Async programming - 1

Agenda

1. Synchronous and asynchronous programming
2. Callbacks
3. Async function using inbuilt modules
4. Event loop and parallel async operations

What is synchronous vs asynchronous operations in layman's term

Synchronous

1. Cars behind each other at toll - one must be cleared before other
2. Everything happens in a strict sequence, and at any given moment, only one operation is being processed

asynchronous programming

1. where multiple tasks can be in progress at the same time, and you don't have to wait for one task to complete before starting another. Tasks can be initiated and then put aside until their results are needed, allowing for other operations to be performed in the meantime.

whether the below code is synchronous or asynchronous

```
const task1 = () => {
   console.log('Task 1');
}


const task2 = () => {
    console.log('Task 2');
}

task1();
task2();
```

It is synchronous because it gets executed one after another

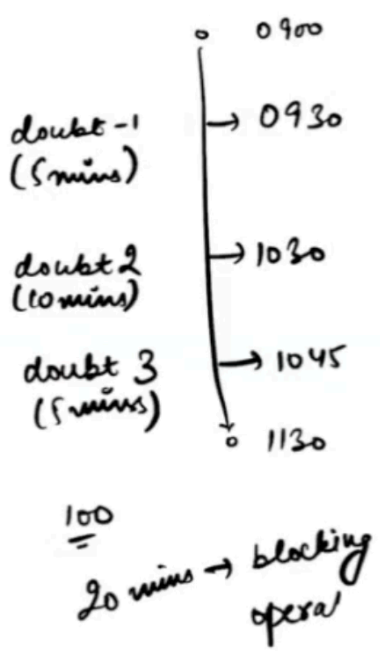
lets see to see the blocking nature of synchronous programming when we encounter a heavy time taking function

```
const task1 = () => {
   console.log('Task 1');
}

const heavyTask = () => {
   console.log('Heavy Task started in sync manner');
   const start = Date.now();// epoch time
   while (Date.now() - start < 3000) {
       // do nothing
   }
   console.log('Heavy Task done');
}

const task2 = () => {
   console.log('Task 2');
}

task1();
heavyTask();
task2();
```

Epoch time - some one on midnight of january 1970 started a clock and that is running ever since. Date.now() gives the time in milliseconds that have passed since the time this clock started
So the idea here is that if one of the function in the code is taking more time then it holds up the execution of rest of the code

Doing things in asynchronous manner

1. Lets take example of classroom setting here

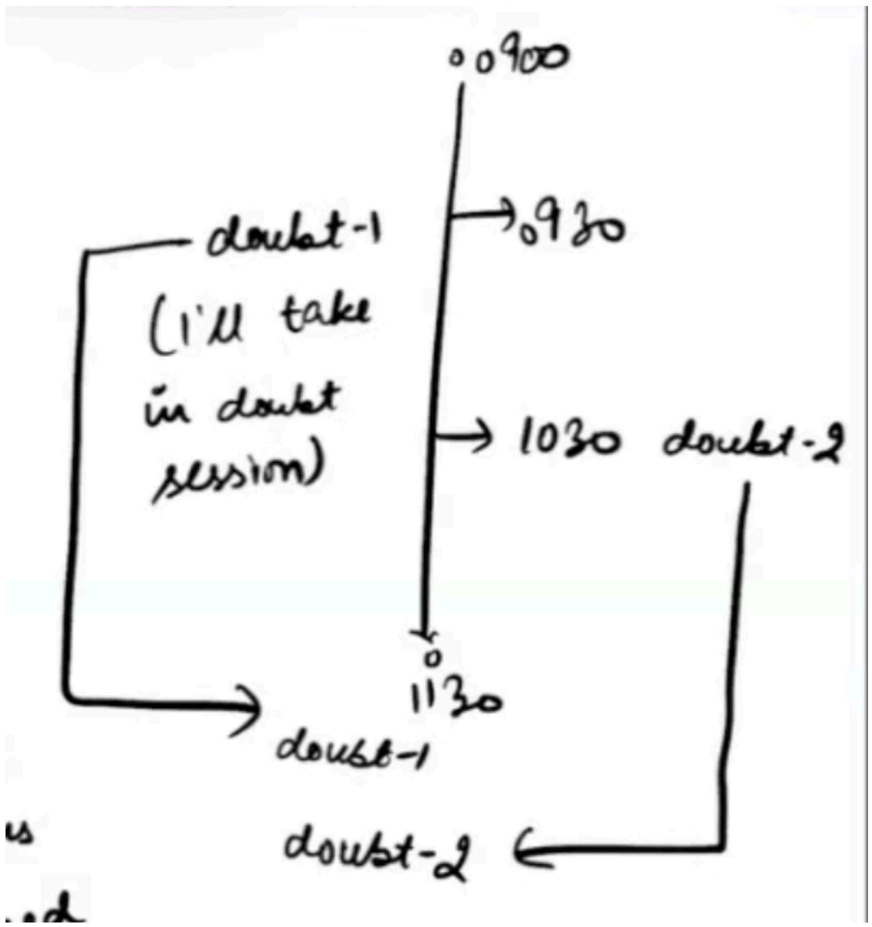


Our batch starts at 9:00, then I teach for some time. At 9:30, one of you have a doubt. Since i am teaching in a synchronous manner, i take that doubt. Now doubt resolution takses 5 mins let say. In that 5 min when I am answering the doubt of one learner, 99 of you have to wait (if there are 100 learners). So I am blocking 99 of you from learning

At 10:30, again someone has a doubt and it takes 10 mins
At 10:45 , there was one more doubt that it takes another 5 mins,

So basically when the lecture ends, I had spent 20 mins blocking multiple learners while taking doubt of one.

If I want to do this in a non blocking manner then how do I do it ?



When doubt comes and maybe that is not in the flow of the topics, I take that up in the doubt session

Lets see the async behavior now via code

Settimeout

setTimeout is a built-in function that allows you to execute a piece of code after a specified delay. The function is **non-blocking**, meaning it schedules the code to run in the future **without stopping the execution** of subsequent code.

```
const task1 = () => {
   console.log('Task 1');
}

const heavyTask = () => {
   console.log('Heavy Task started in sync manner');
   const start = Date.now();// epoch time
   while (Date.now() - start < 3000) {
       // do nothing
   }
   console.log('Heavy Task done');
}

const heavyTaskNonBlocking = () => {
    console.log('async Task skipping it for now');
    setTimeout(() => {
        console.log('Async Task done');
    }, 3000);

}

const task2 = () => {
    console.log('Task 2');
}

task1();
heavyTaskNonBlocking();
task2();
```

Examples of Asynchronous tasks

1. Network calls - your api calls happen over a network which might take time to hit the server and get the response. So we make them asynchronous
2. I/O FILES operation
3. DB operations

Extra notes

Why Some Browser APIs are Asynchronous
Handling of External Operations: Many browser APIs are asynchronous because they deal with operations that require waiting for something outside the JavaScript engine's immediate control, such

as fetching data over the network (fetch API), waiting for a specified time (setTimeout), or reading files (FileReader API in web browsers).

Non-Blocking Behavior: The web environment is designed to be responsive and non-blocking. Asynchronous APIs like setTimeout allow the JavaScript engine to schedule tasks to be executed later, letting other scripts run in the meantime or keeping the UI responsive. This is crucial for performing time-consuming or delay-introduced tasks without freezing the user interface.

The distinction between synchronous and asynchronous functions in JavaScript, especially for browser APIs, is largely about whether the function needs to wait for some external condition or resource.

console.log is synchronous because it performs an immediate operation without needing to wait. In contrast, setTimeout and similar APIs are asynchronous because they introduce a deliberate delay or depend on external conditions that require waiting.

Callbacks

Lets say you want to use the output of the async operation to carry out some next logic or operation but they get executed asynchronously so we dont know when they will be done and how to then plug in the next set of dependent logic

What is a Callback?

1. A callback is essentially a function that is passed into another function as an argument and then executed inside that function. Think of it as a way to say, "once you're done with your task, call me back with the results."
2. This is particularly useful for handling asynchronous operations, such as waiting for a file to load or a timer to complete

Why Use Callbacks?

1. Callbacks provide a way to ensure certain code doesn't execute until other code has already finished its execution.
2. This is crucial in JavaScript, especially for operations that take time to complete, such as retrieving data from a database, making network requests, or simply waiting for user input.

```javascript
const data = []
const fetchResponseBlockign = () => {
   console.log("making a sync api call ")
   // mimicking an api call
   const start = Date.now();
   while (Date.now() - start < 3000) {
       // do nothing
   }
   console.log("api call done")
   data.push({ id: 1, name: 'John' })
}
```

```
const renderResponse = () => {
   console.log("rendering the response")
   console.log(data[0].name)
}

fetchResponseBlockign()
renderResponse()
```

What do we see above- the blocking code as expected stops the execution and the rendering function is blicked. If it was UI, our screen would be left hanging

What to do - we discussed that we can make it async to prevent it from blocking other code.

We have seen that we can use setTimeout to make it non blocking

Let use change our code to make this non blocking

```
const data = []
const fetchResponseBlockign = () => {
   console.log("making a sync api call ")
   // mimicking an api call
   const start = Date.now();
   while (Date.now() - start < 3000) {
       // do nothing
```

```
    }
    console.log("api call done")
    data.push({ id: 1, name: 'John' })
}

const fetchResponseNonBlocking = () => {
    console.log("making a sync api call ")
    // mimicking an api call
    setTimeout(() => {
        console.log("api call done")
        data.push({ id: 1, name: 'John' })
    },5000)
}

const renderResponse = () => {
    console.log("rendering the response")
    console.log(data[0].name)
}

fetchResponseNonBlocking()
renderResponse()
```

Now what do we see - we get an error because althoug we made it non blocking which means the next piece of code runs, **but that next code is dependent on the result of the async code**.

somehow we have to make the rendering wait for the data to be available

Callbacks

We will use callbacks which are function passed as arguments to another function

```
const data = []
const task1 = () => {
   console.log('Task 1');

}
const fetchResponseBlockign = () => {
   console.log("making a sync api call ")
   // mimicking an api call
   const start = Date.now();
   while (Date.now() - start < 3000) {
       // do nothing
   }
   console.log("api call done")
   data.push({ id: 1, name: 'John' })
}

const fetchResponseNonBlocking = () => {
   console.log("making a sync api call ")
   // mimicking an api call
   setTimeout(() => {
       console.log("api call done")
       data.push({ id: 1, name: 'John' })
   },5000)
}
const fetchResponseNonBlockingCallback = (render) => {
   console.log("making a sync api call ")
   // mimicking an api call
   setTimeout(() => {
       console.log("api call done")
       data.push({ id: 1, name: 'John' })
       render()// callback added here
   },5000)
}

const renderResponse = () => {
   console.log("rendering the response")
   console.log(data[0].name)
}
const task2 = () => {
   console.log('Task 2');

}
task1()
fetchResponseNonBlockingCallback(renderResponse)
task2()
// renderResponse()
```

Now we see that task1 is done , fetch data is started, then task2 is done and now when data is available, render method is called

Async behavior with fs module

1. What a module is and how we can import a module

    a. Module for now we can understand that it is a bunch of function with lot of code and we are exposing all that we need as a single unit
    b. We can create custom module or like now we are going to use an inbuilt node module that deals with file systems on operating system
    c. So module is nothing but a packaged collection of functions that do a certain thing
2. Create a file fs.js
    a. Here we will import the fs module
    b. Create two files file1.txt and have large text like node js wiki content
    c. File 2 can be a small file

Code

Let us see the synchronous way of accessing the file data

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

console.log("starting")

const data1 = fs.readFileSync('file1.txt')
console.log("data of file1", data1)

const data2 = fs.readFileSync('file2.txt')
console.log("data of file2", data2)
```

Create a big file programmatically

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

console.log("starting")

// const content = Math.random().toString(36).repeat(1000000)
// fs.writeFileSync('file1.txt', content)

const data1 = fs.readFileSync('file1.txt')
console.log("data of file1", data1.toString())

const data2 = fs.readFileSync('file2.txt')
console.log("data of file2", data2.toString())
```

Run the code - what we see is that since file1 is a big file, it blocks my code till it completes and then only my second file is read

Now changing this to run in async mode

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

console.log("starting")

// const content = Math.random().toString(36).repeat(1000000)
// fs.writeFileSync('file1.txt', content)

// const data1 = fs.readFileSync('file1.txt')
// console.log("data of file1", data1.toString())

fs.readFile('file1.txt', (err, data) => { //  callback for read
    if (err) {
        console.log("error", err)
```

```
   } else {
       console.log("data of file1", data.toString())
   }
})

const data2 = fs.readFileSync('file2.txt')
console.log("data of file2", data2.toString())

console.log("ending")
```

Now we see that it prints starting, then file 2 then ending and finally file 2 content

Now let us make the second file reading asynchronous as well and see the behavior

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

console.log("starting")

// const content = Math.random().toString(36).repeat(1000000)
// fs.writeFileSync('file1.txt', content)

// const data1 = fs.readFileSync('file1.txt')
// console.log("data of file1", data1.toString())
```

```
fs.readFile('file1.txt', (err, data) => {
   if (err) {
       console.log("error", err)
   } else {
       console.log("data of file1")

   }
})

fs.readFile('file2.txt',(err, data) => {
   if (err) {
       console.log("error", err)
   } else {
       // console.log("data of file2", data.toString())
       console.log("data of file2", data.toString())
   }

})

console.log("ending")
```

Behavior: we see that the order reversed, the content of file 2 was read before than file1 and we also see the time taken to read the file 1

** now let us make the content of both the files same

Now run the code

Now what we see is that the order is not guaranteed in asynchronous flow

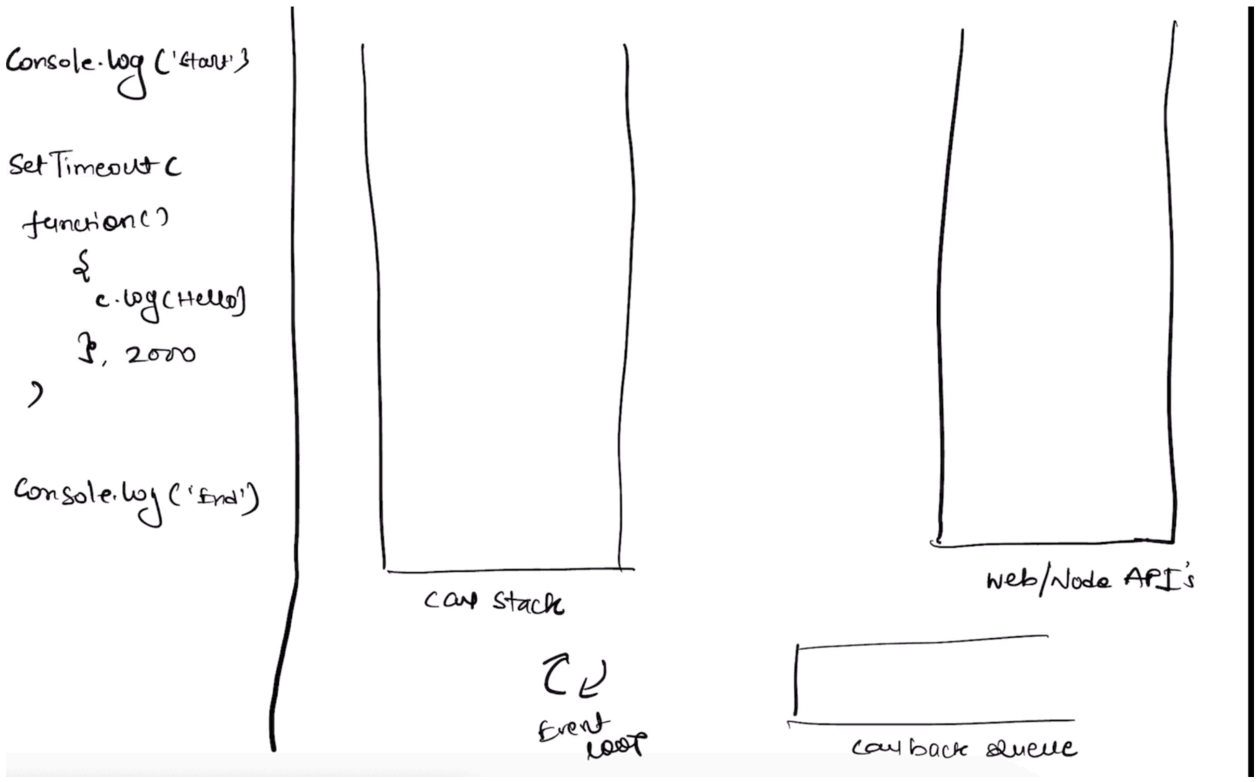
This is called parallel asynchronous processing

Event Loop

Create another file eventLoop.js

```
console.log("starting")

setTimeout(() => {
   console.log("timer1")
}, 10)

setTimeout(() => {
   console.log("timer2")
}, 0)

console.log("ending")
```

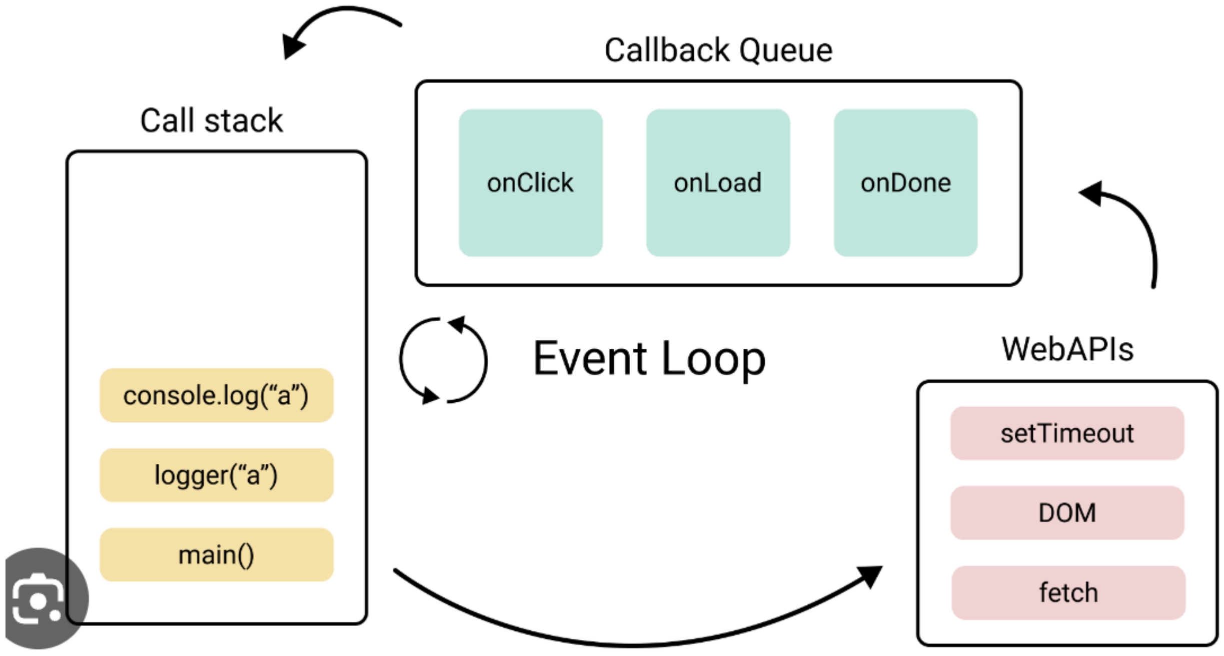
what would be the order



1. First the console.log(start') runs
    a. This line of code logs "starting" to the console. It's a synchronous operation and executes immediately.
2. Then the engine encounters setTimeout and it handover the counter part to the web / node api to keep track of the counter
    a. When JavaScript encounters a setTimeout() function, it sets up a timer. **This timer is handled outside of the JavaScript engine, typically by Web APIs provided by the browser or by the libuv library in Node.js**.
3. The execution does not wait and it moves to the last line
4. When the timer reaches the end, the callback is moved to the callback queue
5. Now the interesting and important part

   a. When the call stack is empty, the event loop has the responsibility of moving the call back from the queue to the call stack one by one
   b. setTimeout(() => { console.log("timer2") }, 0): It's important to note that a delay of 0 doesn't mean the callback executes immediately after this line. It means the callback will be executed as soon as the call stack is empty and after any other pending events in the event queue.
   c. After all the synchronous code has executed, the call stack is empty. The event loop then checks the callback queue.
   d. **The event loop moves callbacks from the event queue to the call stack one by one, as long as the call stack is empty**.
   e. In our example, "timer2" will be pushed to the callback queue immediately because the delay is 0, but it will not execute until the synchronous code has finished and the call stack is clear.
6. Order of Execution:
   a. The console.log("starting") and console.log("ending") are logged first because they are synchronous.
   b. The setTimeout callbacks are moved to the queue and will execute in order of their delays and the state of the call stack and event queue. Even though "timer2" has a delay of 0, it will be executed after "ending" because the call stack needs to be clear for the event loop to push the callback onto the stack.

c. "timer1" has a delay of 10, which means it will be placed in the queue after "timer2" and will execute after "timer2".



Use this online tool to visualise event loop - https://www.jsv9000.app/

Additional resources -
https://www.freecodecamp.org/news/asynchronous-programming-in-javascript/

https://www.freecodecamp.org/news/nodejs-eventloop-tutorial/

Order of execution of asynchronous operations

1. In the fs.js example, we are reading two files. If i have to ensure that file 2 is read only after file 1 is done reading, how do I make that possible
2. We are moving the reading of file 2 inside callback of file1.

```
fs.readFile('file1.txt', (err, data) => {
   if (err) {
       console.log("error", err)
   } else {
       console.log("data of file1")
       console.timeEnd("file1")
       fs.readFile('file2.txt',(err, data) => {
           if (err) {
               console.log("error", err)
           } else {
               // console.log("data of file2", data.toString())
               console.log("data of file2", data.toString())
               console.timeEnd("file2")
           }

       })
   }
})
```

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.
4. This leads to a situation of nested callbacks and we fondly call this as callback hell / pyramid of doom (will see in the next class)