Async JS

1. I/O tasks
2. Callbacks
3. Functional arguments
4. Async vs Sync code
5. Event loops, callback queues, JS

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

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

1. Open vscode
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);
```

1. Create another file (b.txt)
2. 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?

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.

Doing I/O bound tasks in the real world



What if you were tasked with doing 3 things

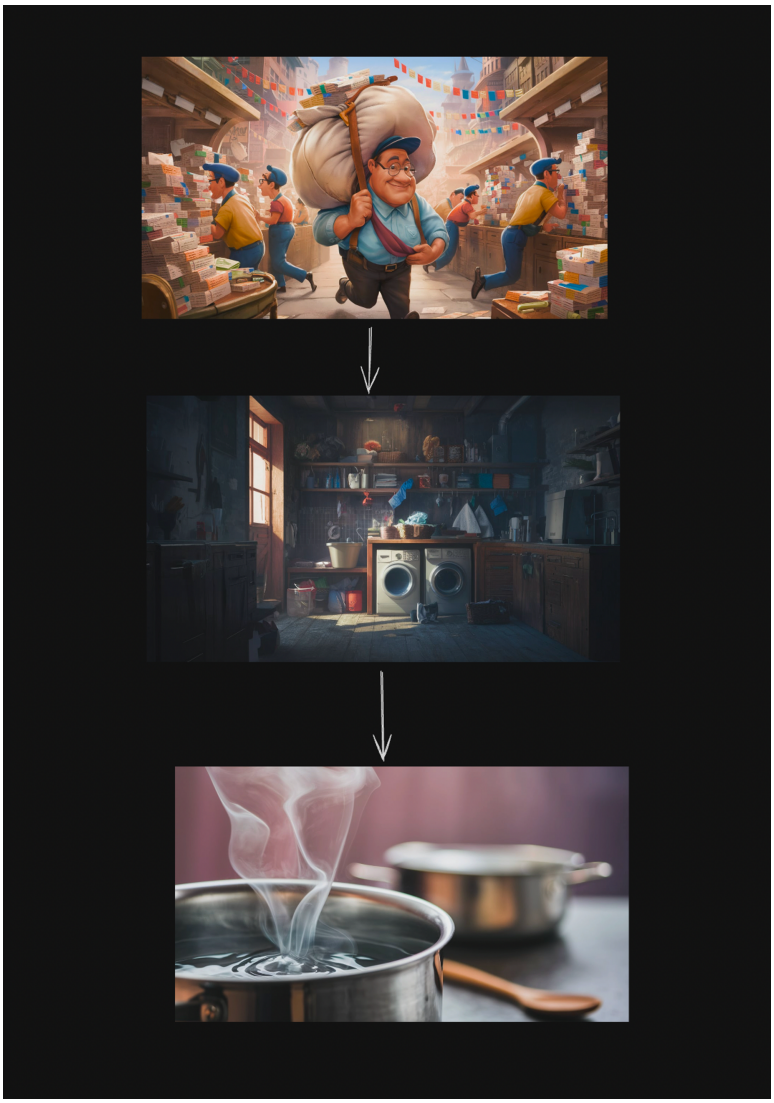
1. Boil some water.
2. Do some laundry
3. Send a package via mail

Would you do these

1. One by one (synchronously)
2. Context switch between them (Concurrently)
3. Start all 3 tasks together, and wait for them to finish. The first one that finishes gets catered to first.

Synchronously (One by one)

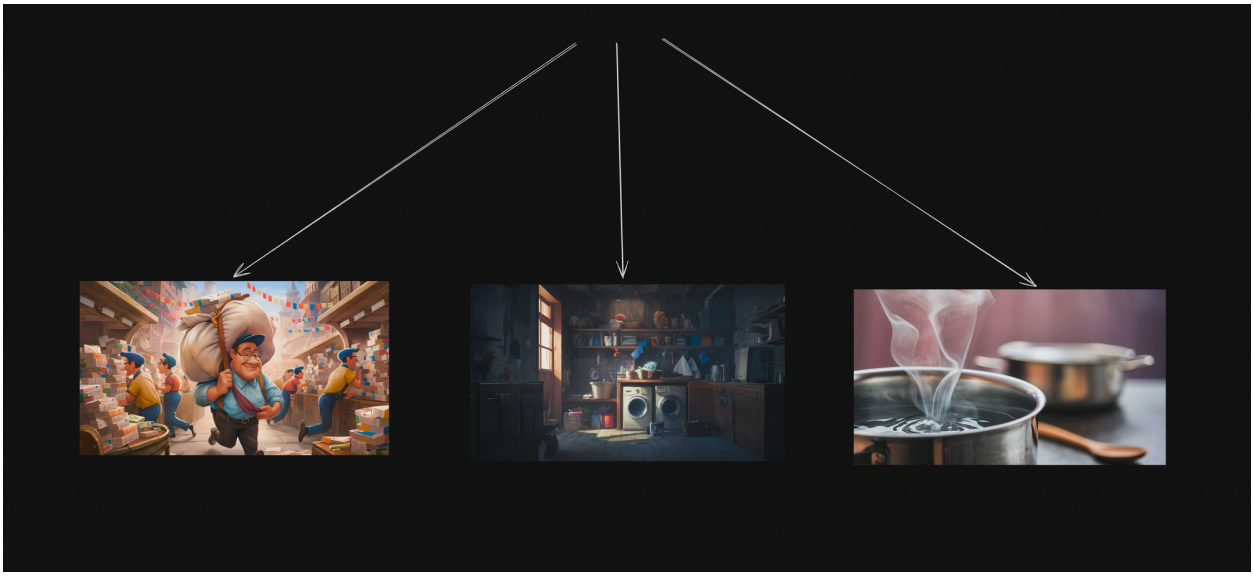
```
const fs = require("fs");

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

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

const contents3 = fs.readFileSync("b.txt", "utf-8");
console.log(contents3);
```



Start all 3 tasks together, and wait for them to finish.

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

fs.readFile("a.txt", "utf-8", function (err, contents) {
  console.log(contents);
});

fs.readFile("b.txt", "utf-8", function (err, contents) {
  console.log(contents);
```

```
});

fs.readFile("a.txt", "utf-8", function (err, contents) {
  console.log(contents);
});
```



Functional arguments

Write a `calculator` program that adds, subtracts, multiplies, divides two arguments.

Approach #1

Calling the respective function

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

function multiply(a, b) {
  return a * b;
}

function subtract(a, b) {
```

```
  return a - b;
}

function divide(a, b) {
  return a / b;
}

function doOperation(a, b, op) {
  return op(a, b)
}

console.log(sum(1, 2))
```

Approach #2

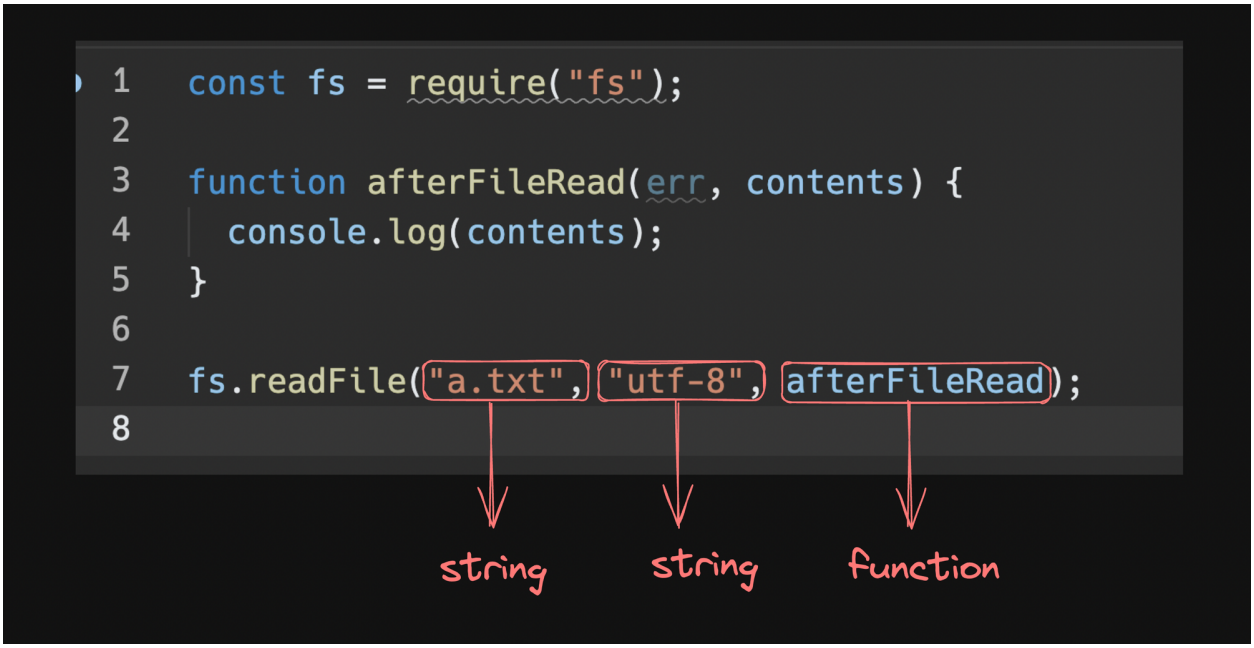
Passing in what needs to be done as an argument.

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

function multiply(a, b) {
  return a * b;
}

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

function divide(a, b) {
  return a / b;
}

function doOperation(a, b, op) {
  return op(a, b)
}
```

```
console.log(doOperation(1, 2, sum))
```

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");

fs.readFile("a.txt", "utf-8", function (err, contents) {
  console.log(contents);
});
```

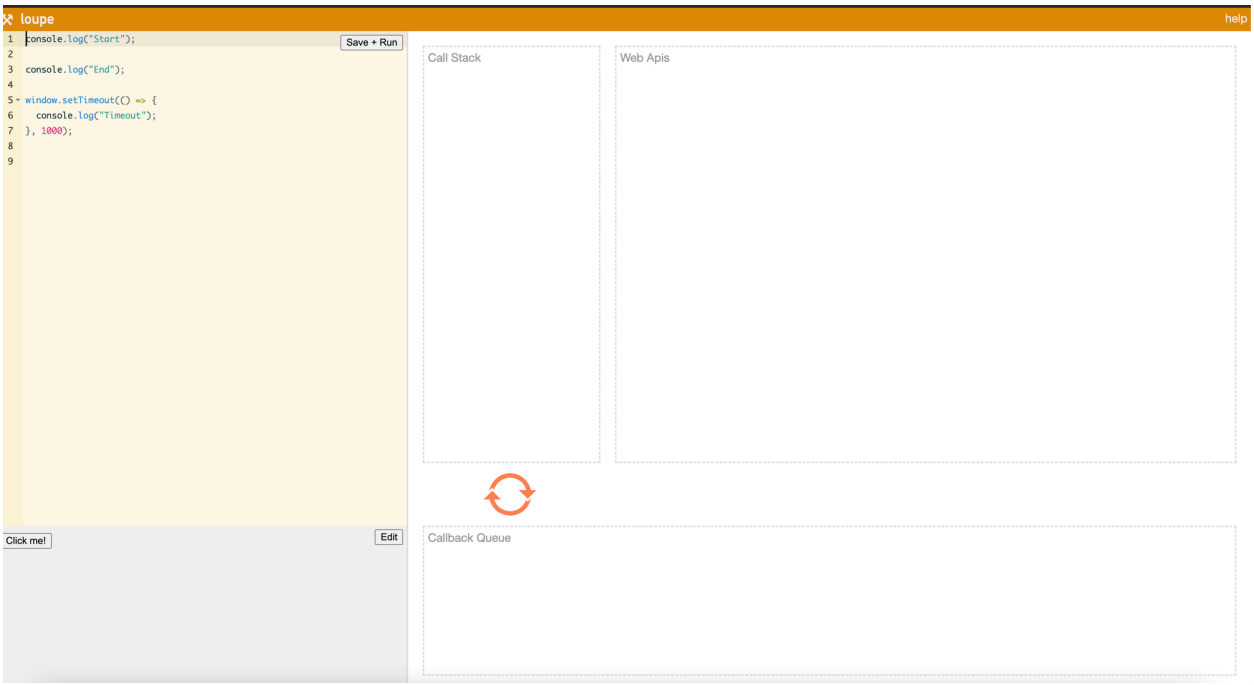
setTimeout

setTimeout is another asynchronous function that executes a certain code after some time

```
function run() {
    console.log("I will run after 1s");
}

setTimeout(run, 1000);
console.log("I will run immedietely");
```

JS Architecture for async code

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



1. Call Stack

- The call stack is a data structure that keeps track of the function calls in your program. It operates in a "Last In, First Out" (LIFO) manner, meaning the last function that was called is the first one to be executed and removed from the stack.
- When a function is called, it gets pushed onto the call stack. When the function completes, it's popped off the stack.

Code

```
function first() {
  console.log("First");
}
function second() {
  first();
  console.log("Second");
}
second();
```

2. Web APIs

- Web APIs are provided by the browser (or the Node.js runtime) and allow you to perform tasks that are outside the scope of the JavaScript language itself, such as making network requests, setting timers, or handling DOM events.

3. Callback Queue

The callback queue is a list of tasks (callbacks) that are waiting to be executed once the call stack is empty. These tasks are added to the queue by Web APIs after they have completed their operation.

4. Event loop

The event loop constantly checks if the call stack is empty. If it is, and there are callbacks in the callback queue, it will push the first callback from the queue onto the call stack for execution.