:** Use `try...catch` blocks.

```
async function fetchData() {
    try {
        let data = await someAsyncFunction();
    } catch (error) {
        console.error(error);
    }
}
```

6. Common Patterns

- **Parallel Execution:** Run multiple promises in parallel using `Promise.all`.

```
Promise.all([promise1, promise2]).then(([result1, result
2]) => {
    console.log(result1, result2);
});
```

- **Sequential Execution:** Run promises sequentially by chaining `.then()` or using `async/await`.

```
async function sequential() {
    let result1 = await promise1;
    let result2 = await promise2;
}
```

7. Async Iteration

`for await...of` is a loop structure that allows asynchronous iteration over data sources like streams.

- **Example:**

```
async function processData(dataArray) {
    for await (let data of dataArray) {
        console.log(data);
    }
}
```

8. Web APIs and Async JS

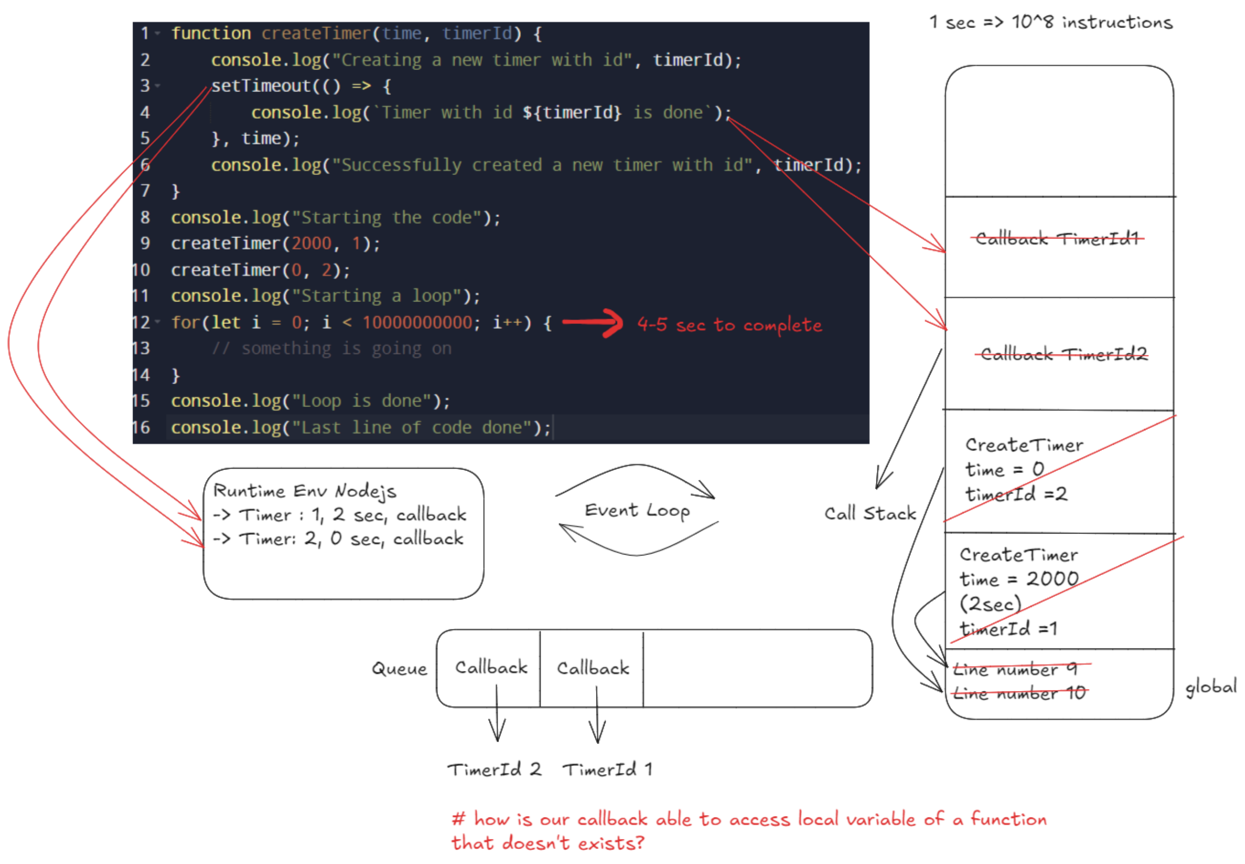
JavaScript in browsers is enriched by Web APIs that provide async capabilities, like `fetch`, `setTimeout`, and `WebSockets`. Understanding these APIs and how they work with async patterns is essential for modern web development.

Conclusion

Async JS is essential for creating smooth, non-blocking applications, especially in web development. By mastering callbacks, promises, async/await, and understanding the event loop, you'll be able to write efficient and manageable asynchronous code.

JS Code:

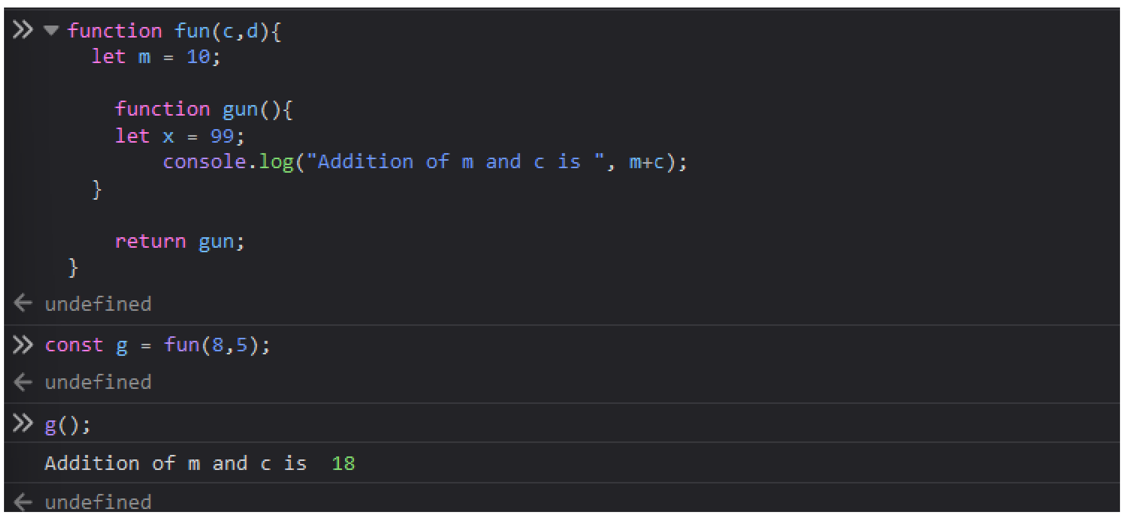
```
function createTimer(time, timerId) {
    console.log("Creating a new timer with id", timerId);
    setTimeout(() => {
        console.log(`Timer with id ${timerId} is done`);
    }, time);
    console.log("Successfully created a new timer with id", time
}
console.log("Starting the code");
createTimer(2000, 1);
createTimer(0, 2);
console.log("Starting a loop");
for(let i = 0; i < 10000000000; i++) {
    // something is going on
}
console.log("Loop is done");
console.log("Last line of code done");
```



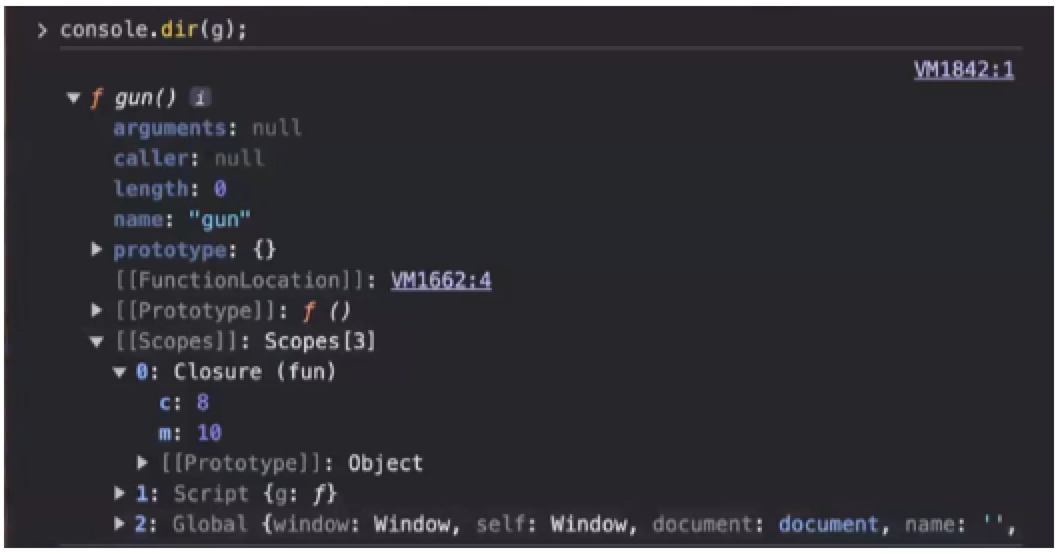
```
function createTimer(time, timerId) {
    console.log("Creating a new timer with id", timerId);
    setTimeout(() => {
        console.log(`Timer with id ${timerId} is done`);
    }, time);
    console.log("Successfully created a new timer with id", timerId);
}
console.log("Starting the code");
createTimer(2000, 1);
createTimer(0, 2);
console.log("Starting a loop");
for(let i = 0; i < 10000000000; i++) {
    // something is going on
}
console.log("Loop is done");
console.log("Last line of code done");
```

Output:

```
Starting the code
Creating a new timer with id 1
Successfully created a new timer with id 1
Creating a new timer with id 2
Successfully created a new timer with id 2
Starting a loop
Loop is done
Last line of code done
Timer with id 2 is done
Timer with id 1 is done
```



```
>> function fun(c,d){
     let m = 10;

       function gun(){
       let x = 99;
           console.log("Addition of m and c is ", m+c);
     }

       return gun;
   }
<- undefined
>> const g = fun(8,5);
<- undefined
>> g();
   Addition of m and c is  18
<- undefined
```



```
> console.dir(g);
                                                              VM1842:1
  ▼ ƒ gun() ℹ
      arguments: null
      caller: null
      length: 0
      name: "gun"
    ▶ prototype: {}
      [[FunctionLocation]]: VM1662:4
    ▶ [[Prototype]]: ƒ ()
    ▼ [[Scopes]]: Scopes[3]
      ▼ 0: Closure (fun)
          c: 8
          m: 10
        ▶ [[Prototype]]: Object
      ▶ 1: Script {g: ƒ}
      ▶ 2: Global {window: Window, self: Window, document: document, name: '',
```

Closure is a mechanism using which a function remembers the variables present in the outer function scope, even when the outer function execution is completed.

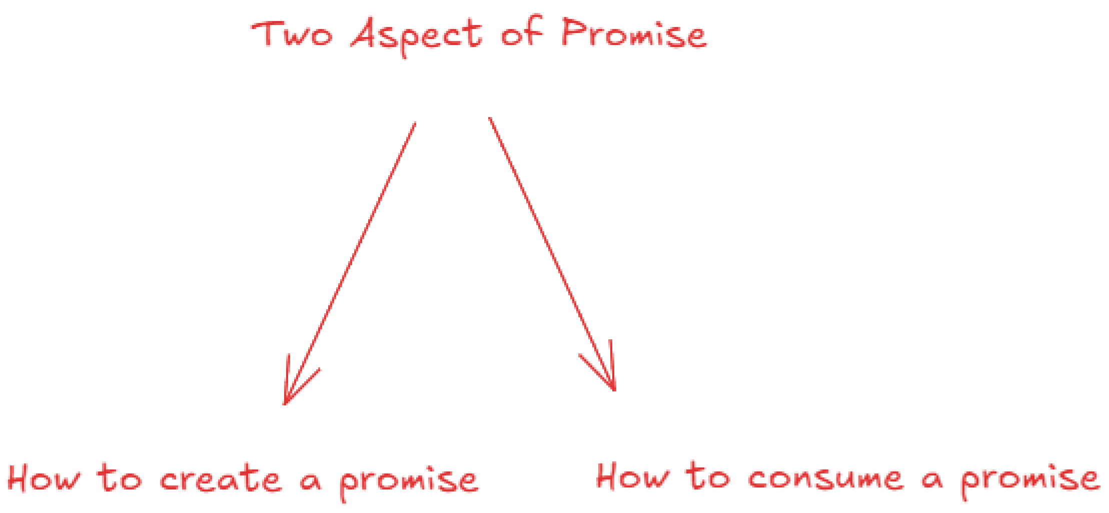
Disadvantages of Callback

1. Callback hell
2. IOC (Inversion of control)

To overcome callback hell, we use Promises in JavaScript. However, it's important to remember that you can still encounter issues similar to callback hell, often referred to as 'promise hell' or 'promise chaining hell,' if Promises are not used properly.

This highlights that while Promises can help manage asynchronous code better than callbacks, improper use can still lead to complexity, hence the importance of structured handling, like using `async/await`.

Promises in JS : (officially part of JS)



1. Promises are readability enhancers.
2. They are a special JS object, which can help us to control future related tasks.

`Note: Generally we prefer using promises when we have to deal with future tasks.(eg: 1. downloading some data 2. using a timer.)`

Instructions

Implement a set of dummy functions that mimic the behavior of the following operations:

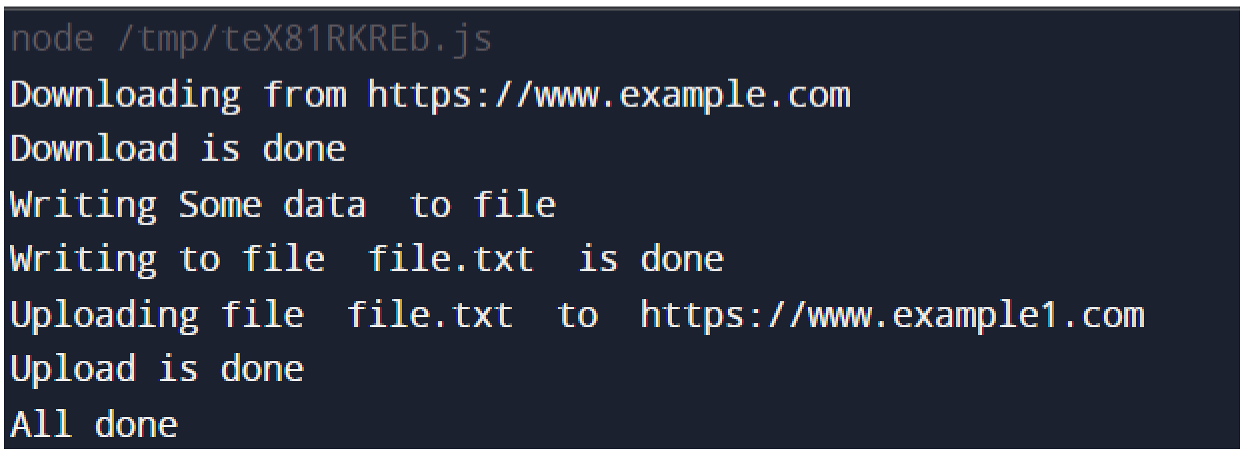
1. `download(url)`: This function should simulate downloading content from a specified URL.
2. `writeFile(filename, content)`: This function should simulate writing the downloaded content to a file.
3. `upload(filename)`: This function should simulate uploading the file to a server.

Now after you've implemented these functions, try to use them in a scenario where we first download a file, then write it to a disk and then upload it to a server.

```
function download(url, callback) {
    console.log("Downloading from", url);
    setTimeout(() => {
        console.log("Download is done");
        let downloadedData = "Some data";
        callback?.(downloadedData);
    }, 3000);
}

function writeFile(data, fileName, callback) {
    /* fileName tells the name of the file to be created in
    which data will be written */
    console.log("Writing", data, " to file");
    setTimeout(() => {
        console.log("Writing to file ", fileName, " is done");
        let status="Success";
        callback?.(status);
    }, 2000);
}

function upload(fileName, url, callback) {
    // fileName tells the name of the file to be uploaded
    console.log("Uploading file ", fileName, " to ", url);
    setTimeout(() => {
        console.log("Upload is done");
        let uploadStatus = "Success";
        callback?.(uploadStatus);
    }, 3000);
}


function process() {
    download("https://www.example.com", function handleDownload
    {
        writeFile(data, "file.txt", function handleWrite(status
            upload("file.txt", "https://www.example1.com",
            function handleUpload(uploadStatus) {
                console.log("All done");
            });
        });
    });
}

process();
```

1. What to do after downloading the data?
    - The process of downloading and what to do with the downloaded data is independent of each other.
2. What to do after downloading can be decided by whosoever is calling our download function.

▶ Execution



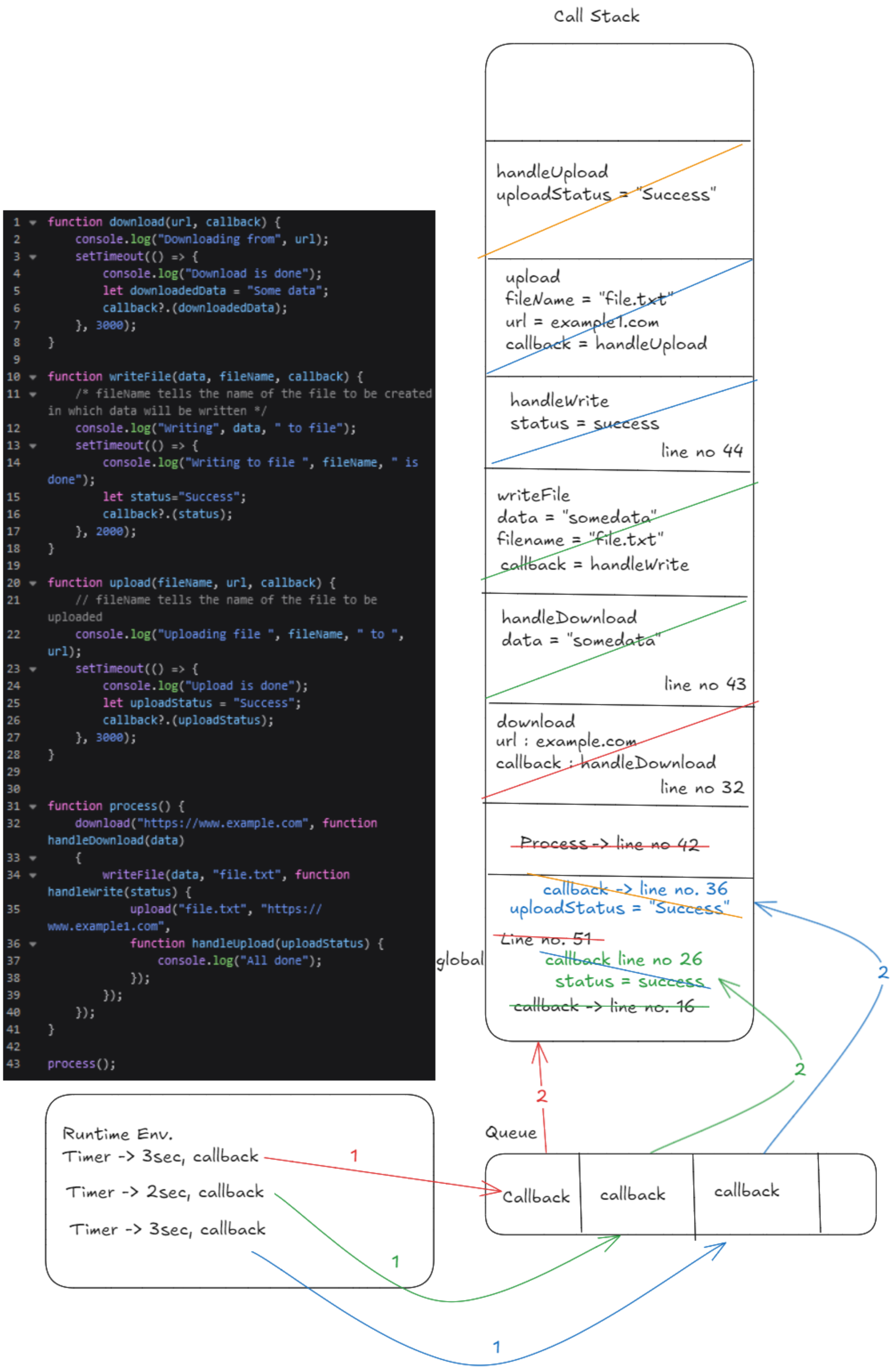
```
node /tmp/teX81RKREb.js
Downloading from https://www.example.com
Download is done
Writing Some data  to file
Writing to file  file.txt  is done
Uploading file  file.txt  to  https://www.example1.com
Upload is done
All done
```

Points to Note

- The functions are designed to simulate the process; they don't perform actual downloading, writing, or uploading.
- The scenario demonstrates the sequence of operations:

  download → write → upload.
- This structure is often used in workflows like file processing pipelines.

Dry Run:



```
1   function download(url, callback) {
2       console.log("Downloading from", url);
3       setTimeout(() => {
4           console.log("Download is done");
5           let downloadedData = "Some data";
6           callback?.(downloadedData);
7       }, 3000);
8   }
9
10  function writeFile(data, fileName, callback) {
11      /* fileName tells the name of the file to be created
    in which data will be written */
12      console.log("Writing", data, " to file");
13      setTimeout(() => {
14          console.log("Writing to file ", fileName, " is
    done");
15          let status="Success";
16          callback?.(status);
17      }, 2000);
18  }
19
20  function upload(fileName, url, callback) {
21      // fileName tells the name of the file to be
    uploaded
22      console.log("Uploading file ", fileName, " to ",
    url);
23      setTimeout(() => {
24          console.log("Upload is done");
25          let uploadStatus = "Success";
26          callback?.(uploadStatus);
27      }, 3000);
28  }
29
30
31  function process() {
32      download("https://www.example.com", function
    handleDownload(data)
33      {
34          writeFile(data, "file.txt", function
    handleWrite(status) {
35              upload("file.txt", "https://
    www.example1.com",
36                  function handleUpload(uploadStatus) {
37                      console.log("All done");
38                  });
39          });
40      });
41  }
42
43  process();
```

Call Stack
~~handleUpload uploadStatus = "Success"~~
~~upload fileName = "file.txt" url = example1.com callback = handleUpload~~
~~handleWrite status = success line no 44~~
~~writeFile data = "somedata" filename = "file.txt" callback = handleWrite~~
~~handleDownload data = "somedata" line no 43~~
~~download url : example.com callback : handleDownload line no 32~~
~~Process -> line no 42~~
~~callback -> line no. 36 uploadStatus = "Success"~~
~~Line no. 51~~
~~callback line no 26 status = success~~
~~callback -> line no. 16~~
global
Runtime Env.
Timer -> 3sec, callback
Timer -> 2sec, callback
Timer -> 3sec, callback
Queue
Callback
callback
callback
1
2
1
2
1
2

https://github.com/singhsanket143/AsyncJS-Tutorial/tree/master/Part2