# JS-5

# Intro to Async Programming and event loop

## Agenda

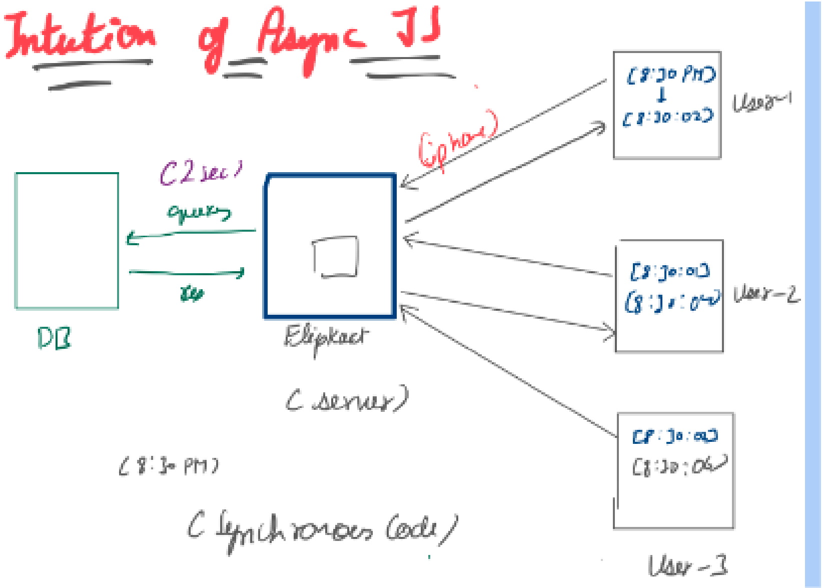
- Intuition of asynchronous code execution
- sync vs async function
- JS and Environment
- callbacks and event loop
- Objects

## Intuition of Async JS

### Case Scenario

Let's assume that there's the Big Billion Day Sale on Flipkart and there's a huge discount on the iPhone 14 Pro. Moreover, there are three users who arrive on the website in quick succession (after an interval of 1 second each).

- The User 1 arrives at 8:00 PM and requests for the page of iPhone 14 Pro
- The Flipkart server requests the page from the database through a query
- It takes 2 seconds to fetch the answer.



`Question`

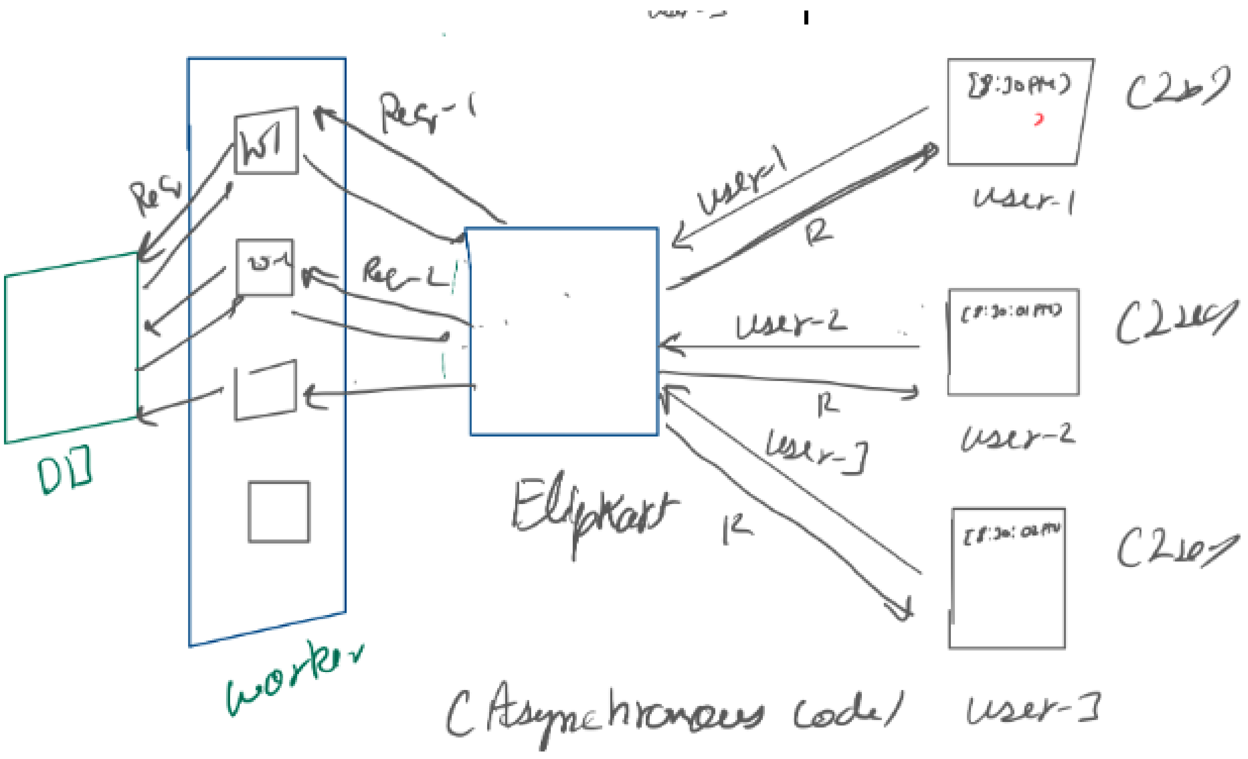
- Can user 2 make his request right at the moment he arrives or does he need to wait until the request-response finishes for user 1?
- **Answer:** User 2 will have a waiting time of 1 sec, following which he'd be able to make a request.

Now, user 2 arrives and asks for the Nokia's page and gets the response back at 8:00 PM and 4 seconds (since he made a request at 8:00 PM & 2 seconds).

Eventually, the user 3 will eventually have a waiting time of 2 seconds and also a delay in response. This continues to increase for the newly arriving users.

> *So, this illustrates the fact that this approach is not appropriate for handling numerous requests.*

**Solution:**



Assume we have a group of workers where the Flipkart server can delegate the tasks to the workers.

The workers will fetch the answer from the database and get the response which is then sent back to the user.

This is how the multiple requests can be handled when it's solely the responsibility of workers to fetch the response.

## Synchronous and Asynchronous functions

Here's how the synchronous code works:

```
/**
 * Synchronous code -> the code that executes line by line
 */

console.log("Before");
function fn() {
    console.log("I am fn");
}

fn();
console.log("After");
```

**Output:**

```
Before
I am fn
After
```

So, here the statement "Before" gets printed and the function `fn()` is allocated a memory space and is then executed. The statement "After" gets printed at the end.

## Async functions with callbacks

Now, let's see what happens in the case of Asynchronous programming:

```
/**
 * Asynchronous code -> piece of code that's executed at the current point
of time
 * and other piece of code is executed on later part
 */

//1
```

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

function fn() {
    console.log("I am fn");
}
//3
setTimeout(fn, 2000);
//2
console.log("After");
```

**Output:**

```
Before
After
I am fn
```

Here, the function `setTimeout` had been called earlier but since there's a delay of 2000 milliseconds, it would be executed once the "Before" and "After" get printed.

### Ask yourself

- Is `console.log()` a part of the JavaScript language?
- **Answer:** No, it's not.
- Is window a part of the JavaScript language?
- **Answer:** The `window` object is not actually a part of the JavaScript language itself. Instead, it is a feature provided by web browsers to enable interaction between JavaScript code and the browser environment.

> *the Environment provides the features whereas JavaScript provides us the logic.*

## JS and environment

For example there different platform and there enviornments

|  | Env |  | Apps |
|---|---|---|---|
| → | React Native | → | Mobile Apps |
| → | Electron | → | Desktop App |
| → | Browser | → | Frontend Apps |
| → | Node | → | Server |

As for the browser, the features are provided by the Web API and logic is provided by JavaScript. However, Node provides both Node API as well as JS logic which enables to use the same logic for both frontend as well as backend.

fact As a programmer, one cannot create asynchronous functions. They are provided by the Environment itself.

Now, there's a simple Question following what we've discussed.

---

## quiz 1

Here's the code. And what will be the output here? (What gets printed in the console)

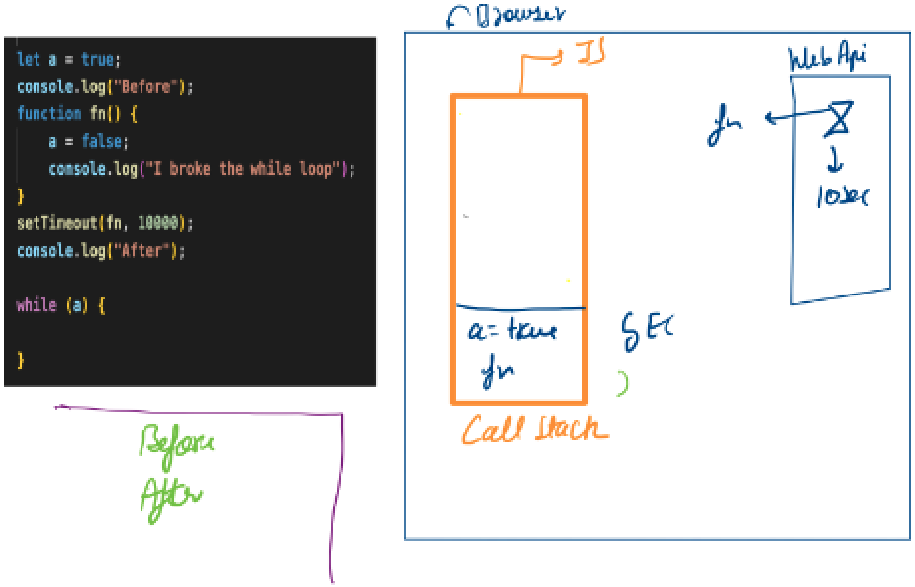
```
let a = true;
console.log("Before");
setTimeout(() => {
    a = false;
console.log("I broke the while loop");
}, 1000);
console.log("After");

while(a){

}
```

### Output:

```
Before
After
(the while loop never terminates)
```

Now, let's perceive why it happens.

**Note:** You'll get the same output in every environment.



```
let a = true;
console.log("Before");
function fn() {
    a = false;
    console.log("I broke the while loop");
}
setTimeout(fn, 10000);
console.log("After");

while (a) {

}
```

### Explanation:

We need to have clarity of each of the lines in the code.

- a's value is set to true in the callstack
- "Before" gets printed in the console
- Now, `setTimeout` is a feature provided by the Web/Node API, so it goes to the Node API. Moreover, it contains a function with some logic.
- Even if the `setTimeout` wishes to go to the callstack, it cannot until the callstack is empty.
- However, the value of `a` will be true forever and therefore, the while loop never terminates.

## Event loop with callback queue

The event loop is a fundamental concept in JavaScript that governs how asynchronous operations are managed and executed within a single-threaded environment.

The event loop continuously checks the call stack and the callback queue. If the call stack is empty, it takes the first function from the callback queue and pushes it onto the call stack for execution.

This process ensures that the asynchronous code is executed when the call stack is clear.

**Takeaways:**

- JavaScript is single-threaded. It provides us the logic whereas the Web API provides the features.
- **Callback queue:** When an asynchronous operation (like a `setTimeout` or an event listener) is ready to be executed, its callback function is placed in the callback queue.
- **Event loop:** The event loop continuously checks the call stack and the callback queue and pushes the function from the callback to the callstack.
  Here's are following rules you have to always remember about how JS is executed by event loop

```
/****
 * 1. Every line of code that you wrote in your js file -> will only execute in call stack
 * 2. For a cb coming from WebAPIS to execute -> callstack should be empty
 * 3. callback queue :As soon as task of asynchronous function is done there cb is immediatley
 * enqueued in that queue
 * 4. event loop : checks the queue and as soon as a cb arrived in the queue it continosly
 * check if callstack is empty or not and pushed that cb in callStack .
 * **/
```

## Quiz 2

What will be the output of this code?

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

const cb2 = () => {
        console.log("Set timeout 1");
        while(1){

        }
}

const cb1 = () => console.log("hello");

setTimeout(cb2, 1000)

setTimeout(cb1, 2000)

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

### Output:

```
Before
After
Set timeout 1
(while loop continues)
```

### Explanation:

- "Before" gets printed and then we have `const` variable `cb2` and `cb1`.
- Both the `setTimeout` are sent to the Web API.
- We also have a callback queue where `cb2` goes first and is followed by `cb1`.
- Since the callstack is empty, the setTimeout with `cb2` goes to the callstack.
- It contains a while loop that won't terminate, so the code continues to be in the same state waiting for the termination of the loop.

## Quiz 3

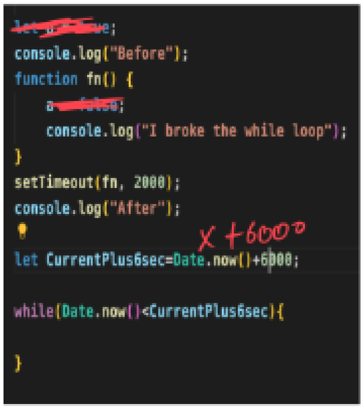
What will be the output if we modify the code to be this way?

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

const cb2 = () => {
        console.log("Set timeout 1");
        let timeInFuture = Date.now() + 5000;
        while(Date.now() < timeInFuture){

        }
}

const cb1 = () => console.log("hello");

setTimeout(cb2, 1000)

setTimeout(cb1, 2000)

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

**output:**

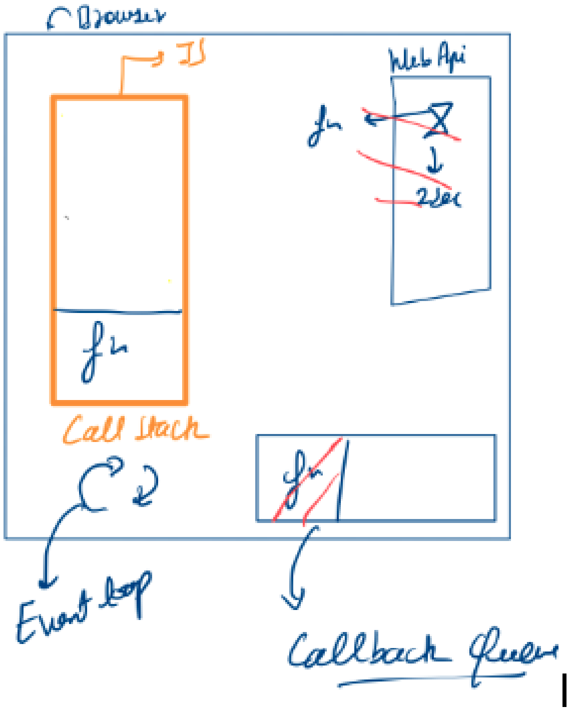
```
Before
After
Set timeout 1
.
.
.
hello (after 6 seconds)
```

**Explanation:** The "hello" gets printed after 6 seconds because the `cb2` has a `setTimeout` of 1 second, followed by the wait for 5 more seconds in the while loop. The `cb2` that has a waiting time for 2 seconds can now get executed immediately and "hello" gets printed.



```
~~let a = true~~;
console.log("Before");
function fn() {
    ~~a = false~~;
    console.log("I broke the while loop");
}
setTimeout(fn, 2000);
console.log("After");

let CurrentPlus6sec=Date.now()+6000;

while(Date.now()<CurrentPlus6sec){

}
```

```
Before
After
I broke the
while loop
```



## Objects in JavaScript

### Introduction

We have covered non-primitives like functions and arrays. Now, let's understand objects in JavaScript.

### Basic Concepts

- **Objects**: Collections of key-value pairs. Keys can be strings or numbers, and values can be any valid JavaScript type.
- **Properties**: Values in an object.
- **Methods**: Functions in an object.

### Creating and Modifying Objects

1. **Creating an Empty Object**

```
let obj = {};
```

2. **Adding Key-Value Pairs**

```
obj.name = "Jasbir";
obj.age = 28;
```

3. **Updating Values**

```
obj.age = 25;
```

4. **Printing an Object**

```
console.log(obj);
```

## Detailed Example

Let's look at a more detailed example of an object:

```
let cap = {
    firstName: "Steve",
    lastName: "Rogers",
    movies: ["First Avenger", "Winter Soldier", "Civil War"],
    address: {
        state: "Newyork",
        district: "Manhatten"
    },
    isAvenger: true,
    age: 35,
    sayHi: function () {
        console.log("cap say's hi");
    }
}
```

## Working with Objects

1. **Printing the Entire Object**

```
console.log("cap", cap);
```

### 2. Accessing Values

```
console.log("firstName", cap.firstName);
console.log("second movie:", cap.movies[1]);
console.log("state", cap.address.state);
```

### 3. Calling Methods

```
cap.sayHi();
```

### 4. Adding New Properties

```
cap.friends = ["Tony", "Peter", "Bruce"];
cap.isAvenger = false;
```

### 5. Deleting Properties

```
delete cap.movies;
console.log("cap", cap);
```

### 6. Looping Through an Object

```
for (let key in cap) {
    console.log("key:", key, "value:", cap[key]);
}
```