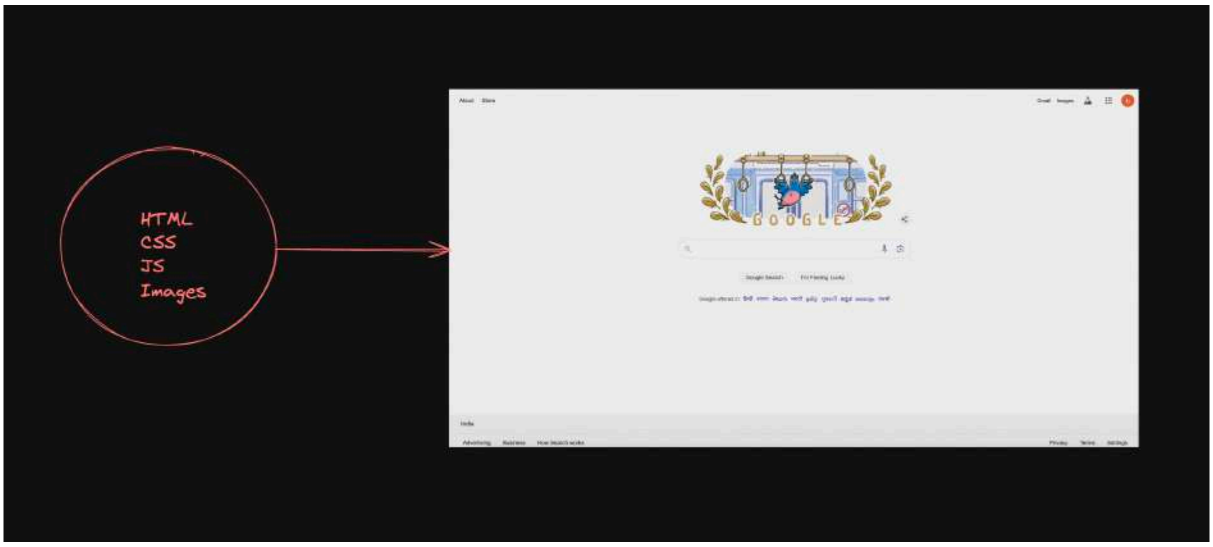
Javascript - The basics

Web development

Web development involves writing a lot of HTML, CSS and JS code.

Historically (and even today to some extend), browsers could only understand HTML, CSS and JS

Any website that you see, is a bunch of HTML, CSS and JS files along with some assets (images, videos etc)



Facts/Callouts

1. React, NextJS are `frameworks` . They compile down to HTML, CSS, JS in the end. That is what your browser understands.
2. When you run your C++ code on `leetcode` , it does not run on your browser/machine. It runs somewhere else. Your browser can't (almost) compile and run C++ code.
3. If someone asks — What all languages can your browser interpret, the answer is HTML, CSS, JS and WebAssembly. It can, technically, run C++/Rust code that is compiled down to Wasm

Before we proceed, do one of the following -

1. Create an account on replit
2. Install Node.js locally
3. Keep your browser console open for testing locally

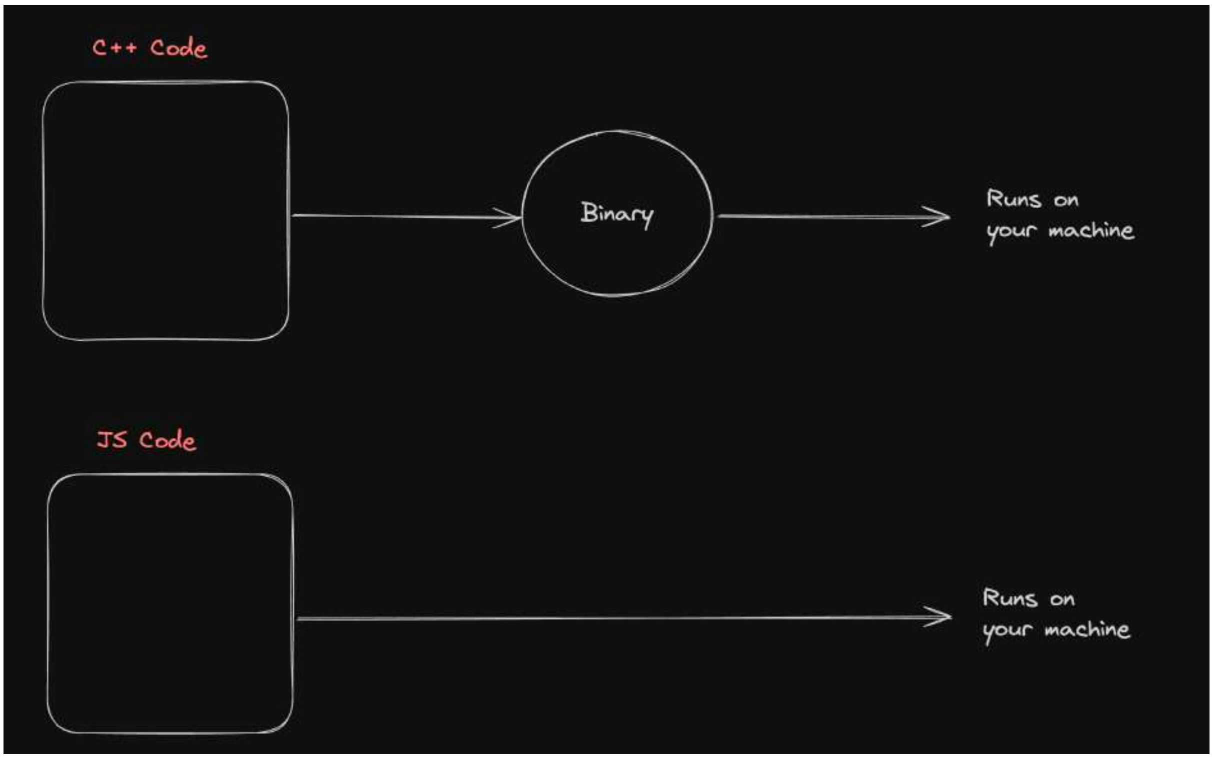
Properties of JS

Every language comes with it's unique set of features.

Javascript has the following -

1. Interpreted

JavaScript is an interpreted language, meaning it's executed line-by-line at runtime by the JavaScript engine in the browser or server environment, rather than being compiled into machine code beforehand.



Upsides -

1. There is one less step to do before running your code

Downsides -

1. Performance Overhead:
2. More prone to runtime errors

2. Dynamically Typed

Variables in JavaScript are not bound to a specific data type. Types are determined at runtime and can change as the program executes

C++ Code (won't compile)

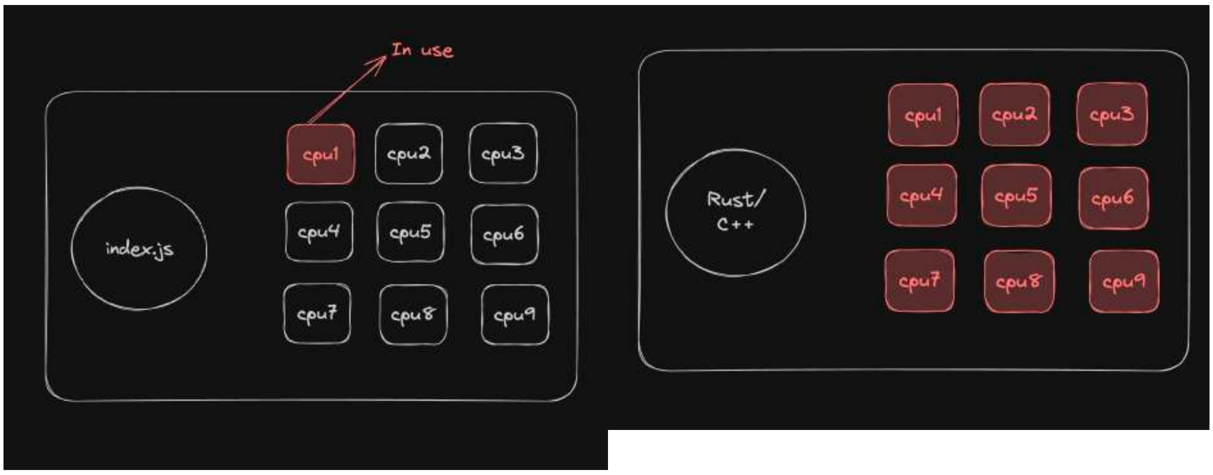
```
#include <iostream>

int main() {
  int a = 1;
  a = "hello";
  a = true;
}
```

JS Code (will compile)

```


var a = 1;
a = "harkirat";
a = true;

console.log(a)
```

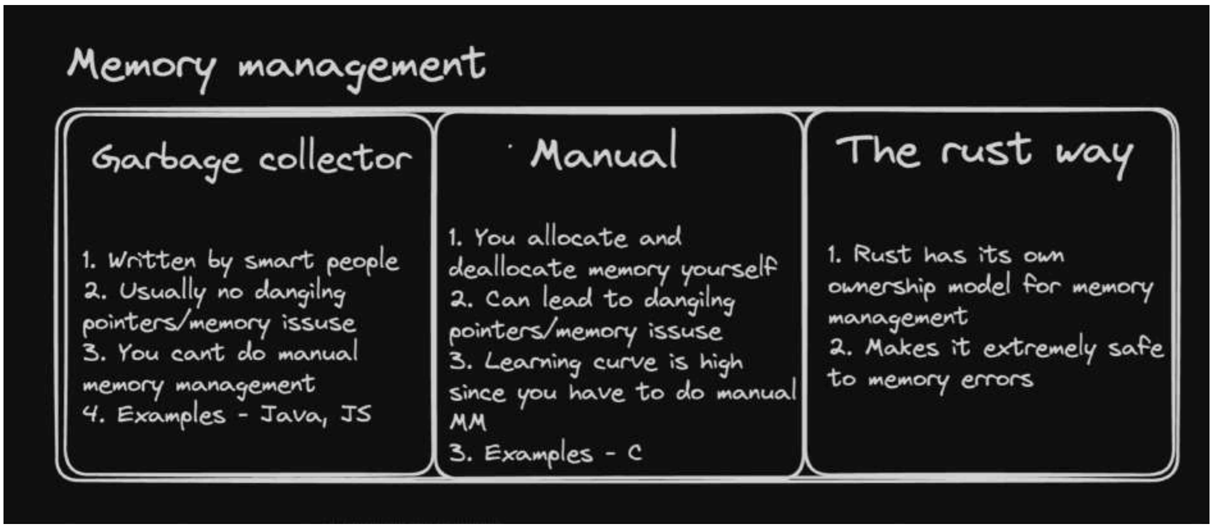
3. Single threaded

JavaScript executes code in a single-threaded environment, meaning it processes one task at a time. We will dive deeper into this next week.



4. Garbage collected

JavaScript automatically manages memory allocation and deallocation through garbage collection, which helps prevent memory leaks by automatically reclaiming memory used by objects no longer in use.



Memory management

Garbage collector	Manual	The rust way
1. Written by smart people 2. Usually no dangilng pointers/memory issuse 3. You cant do manual memory management 4. Examples - Java, JS	1. You allocate and deallocate memory yourself 2. Can lead to dangilng pointers/memory issuse 3. Learning curve is high since you have to do manual MM 3. Examples - C	1. Rust has its own ownership model for memory management 2. Makes it extremely safe to memory errors

Conclusion

Is JS a good language?

Yes and no. It is beginner friendly, but has a lot of performance overhead. **Bun** is trying to solve for a lot of this, but there’s a long way to go before JS can compete with languages like C++/Rust

Syntax of Javascript

1. Variables

Variables are used to store data. In JavaScript, you declare variables using `var`, `let`, or `const`.

```
let name = "John";     // Variable that can be reassigned
const age = 30;        // Constant variable that cannot be reassigned
var isStudent = true;  // Older way to declare variables, function-scoped
```

▼ Assignment

Create a variable for each of the following: your favorite color, your height in centimeters, and whether you like pizza. Use appropriate variable declarations (`let`, `const`, or `var`). Try logging it using `console.log`

2. Data types

```
let number = 42;            // Number
let string = "Hello World";  // String
let isActive = false;        // Boolean
let numbers = [1, 2, 3];     // Array
```

3. Operators

```
let sum = 10 + 5;          // Arithmetic operator
let isEqual = (10 === 10); // Comparison operator
let isTrue = (true && false); // Logical operator
```

4. Functions

```
// Function declaration
function greet(name) {
  return "Hello, " + name;
}
```

```
// Function call
let message = greet("John"); // "Hello, John"
```

▼ Assignment #1

Write a function `sum` that finds the sum of two numbers.
Side quest - Try passing in a string instead of a number and see what happens?

▼ Assignment #2

Write a function called `canVote` that returns true or false if the `age` of a user is > 18

5. If/Else

```
if (age >= 18) {
   console.log("You are an adult.");
} else {
   console.log("You are a minor.");
}
```

▼ Assignment

Write an if/else statement that checks if a number is even or odd. If it's even, print "The number is even." Otherwise, print "The number is odd."

6. Loops

```
// For loop
for (let i = 0; i < 5; i++) {
   console.log(i); // Outputs 0 to 4
}

// While loop
let j = 0;
while (j < 5) {
   console.log(j); // Outputs 0 to 4
   j++;
}
```

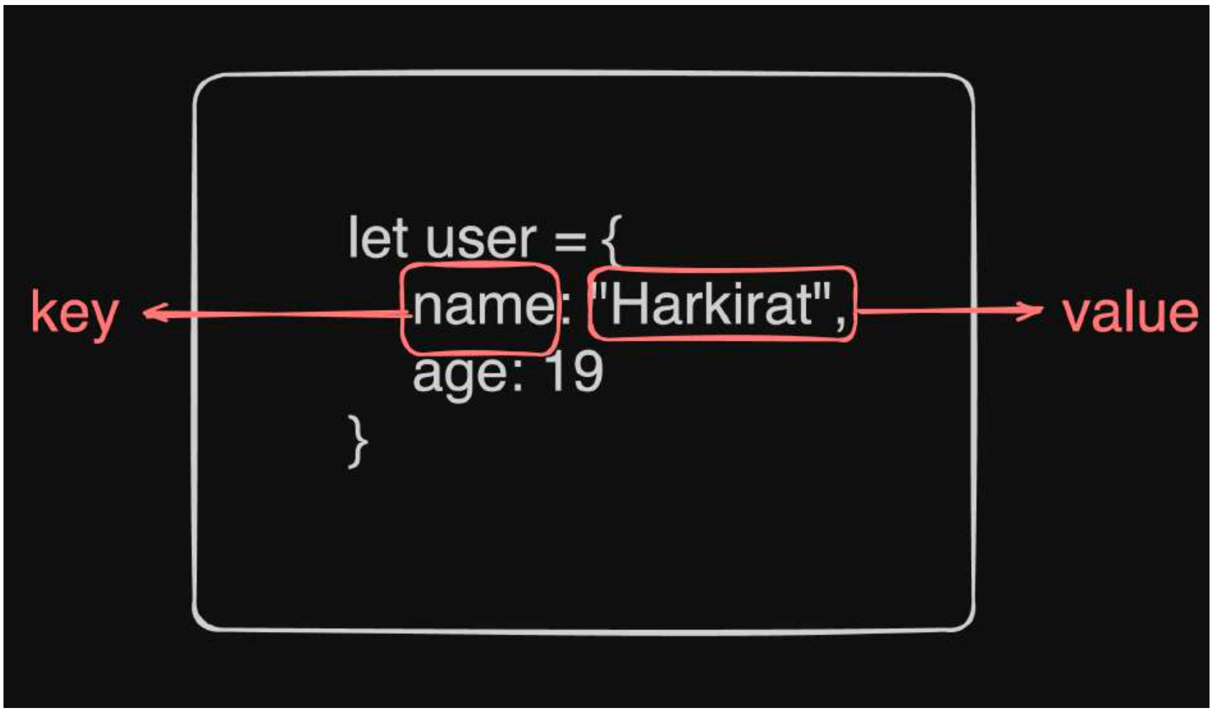
▼ Assignment

Write a function called sum that finds the `sum` from 1 to a number

Complex types

Objects

An object in JavaScript is a collection of `key-value pairs`, where each `key` is a string and each `value` can be any valid JavaScript data type, including another object.



```
let user = {
    name: "Harkirat",
    age: 19
}

console.log("Harkirats age is " + user.age);
```

▼ Assignment #1

Write a function that takes a `user` as an input and greets them with their name and age

▼ Assignment #2

Write a function that takes a new object as input which has `name` , `age` and `gender` and greets the user with their gender (Hi `Mr/Mrs/Others` harkirat, your age is 21)

▼ Assignment #3

Also tell the user if they are legal to vote or not

Arrays

Arrays let you group data together

```
const users = ["harkirat", "raman", "diljeet"];
const tatalUsers = users.length;
const firstUser = users[0];
```

▼ Assignment

Write a function that takes an array of numbers as input, and returns a new array with only even values. Read about `filter` in JS

Array of Objects

We can have more complex objects, for example an array of objects

```
const users = [{
		name: "Harkirat",
		age: 21
	}, {
		name: "raman",
		age: 22
	}
}

const user1 = users[0]
const user1Age = users[0].age
```

▼ Assignment

Write a function that takes an array of users as inputs and returns only the users who are more than 18 years old

Object of Objects

We can have an even more complex object (object of objects)

```
const user1 = {
    name: "harkirat",
    age: 19,
    address: {
        city: "Delhi",
        country: "India",
        address: "1122 DLF"
    }
}

const city = user1.address.city;
```

▼ Assignment

Create a function that takes an array of objects as input, and returns the users whose age > 18 and are male

Introduction

Shoutouts

Web based wallets ($50 each)

https://x.com/kairveee/status/18222632870791742 https://x.com/DweetParikh/status/182224945666

Post

Kairvee
@kairveee

completed the week-2 assignment of web-3 cohort
live- kairveeehh.github.io/web-3-a1/
repo - github.com/kairveeehh/web...

@kirat_tw @100xDevs

https://github.com/kairveeehh/web-3-a1

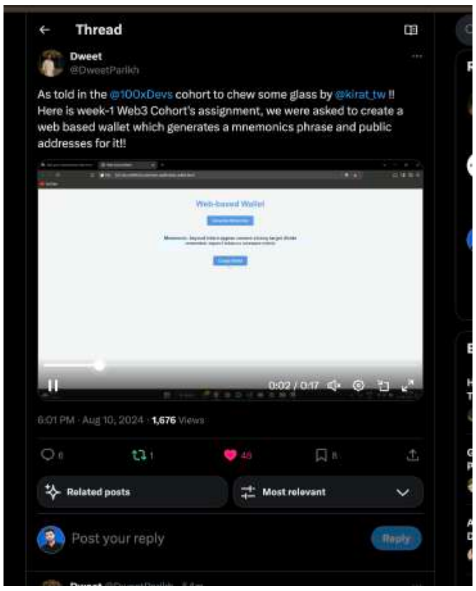
GitHub
GitHub - kairveeehh/web-3-a1
Contribute to kairveeehh/web-3-a1 development by creating an account on GitHub.

kairveeehh/**web-3-a1**

@DRAKe https://github.com/kairveeehh/web-3-a1
TA Harry Potter Today at 18:38
yeah looks correct man.
nice work

6:56 PM · Aug 10, 2024 · **16** Views

Related posts


Thread

Dweet
@DweetParikh

As told in the @100xDevs cohort to chew some glass by @kirat_tw !!
Here is week-1 Web3 Cohort's assignment, we were asked to create a web based wallet which generates a mnemonics phrase and public addresses for it!!

6:01 PM · Aug 10, 2024 · **1,676** Views

- Reports Platform ($100 for creator) - https://report-100xdevs.vercel.app/

Goal of todays class -

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

Goal of tomorrows class

1. Async await, Promises
2. Practising async JS

Hopefully, by the end of the class, you are able to understand the following code -

▼ Functional arguments

```
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))
```

▼ Callbacks

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

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

Normal functions in JS

The way to write functions in JS is as follows -

Find sum of two numbers

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

let ans = sum(2, 3)
console.log(sum);
```

Find sum from 1 to a number

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

const ans = sum(100);
console.log(ans);
```

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

```
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

💡 We're going to introduce imports/requires next. A `require` statement lets you import code/functions export from another file/module.

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?

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

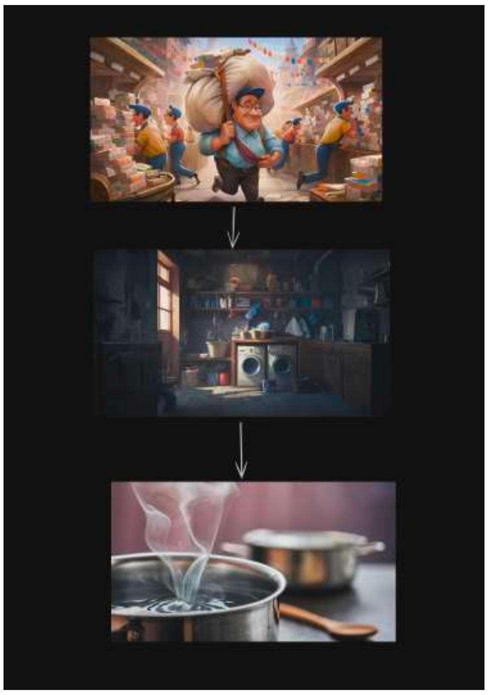
1. One by one (synchronously)
2. Context switch between them (Concurrently)
3. Start all 3 tasks together, and wait for them to finish. The first one that finishes gets catered to first.

💡 Good talk - Concurrency is not parallelism - https://www.youtube.com/watch?v=oV9rvDllKEg

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))
```

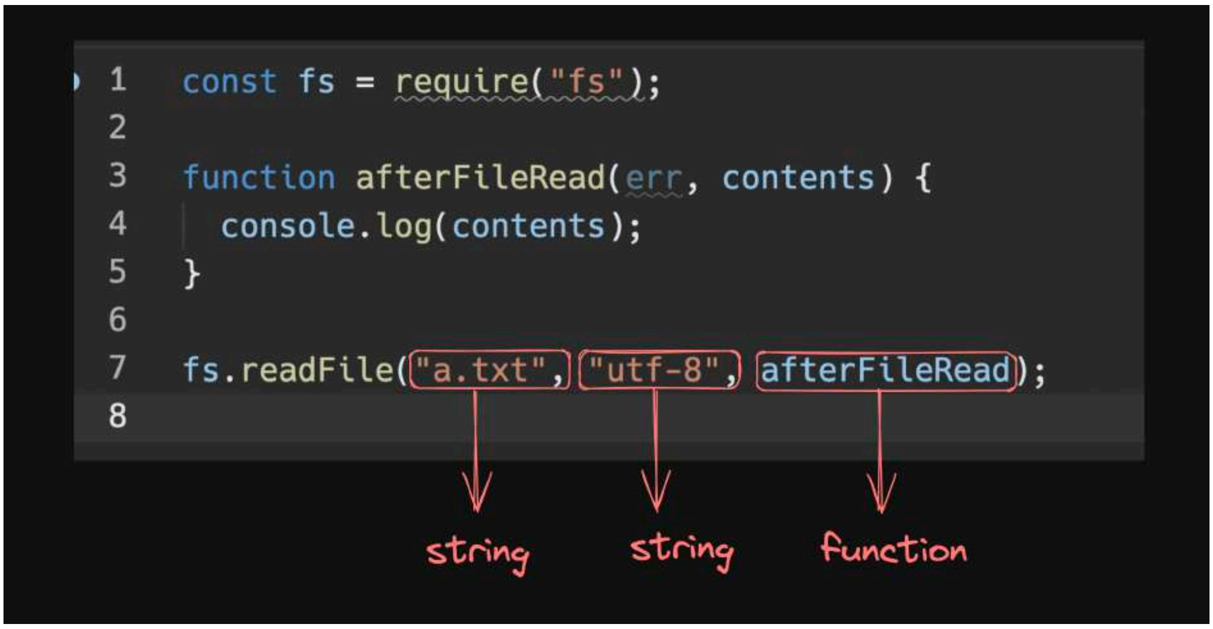
Approach #2

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");

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

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

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

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

setTimeout

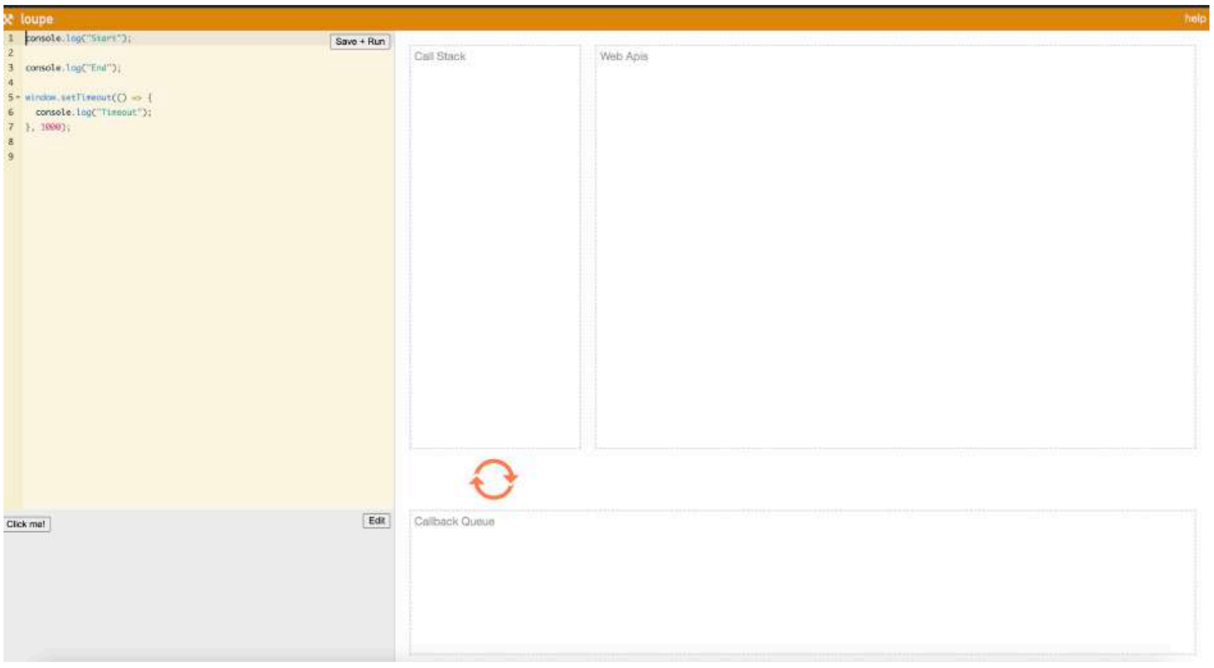
setTimeout is another asynchronous function that executes a certain code after some time

```
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/



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

console.log("End");

window.setTimeout(() => {
  console.log("Timeout");
}, 1000);
```

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.