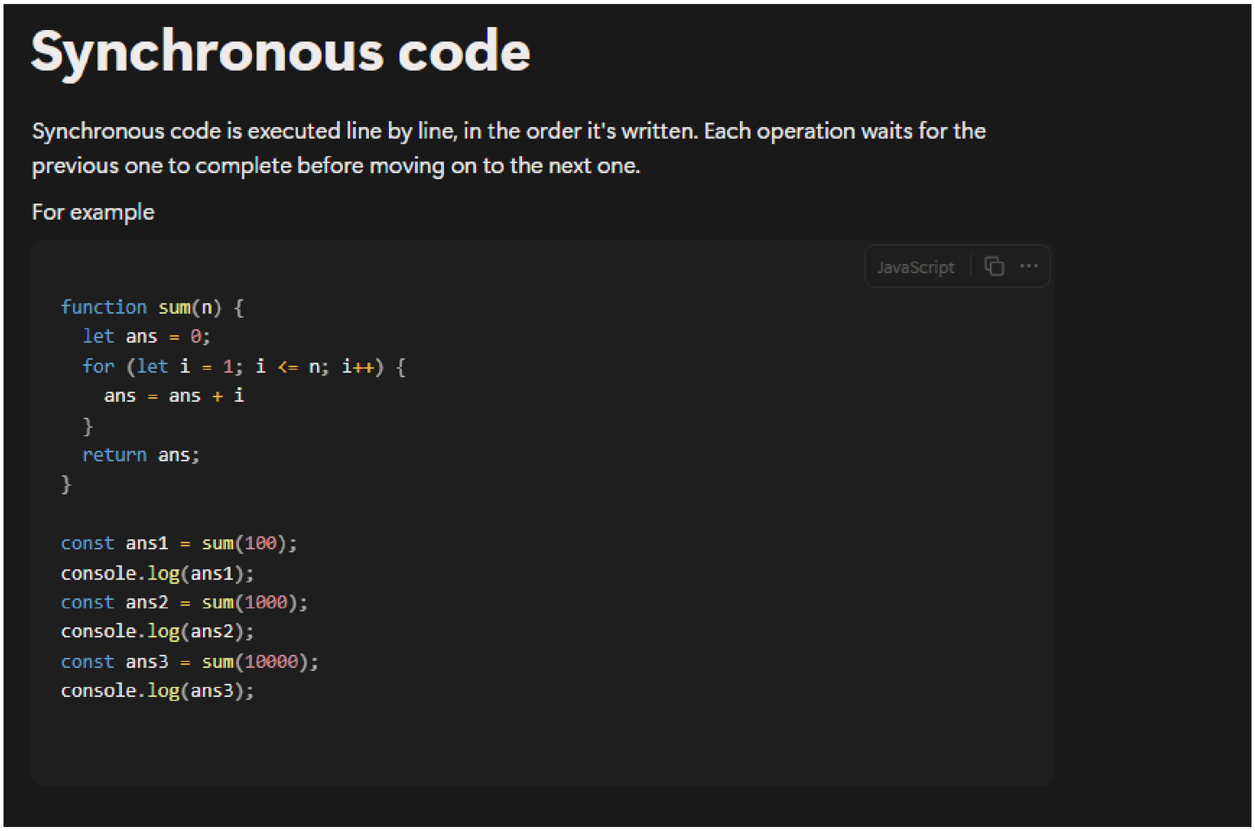
Notes: https://petal-estimate-4e9.notion.site/Class-2-JS-101-2f17dfd107358074a477dc0f71bf9a11

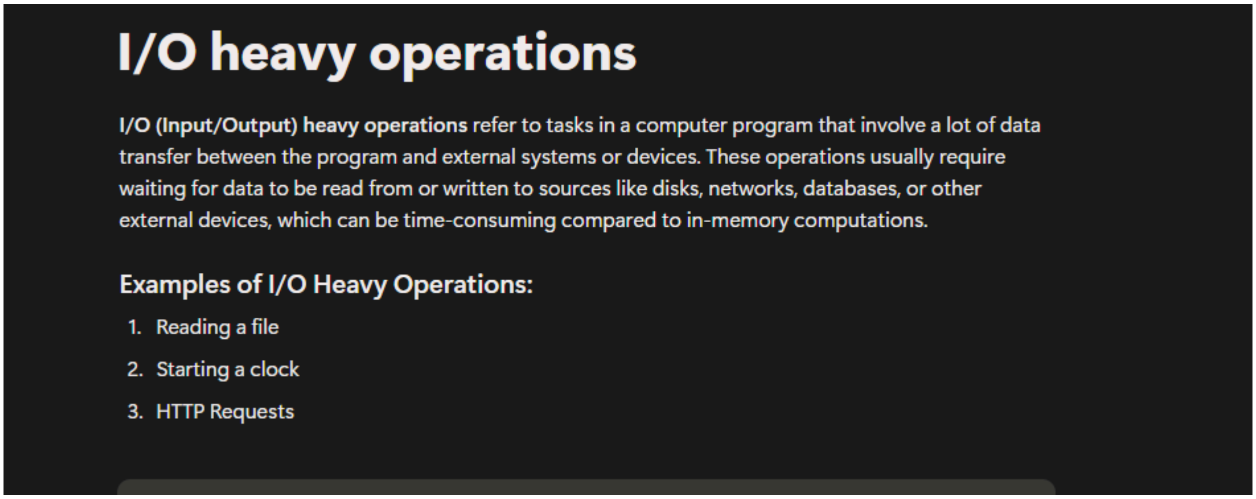


Synchronous code

Synchronous code is executed line by line, in the order it's written. Each operation waits for the previous one to complete before moving on to the next one.

For example

```javascript
function sum(n) {
  let ans = 0;
  for (let i = 1; i <= n; i++) {
    ans = ans + i
  }
  return ans;
}

const ans1 = sum(100);
console.log(ans1);
const ans2 = sum(1000);
console.log(ans2);
const ans3 = sum(10000);
console.log(ans3);
```



I/O heavy operations

I/O (Input/Output) heavy operations refer to tasks in a computer program that involve a lot of data transfer between the program and external systems or devices. These operations usually require waiting for data to be read from or written to sources like disks, networks, databases, or other external devices, which can be time-consuming compared to in-memory computations.

Examples of I/O Heavy Operations:

1. Reading a file
2. Starting a clock
3. HTTP Requests

In this example, the `index.js` program is dependent on the Operating System because it needs the OS to read the file (`a.csv`) from disk.
https://petal-estimate-4e9.notion.site/I-O-heavy-operations-2f17dfd107358038a458dea99e0f8ef4

When we call something like:

read("a.csv")

Node.js does not read the file by itself using the CPU. Instead, it asks the Operating System to perform the file read operation.

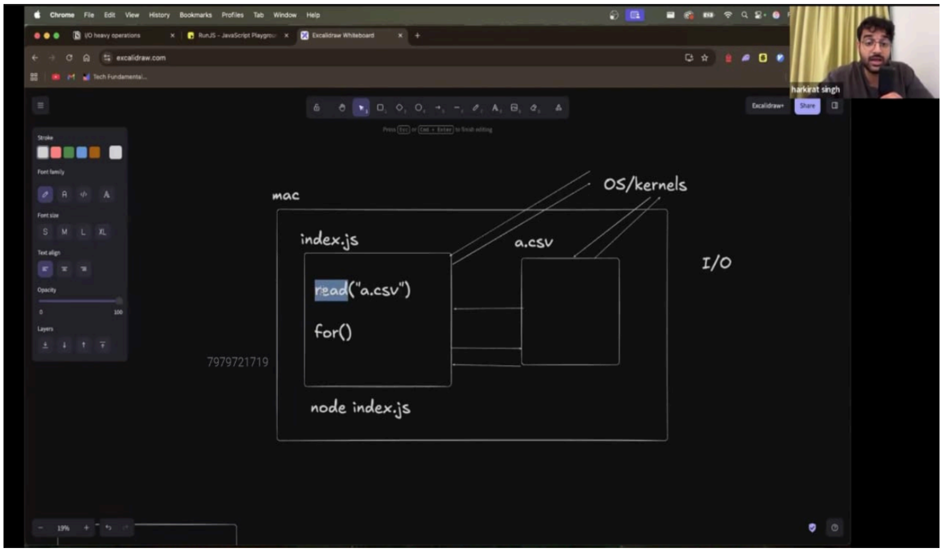
So the total time taken depends on:

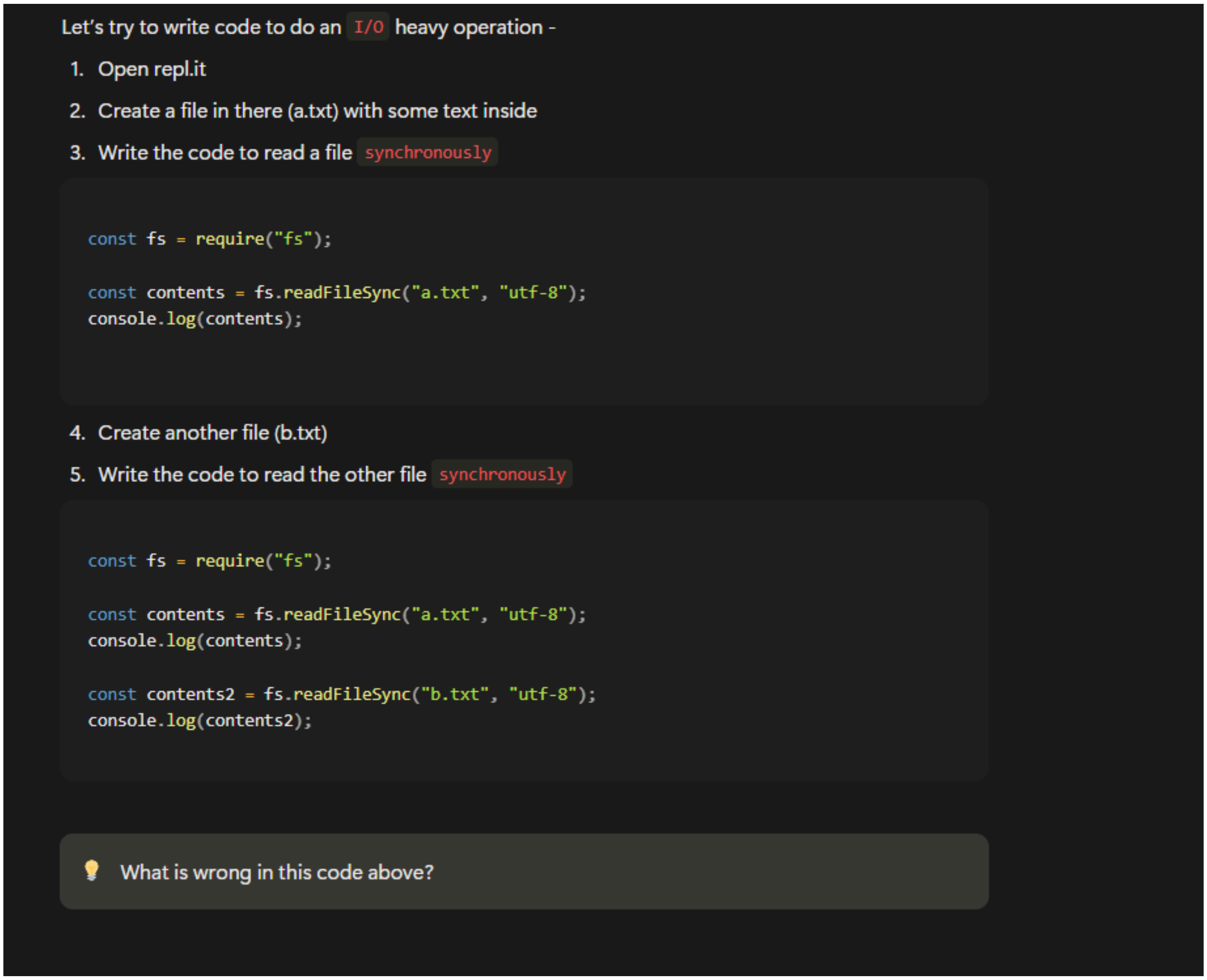
- How fast the OS can access the disk
- File size
- Disk speed (SSD vs HDD)
- System I/O performance

During this waiting time, the CPU is mostly idle (not doing heavy computation).

That's why this task is called:

👉 **I/O Heavy (Input/Output Heavy)**
because it depends on input/output operations handled by the Operating System, not on CPU processing power.





Let's try to write code to do an `I/O` heavy operation -

1. Open repl.it
2. Create a file in there (a.txt) with some text inside
3. Write the code to read a file `synchronously`

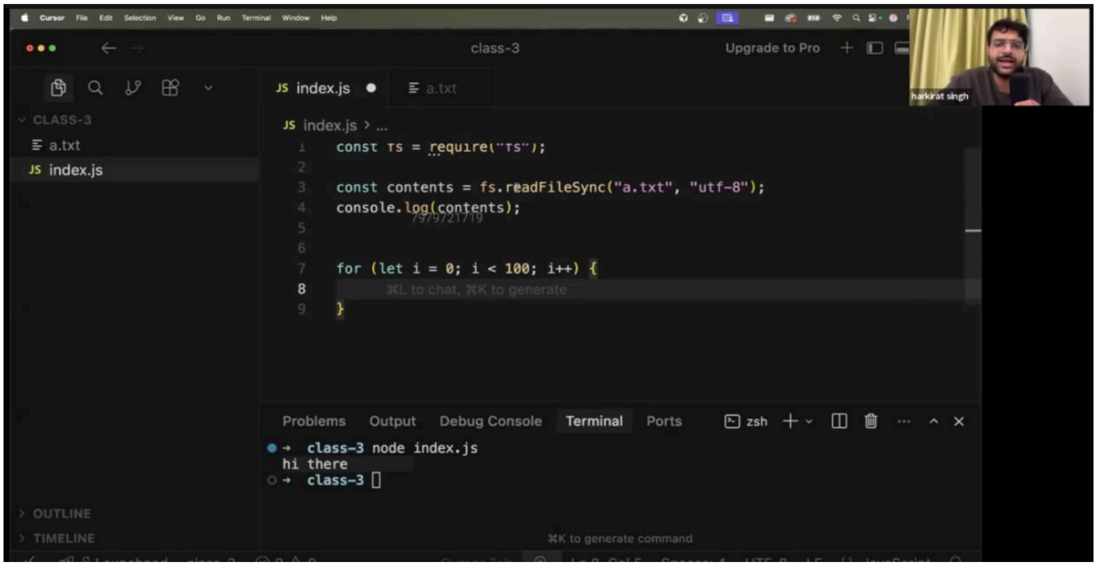
```
const fs = require("fs");

const contents = fs.readFileSync("a.txt", "utf-8");
console.log(contents);
```

4. Create another file (b.txt)
5. Write the code to read the other file `synchronously`

```
const fs = require("fs");

const contents = fs.readFileSync("a.txt", "utf-8");
console.log(contents);

const contents2 = fs.readFileSync("b.txt", "utf-8");
console.log(contents2);
```

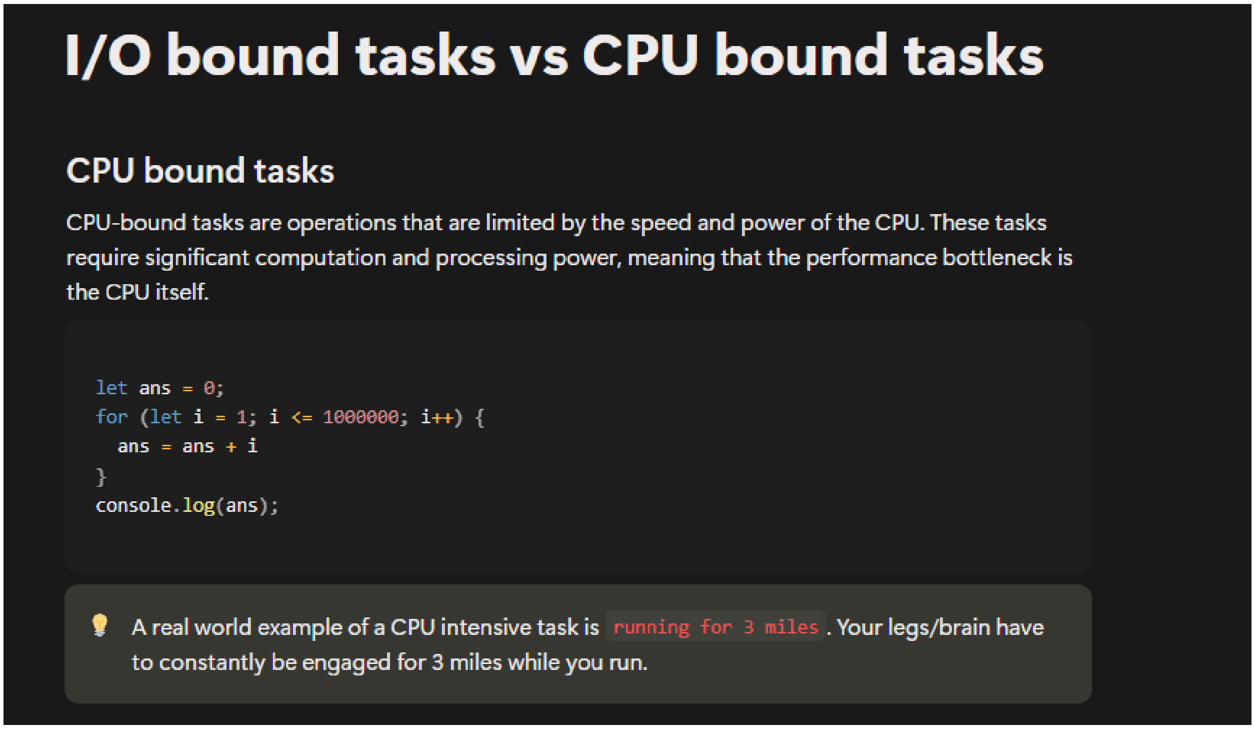
💡 What is wrong in this code above?

We are reading code synchronously it means we are waiting each time when we reading file because it depend on operating system so cpu heavy task also get delay like for loop



```
const fs = require("fs");

const contents = fs.readFileSync("a.txt", "utf-8");
console.log(contents);


for (let i = 0; i < 100; i++) {

}
```

```
→ class-3 node index.js
hi there
→ class-3
```



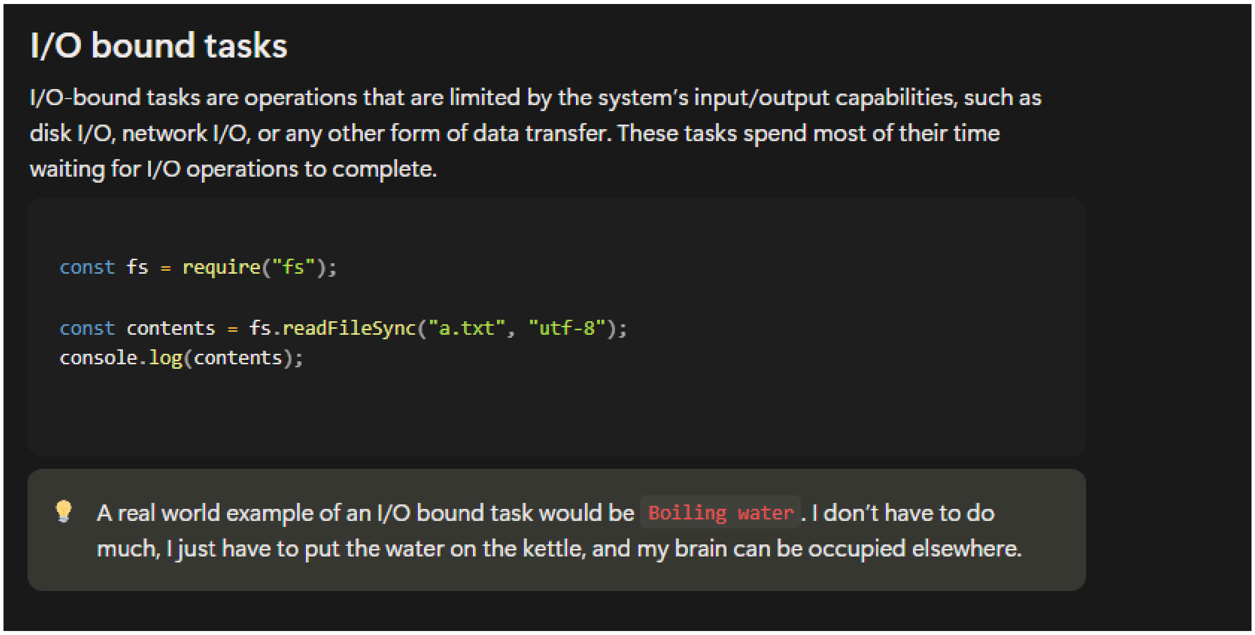
I/O bound tasks vs CPU bound tasks

CPU bound tasks

CPU-bound tasks are operations that are limited by the speed and power of the CPU. These tasks require significant computation and processing power, meaning that the performance bottleneck is the CPU itself.

```
let ans = 0;
for (let i = 1; i <= 1000000; i++) {
  ans = ans + i
}
console.log(ans);
```

💡 A real world example of a CPU intensive task is `running for 3 miles`. Your legs/brain have to constantly be engaged for 3 miles while you run.



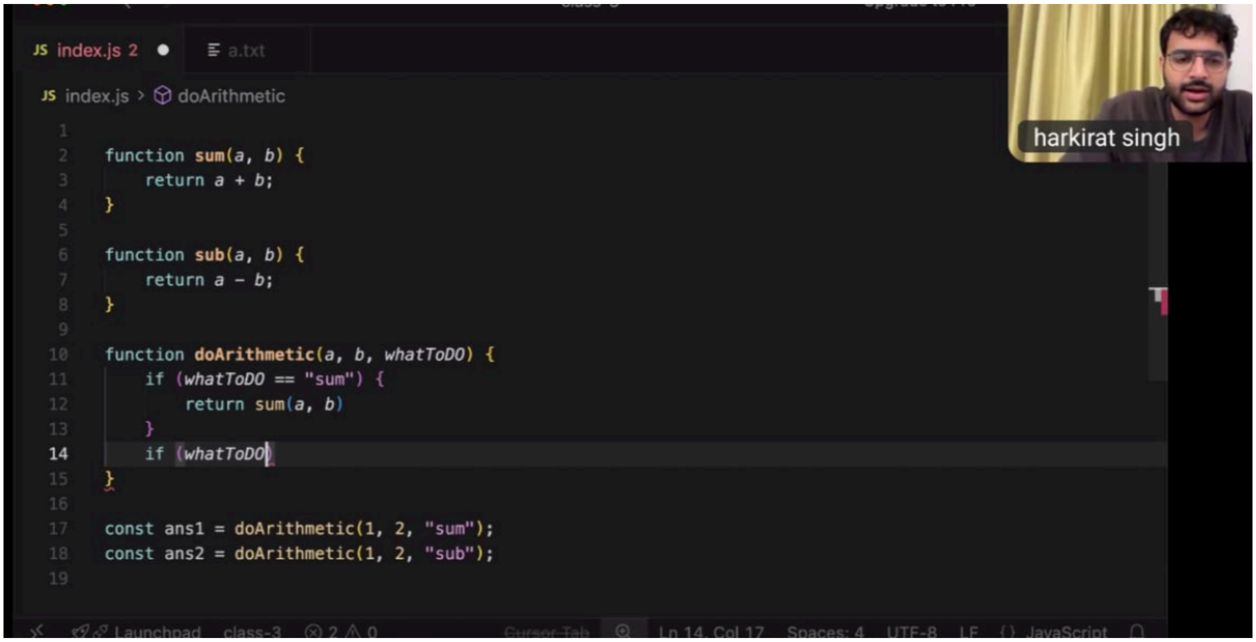
I/O bound tasks

I/O-bound tasks are operations that are limited by the system's input/output capabilities, such as disk I/O, network I/O, or any other form of data transfer. These tasks spend most of their time waiting for I/O operations to complete.

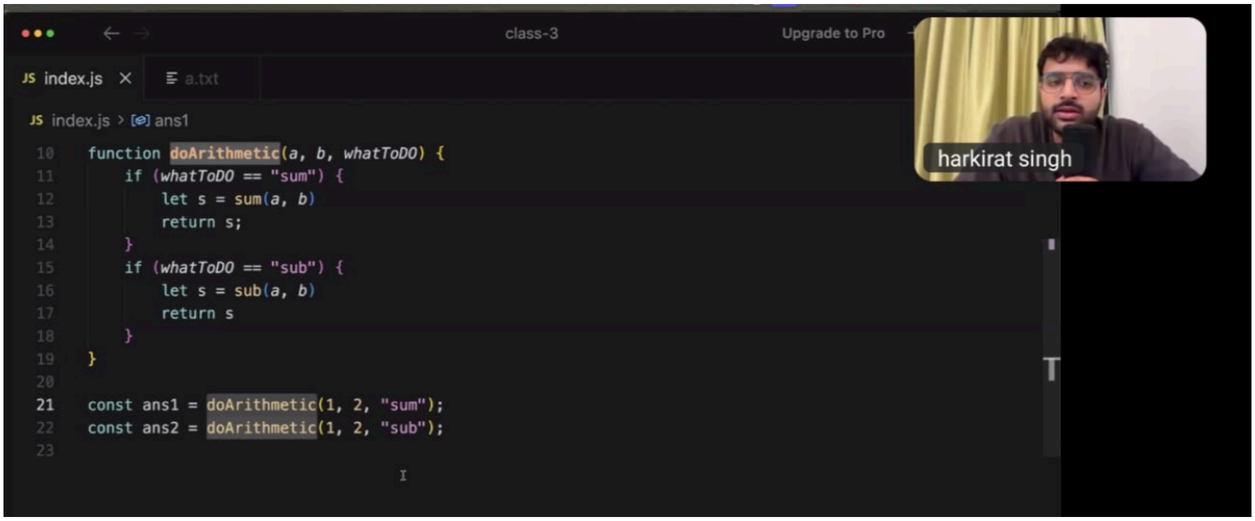
```
const fs = require("fs");

const contents = fs.readFileSync("a.txt", "utf-8");
console.log(contents);
```

> 💡 A real world example of an I/O bound task would be `Boiling water`. I don't have to do much, I just have to put the water on the kettle, and my brain can be occupied elsewhere.

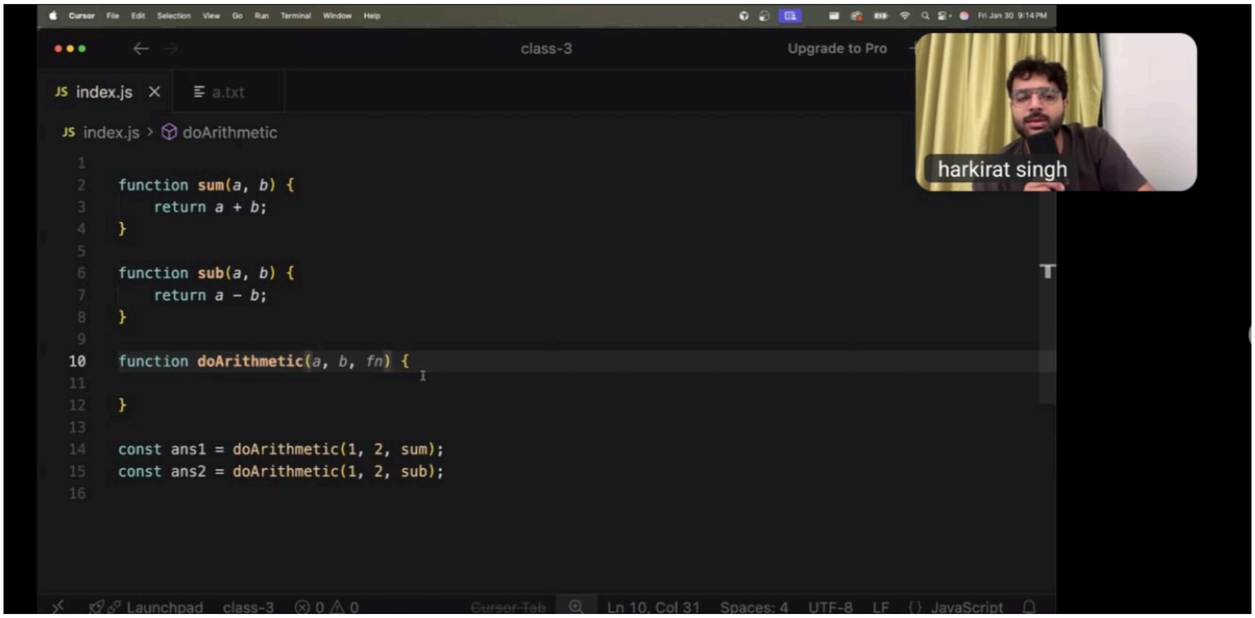
Functional arguments - https://petal-estimate-4e9.notion.site/Functional-arguments-2f17dfd107358023ad2efc88a2edfc3f



```
function sum(a, b) {
    return a + b;
}

function sub(a, b) {
    return a - b;
}

function doArithmetic(a, b, whatToDO) {
    if (whatToDO == "sum") {
        return sum(a, b)
    }
    if (whatToDO)
}

const ans1 = doArithmetic(1, 2, "sum");
const ans2 = doArithmetic(1, 2, "sub");
```

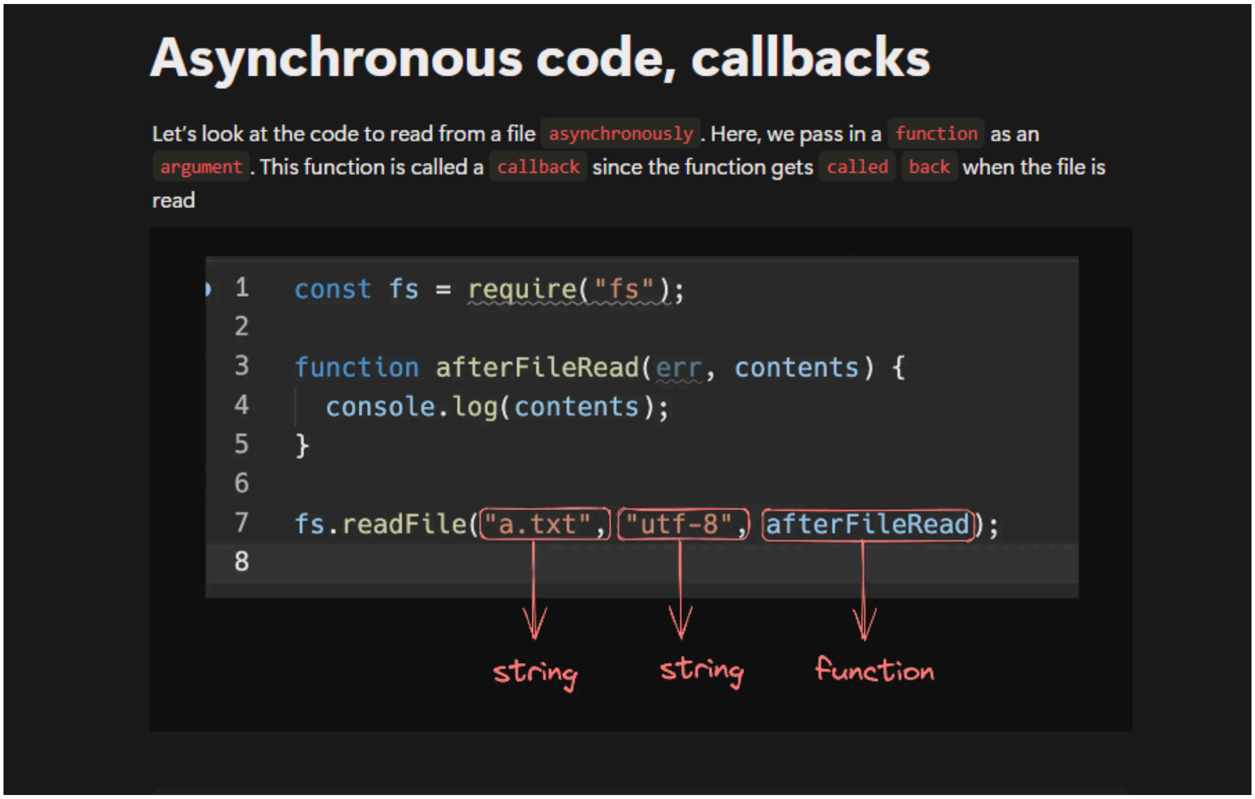


```
function doArithmetic(a, b, whatToDO) {
    if (whatToDO == "sum") {
        let s = sum(a, b)
        return s;
    }
    if (whatToDO == "sub") {
        let s = sub(a, b)
        return s
    }
}

const ans1 = doArithmetic(1, 2, "sum");
const ans2 = doArithmetic(1, 2, "sub");
```



class-3
Upgrade to Pro
index.js
a.txt
index.js > doArithmetic
harkirat singh
Launchpad class-3 ⊗ 0 ⚠ 0
Ln 10, Col 31 Spaces: 4 UTF-8 LF {} JavaScript

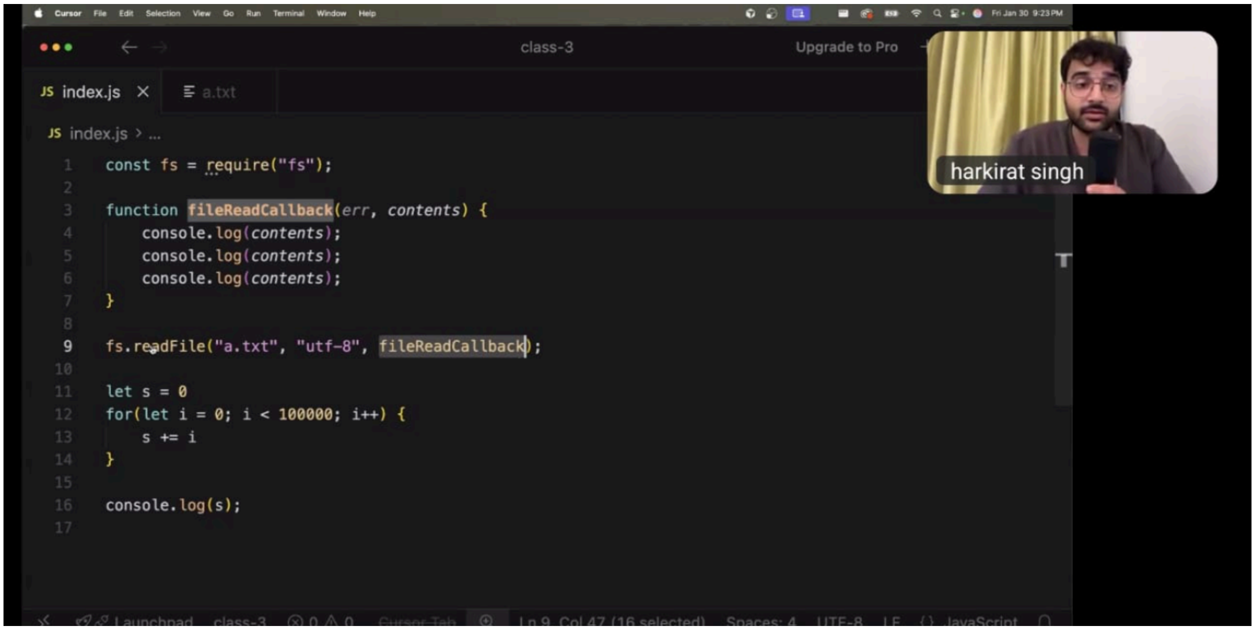
```
function sum(a, b) {
    return a + b;
}

function sub(a, b) {
    return a - b;
}

function doArithmetic(a, b, fn) {

}

const ans1 = doArithmetic(1, 2, sum);
const ans2 = doArithmetic(1, 2, sub);
```



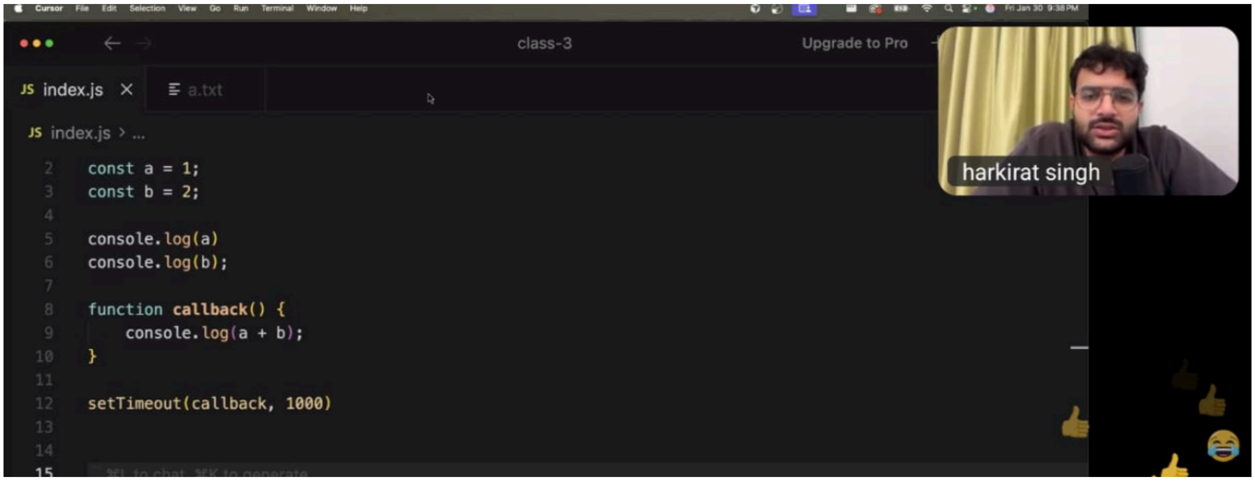
Asynchronous code, callbacks

Let's look at the code to read from a file `asynchronously`. Here, we pass in a `function` as an `argument`. This function is called a `callback` since the function gets `called` `back` when the file is read

```
const fs = require("fs");

function afterFileRead(err, contents) {
  console.log(contents);
}

fs.readFile("a.txt", "utf-8", afterFileRead);
```


class-3
Upgrade to Pro
index.js
a.txt
index.js > ...
harkirat singh
Launchpad class-3 ⊗ 0 ⚠ 0 Cursor Tab Ln 9, Col 47 (16 selected) Spaces: 4 UTF-8 LF {} JavaScript

```
const fs = require("fs");

function fileReadCallback(err, contents) {
    console.log(contents);
    console.log(contents);
    console.log(contents);
}

fs.readFile("a.txt", "utf-8", fileReadCallback);

let s = 0
for(let i = 0; i < 100000; i++) {
    s += i
}

console.log(s);
```



```
const a = 1;
const b = 2;

console.log(a)
console.log(b);

function callback() {
    console.log(a + b);
}

setTimeout(callback, 1000)
```

JS Architecture for async code

How JS executes asynchronous code - http://latentflip.com/loupe/