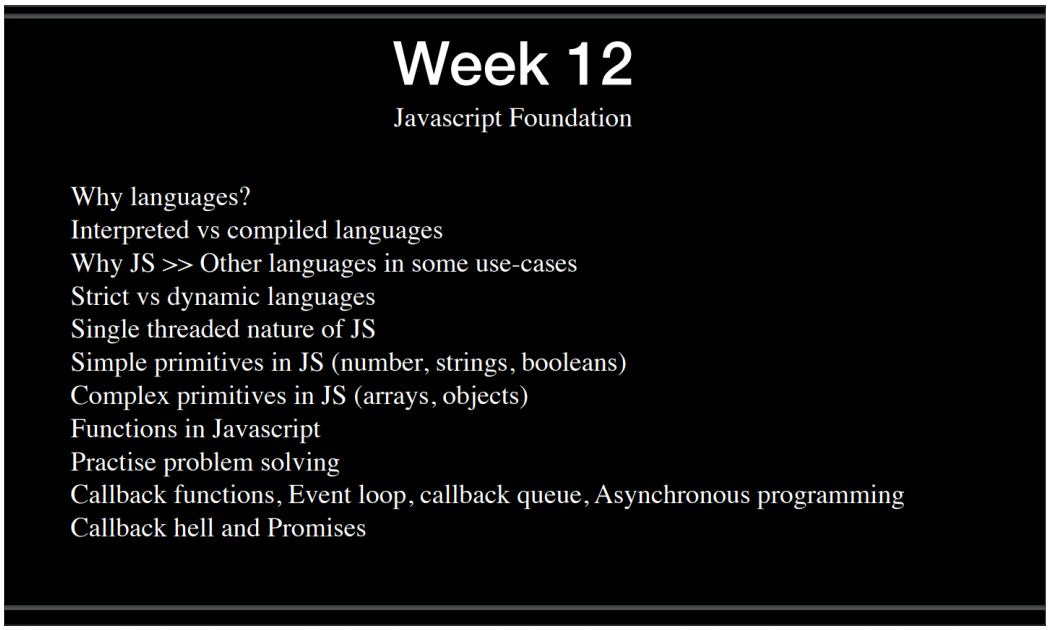
## Week 1.1:

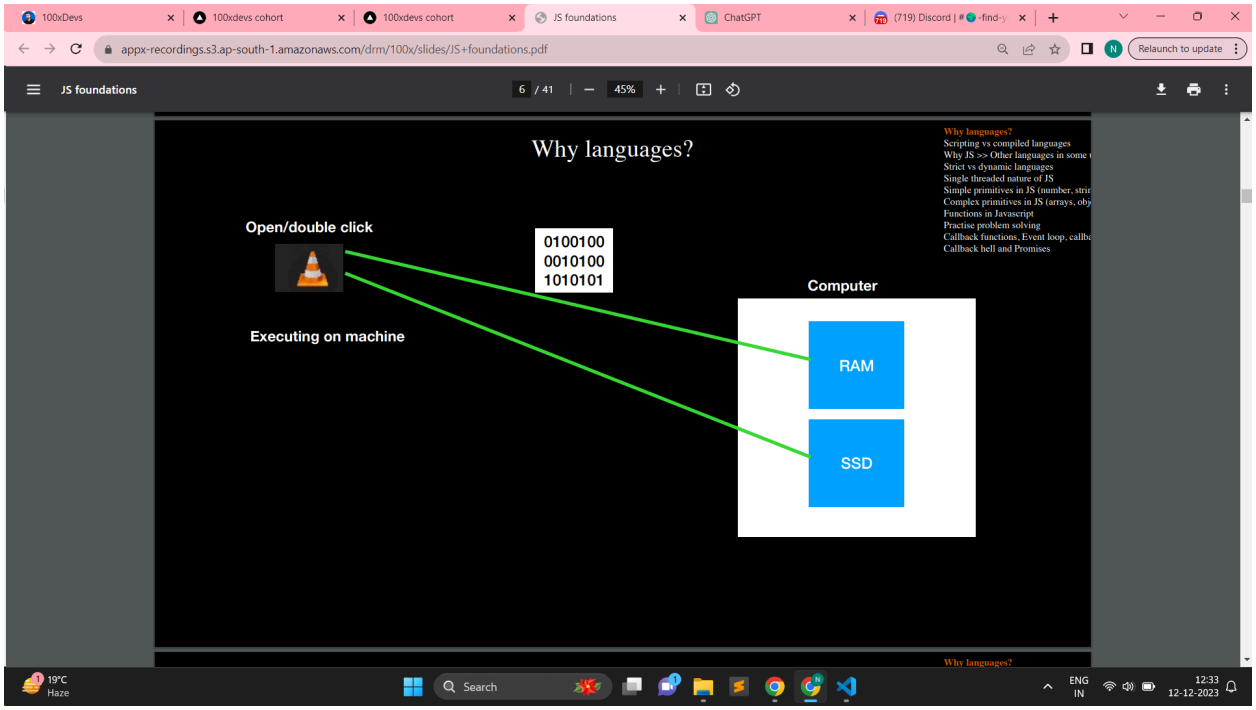
- Make sure your open-source contribution looks good (UI)
- If you are designing frontend make sure you go the extra mile on design
- Abstraction, using typescript, etc are a sign, files are in the right places,es-lint of good developers, writing test, indentation is correct
- dumb.sh, cal. sh

## Week 1.2: JS Foundation

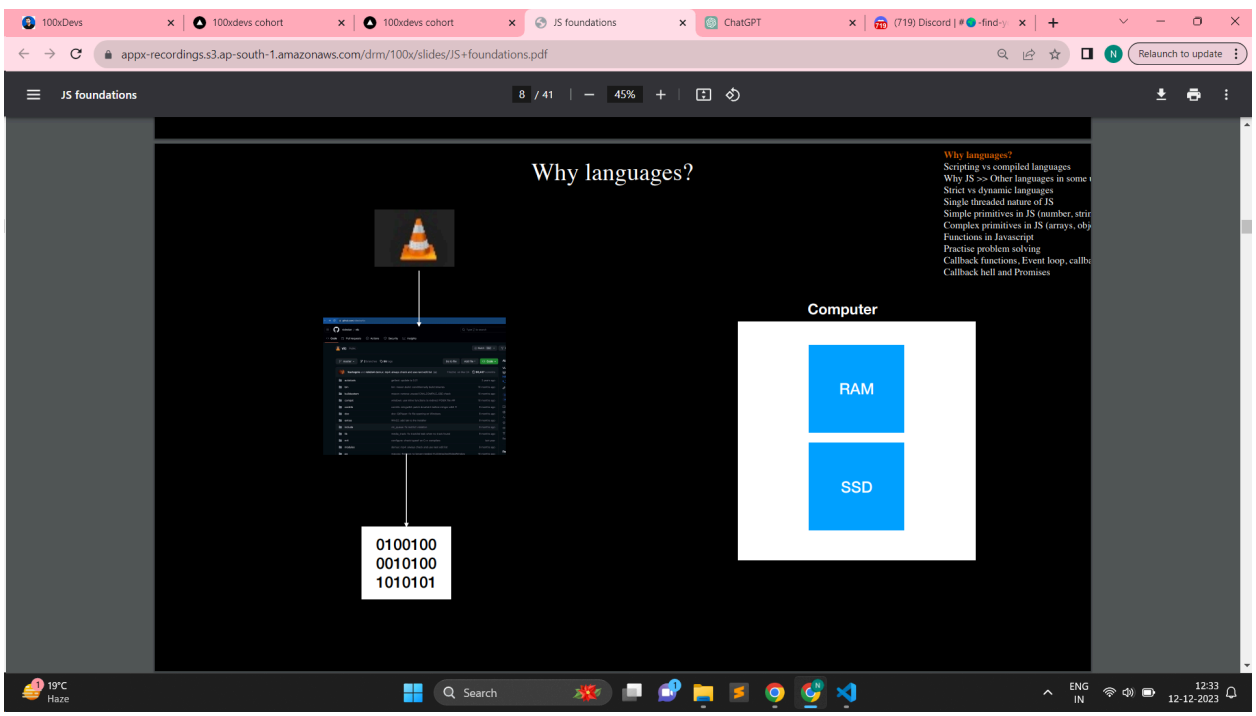



### Why languages?

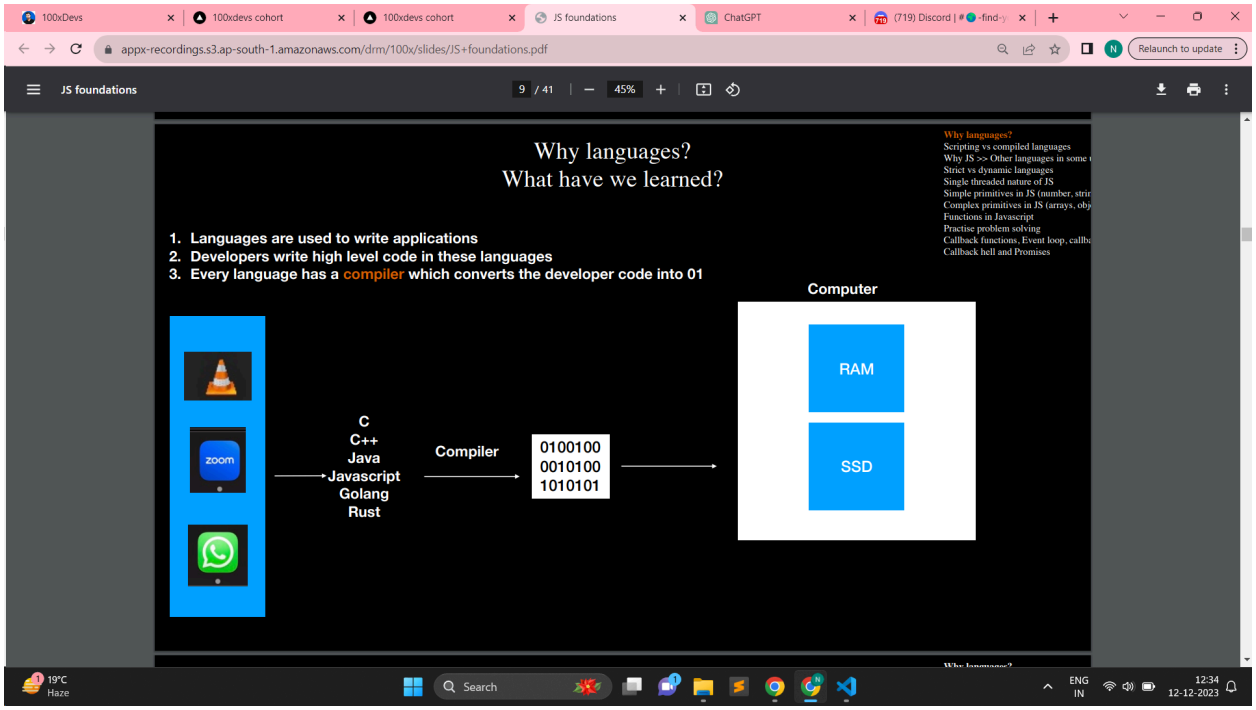
The computer has RAM and SSD, whenever we have an application locally, it resides on the SSD(hard disk drive) and when we run something, it runs on the RAM, currently running things reside inside the RAM, when we double click on the app then it goes to RAM




What exactly goes to RAM? Basically 0, 1




How does the C code in which the VLC media is written get converted into 0,1



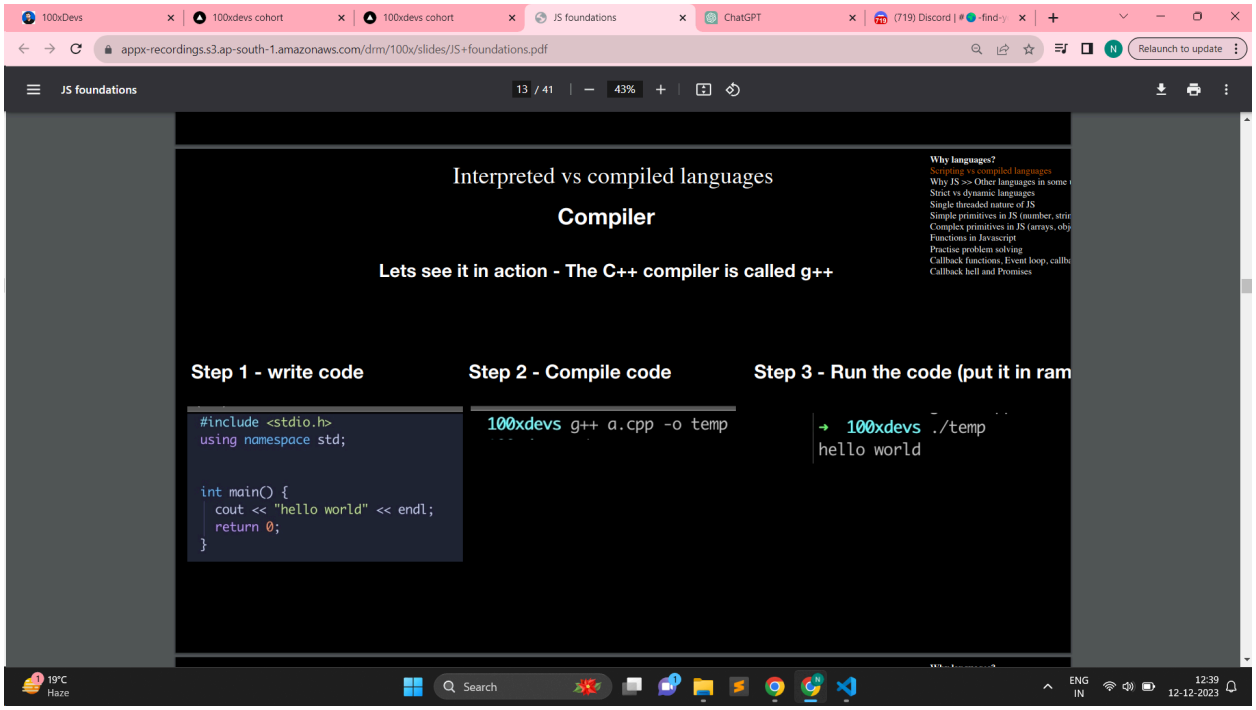

With the help of a compiler(each language has a compiler).

High level code to 0,1

Compiler convert high level developer friendly code into 0s and 1s

## Interpreted Vs Compiled Languages

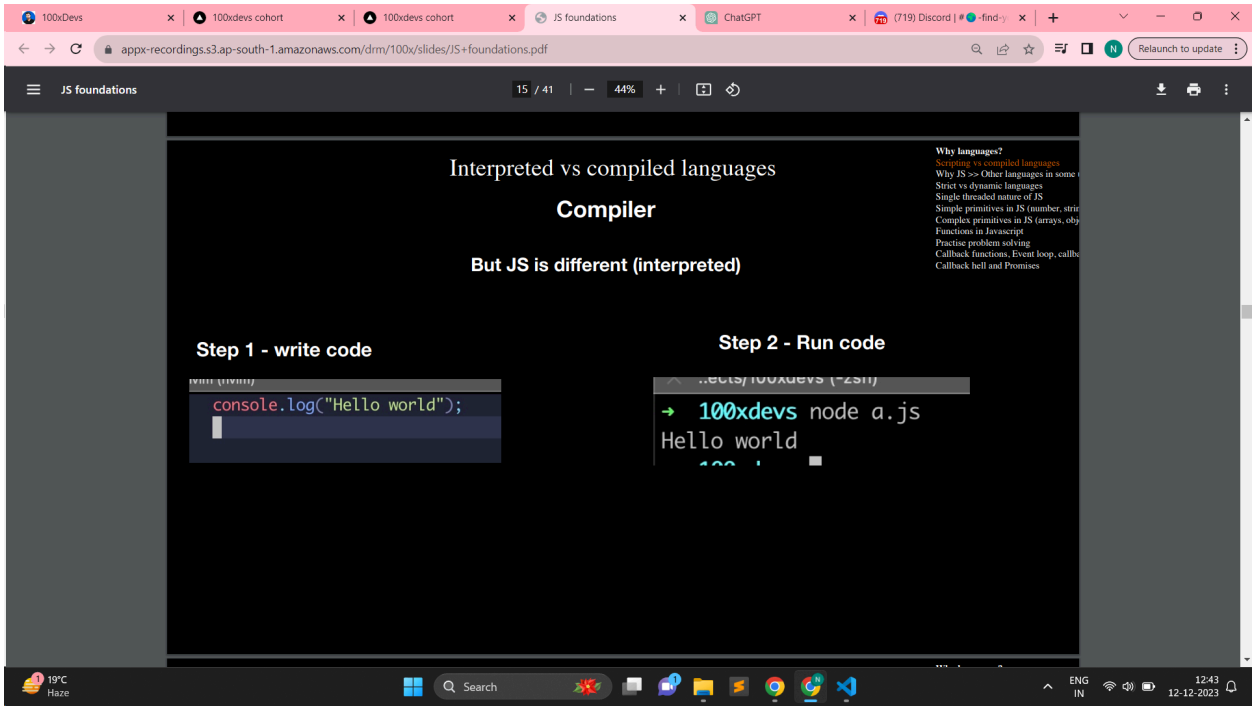
### C++ is compiled language




Here the a.cpp file is converted(compiled) into binary code and saved in a file called temp

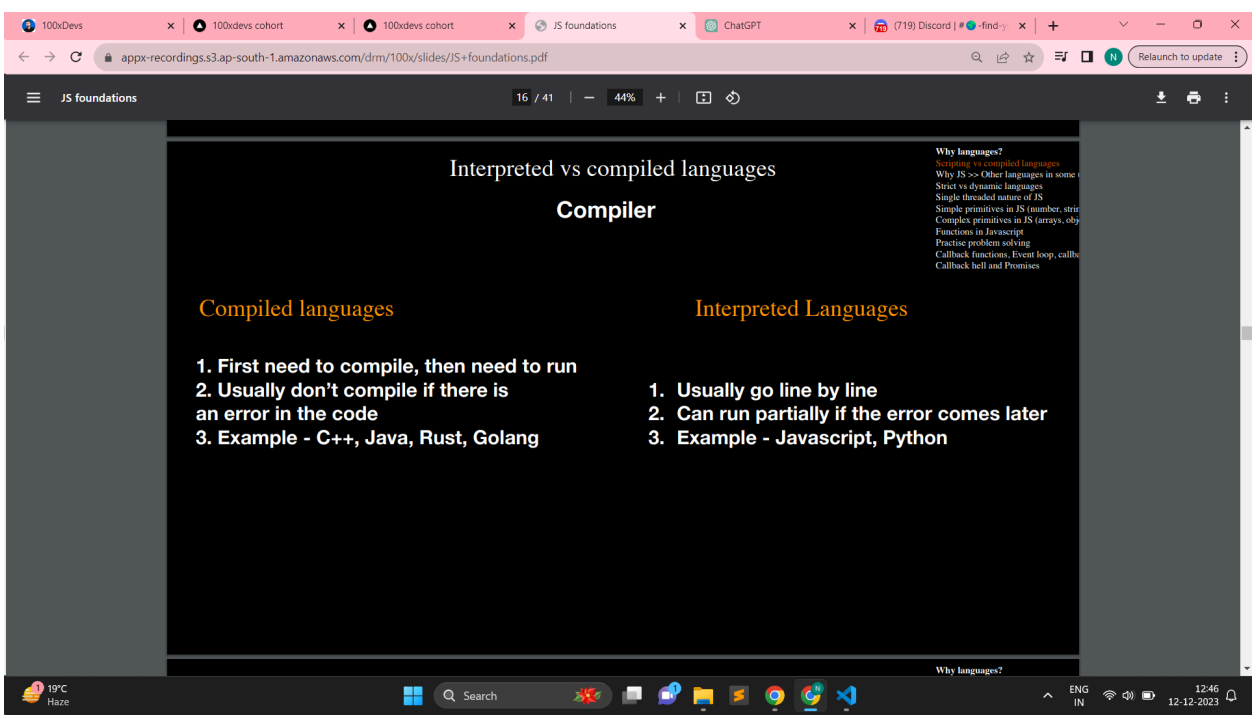
High level language to Binary code then it will run

## Javascript is interpreted language

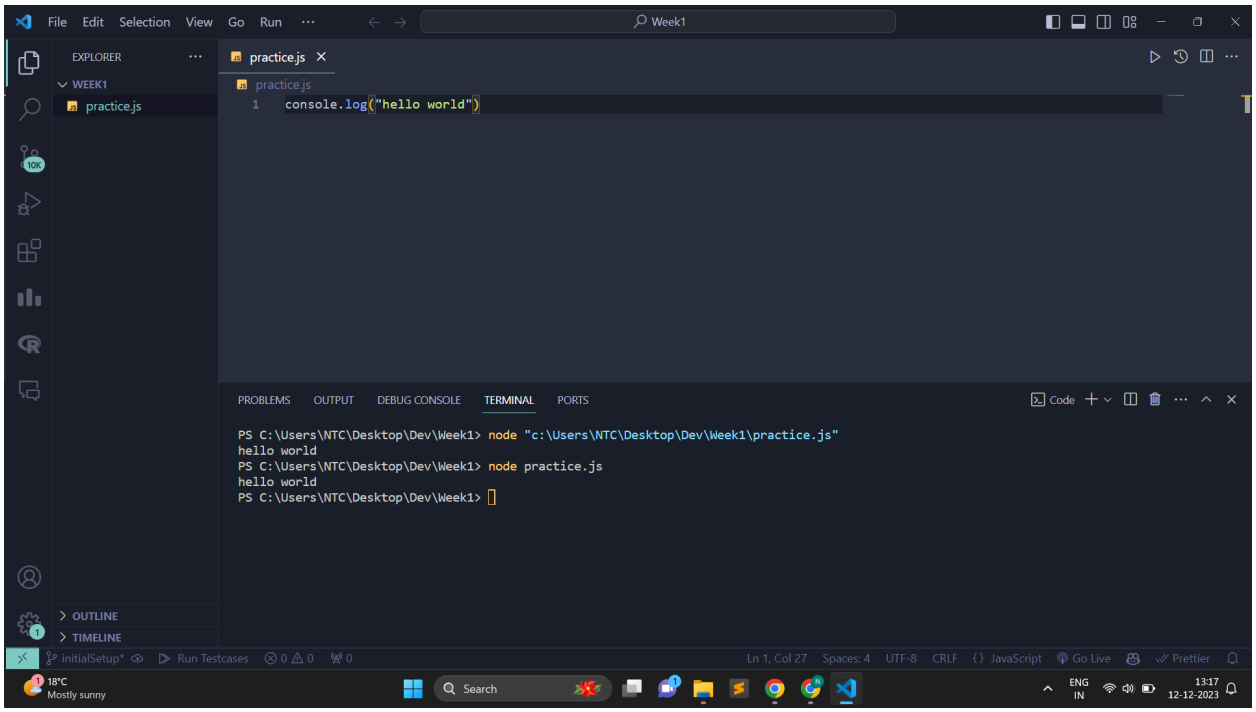



We can directly run this code, we dont do specific compilation step in Js.

This code is still getting converted into binary but it is going line by line.




In compiled language if we have error in c++ code then we wont be able to even make a temp file(consisting binary code)



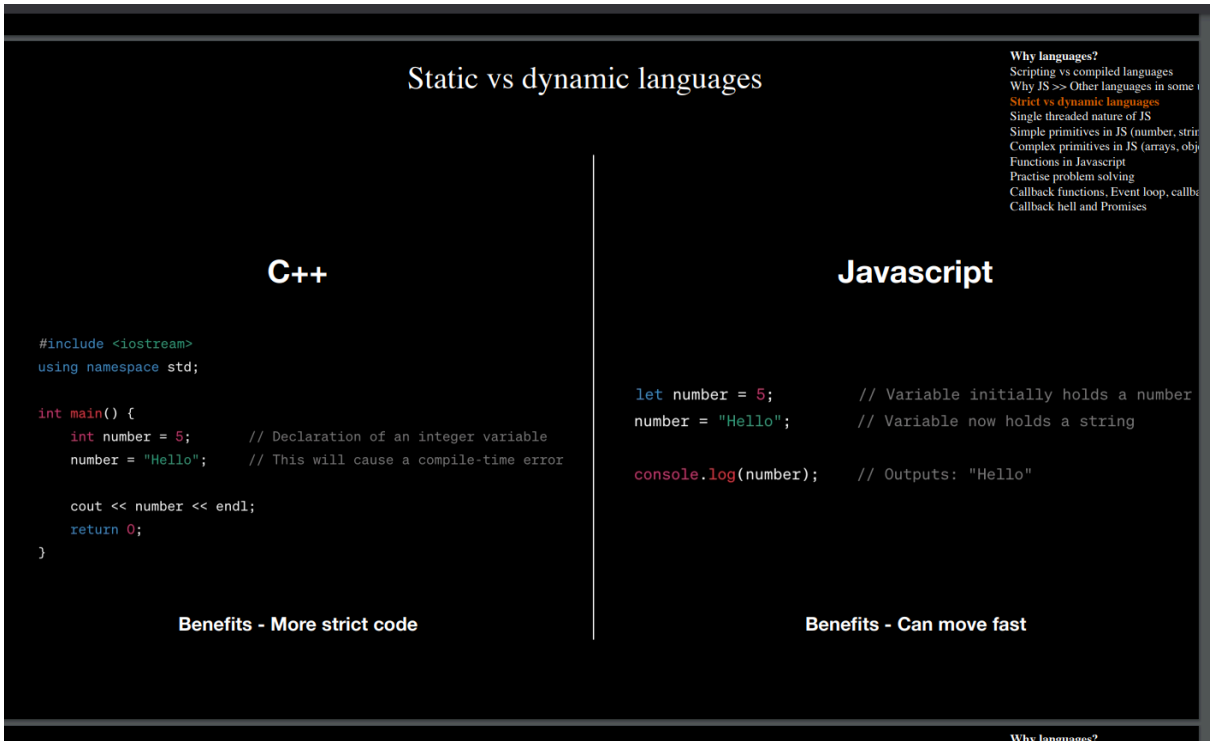

```
console.log("hello world")
console.log(a);
// partially runs in javascript, wont even run partially in C++
```

## Why Js is better than other language

Browsers can only understand HTML/CSS/JS (not technically true)
(All the existing browsers like Brave, Edge, and Chrome are made in such a way that they can understand(interpret) javascript ) but now if we want to make any other language primary like Javascript in the websites then all the existing websites will go and second, we have to create new Browsers that can interpret these new websites in some other primary language used.

Thanks to Node.js, Javascript can also be used for Backend Development

## Strict Vs Dynamic Languages




Focus on the number variable in both languages and see what is happening.
In C++ we face a compile-time error, it basically says that you are trying to change the data type of the number variable from int to string

But in Javascript this thing runs , Js is loosely typed , this is not great because , as the project grows we may face run time error since we are frequently changing the data type. This is not much good for big projects
Here Comes **Typescript** in the picture which make Js more Static .
Typescript is optimization on Javascript

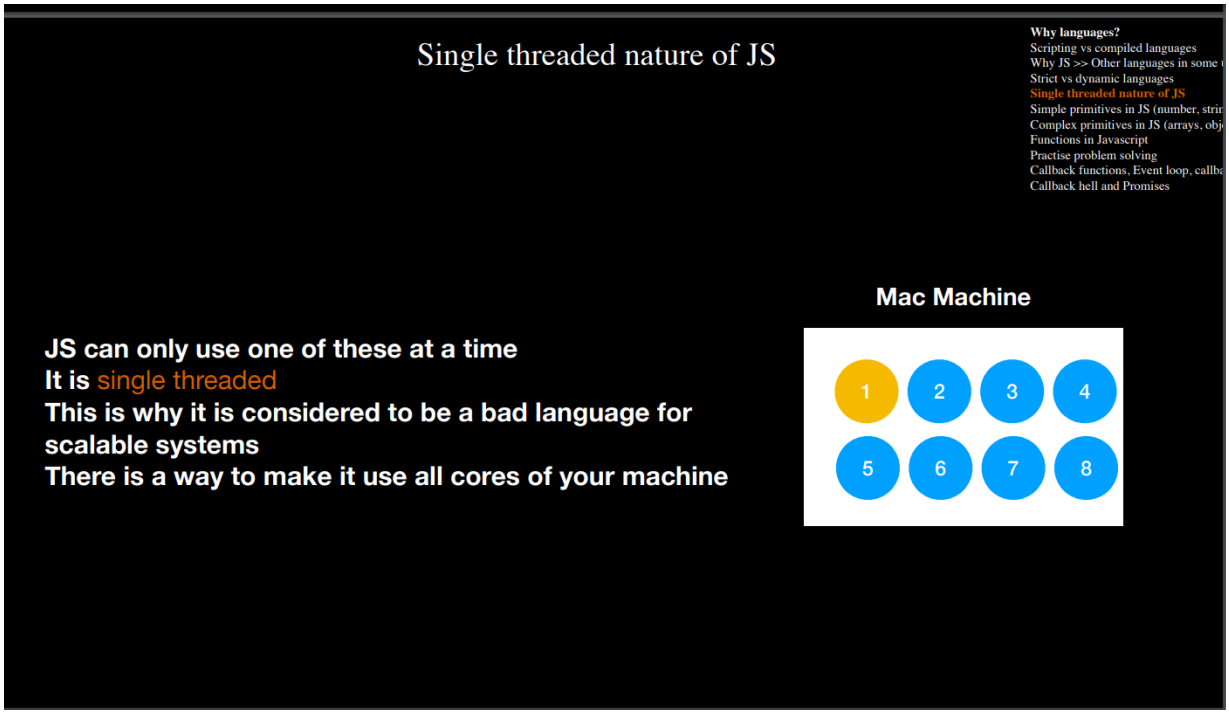
## Single-threaded nature in Js

When we buy a pc we see usually a core, more no. of core more no. of the process we can start.


We can still run more processes than core due to context-switching

Javascript is always restricted to a single core.

We can't split our code into different cores. Which different languages like Java and Golang allow.




Hence it is considered a bad language for scalable systems.

For example, if we have rented 20 core processor in AWS for $500 and if we run nodejs process on it it still runs on a single processor and gives the same performance as if we were running it on a single-core small processor machine.

There is a way to make it use all cores of your machine (by cluster module)

**Simple primitives**

Variables: its values can change

let,var(very less used),const(constant only define once)

```
var a = 1;
a=2;//change value of error
console.log(a); //2
```

let gives the same output but const raises an error(doesn't allow value changing)

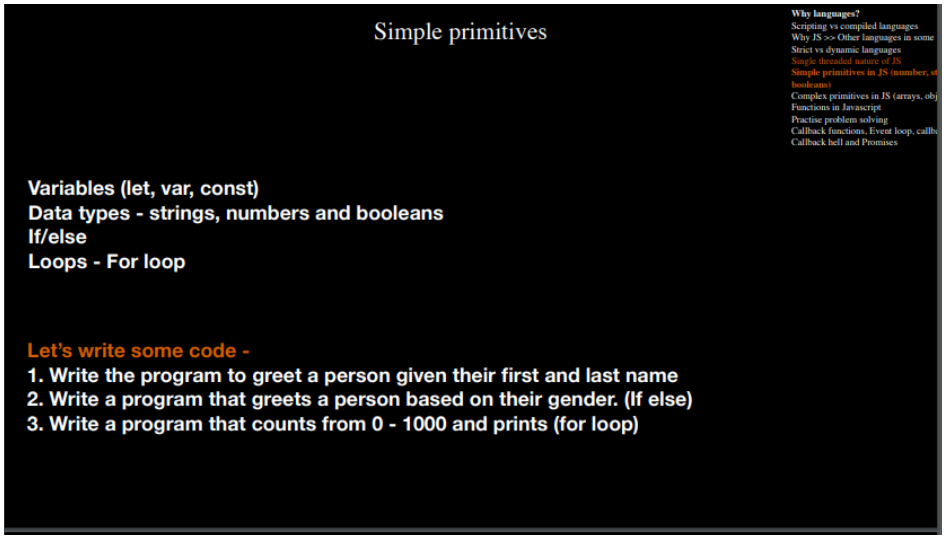
Data types:

let lname = “thapa”

let age = 22

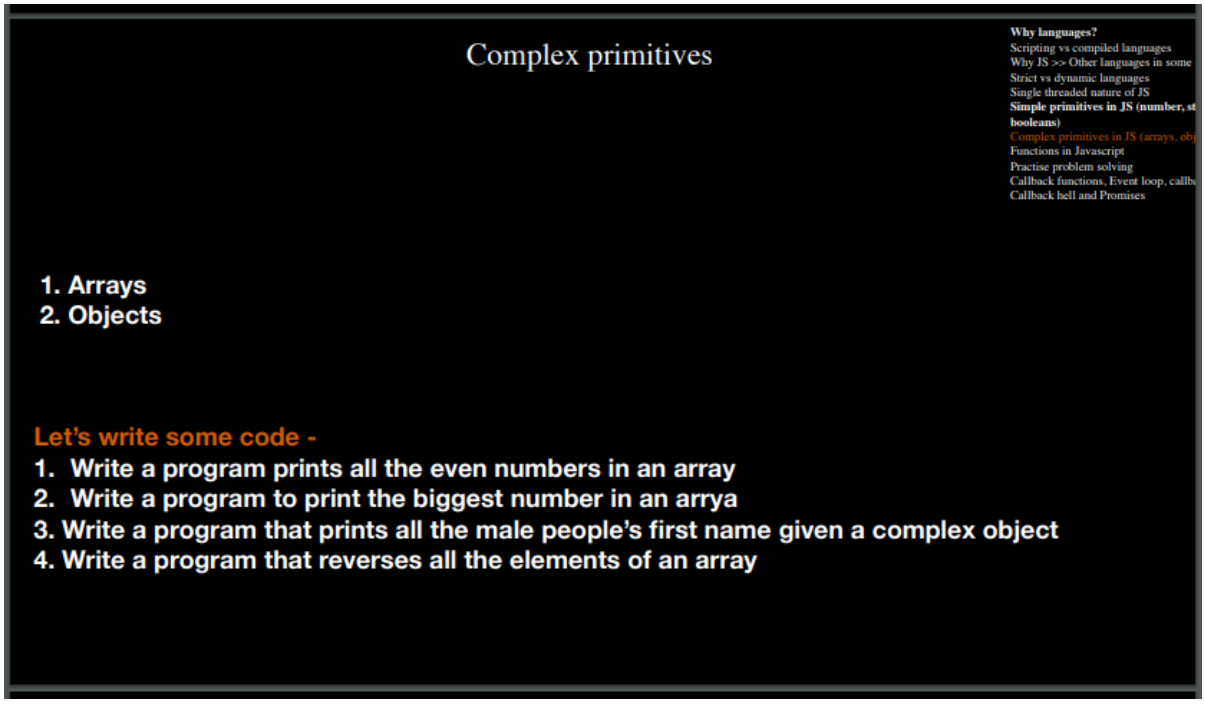
let isMarried = false

console.log(“this person name is “ + name +” their age is “ +age)




```
// Program 1
let firstName ="Nishu"
let lastName = "Thapa"
console.log("Hello " + firstName + " " + lastName)

// Program 2
let gender = 'M'
if(gender=='M'){
    console.log("Hello Miss " + firstName + " " + lastName)
}else{
    console.log("Hello Mr " + firstName + " " + lastName)
}

// Program 3
let sum =0;
for ( let i=1;i<=1000;i++){
    sum = sum + i;
}
console.log(sum)
```

## Complex Primitives




**Arrays**

```
const ages =[21,22,23,25]
console.log(ages[0])
```

**Objects**

```
const person1="Nishant"
const gender1="male"
const person2="Nishu"
const gender2="female"
```

One solution is using arrays for personArray and genderArray, but what if there are more than two component, then we have to define array for each

We can define object

```
const users1={
        firstName:”Nishu”,
        gender:”M”
}
 console.log(users1[“firstName”]) //Nishu
const allUsers = [{
        firstName:’Nishu’,
        gender:’M’
},{
        firstName:’Mishu’,
        gender:’F’
},{
        firstName:’Lishu’,
        gender:’M’
}]
Allusers[i][“gender”] == “male”
// it can become more and more nested . object can also be nested
```

Example:

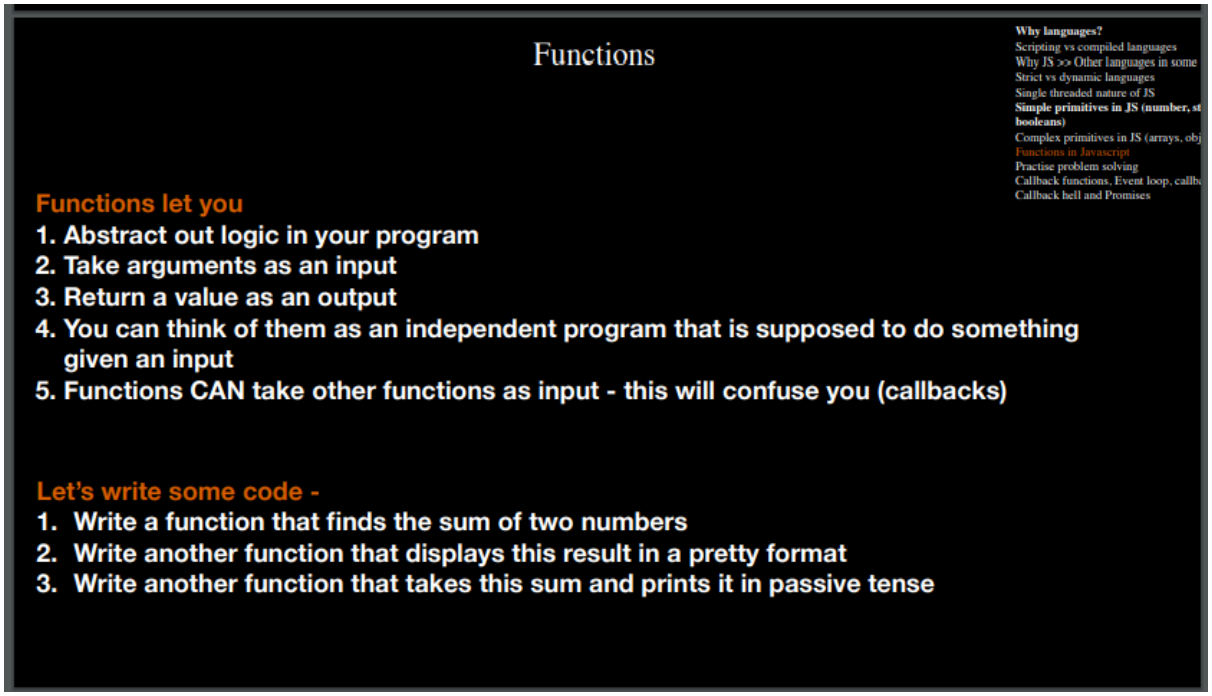
```
const user = {
    name: 'Nishu',
    age: 21,
    address: {
        houseNumber:"11",
        street:"Bhaktapur",
    }
}
console.log(user.address.street);
console.log(user['address']['street']);
const address = user['address'];
const houseNumber = address['houseNumber'];
```

**Solution:**

```
// Program1
const arr = [1,2,12,4,5,6,7,8,9,10];
for(i=0;i<arr.length;i++){
    if(arr[i]%2==0){
        console.log(arr[i]);
    }
}

// Program2
const arr1 = [1,2,12,42,51,60,75,86,99,15];
let biggest = arr1[0];
for(i=0;i<arr1.length;i++){
    if(arr1[i]>biggest){
        biggest = arr1[i];
    }
}
console.log(biggest);

// Program3
const people = [
    {
      fname: 'Alice',
      lname: 'Thapa',
      age: 28,
      gender: 'F'
    },
    {
      fname: 'Bob',
      lname: 'Gurung',
      age: 35,
      gender: 'M'
    },
    {
      fname: 'Charlie',
      lname: 'Magar',
      age: 22,
      gender: 'M'
```

```
    },
    {
        fname: 'Eve',
        lname: 'Lamichhane',
        age: 30,
        gender: 'F'
      },
      {
        fname: 'Frank',
        lname: 'Chhetri',
        age: 40,
        gender: 'M'
      }
  ];

  for(i=0;i<people.length;i++){
      if(people[i].gender=='M'){
          console.log(people[i].fname);
      }
  }
  console.log(" ");

//Program4
// reverse() is an array method, it is not a string method
const arr2=["apple","banana","mango","orange","grapes"];
for(i=0;i<arr2.length;i++){
    const str=arr2[i];
    const rev = str.split('').reverse().join('');
    console.log(rev);
}
// split('') breaks the string into individual characters.
// reverse() reverses the order of these characters in the array.
// join('') joins these reversed characters back into a single string
without any separators, effectively reversing the original string.
```

## Functions




**htop:** gives cores and shows its usage.

for(let i=0; i<10000000000;i++){

sum=sum+i;

}

console.log(sum)

Since javascript is single threaded then why isn't one of the core is not

Going to 100 percent?

It is just that htop command is not working properly in mac

## Challenge

```
function sum(a,b){
    let res= a+b;
    return res;
}
function displaySum(data){
    console.log("result of the sum is: "+data);
}
```

```
function displayResultPassive(data){
    console.log("Sum's result is "+data);
}

//We are allowed to call one function after this
//How will you displayResult of a sum
```

**Solution:**

To run code node practice.js

```
function sum(a,b){
    let res= a+b;
    return displaySum(res);
}
function displaySum(data){
    console.log("result of the sum is: "+data);
    return displayResultPassive(data);
}
function displayResultPassive(data){
    console.log("Sum's result is "+data);
}
sum(2,3);
```

OR

```
function sum(a,b,fnToCall){
    // passing the function as a parameter(argument)
    let res= a+b;
    fnToCall(res);
}
function displaySum(data){
    console.log("result of the sum is: "+data);
}
function displayResultPassive(data){
    console.log("Sum's result is "+data);
}
// Callbacks
const ans =sum(2,3,displaySum);
// passing function as an argument
const ans1 =sum(2,13,displayResultPassive);
```

## Examples

```
function calculateArithmetic(a,b,type){
    if(type=='add'){
        return a+b;
    }else if(type=='sub'){
        return a-b;
    }else if(type=='mul'){
        return a*b;
    }else if(type=='div'){
        return a/b;
    }else{
        return "Invalid type";
    }
}
const value = calculateArithmetic(2,3,'add');
console.log(value);
```

Now we cant write the if logic of addition etc we have to write separate functions for it

```
function calculateArithmetic(a,b,type){
    if(type=='sum'){
        const value = sum(a,b);
        return value;
    }
    if(type=='sub'){
        const value = sub(a,b);
        return value;
    }
}
function sum(a,b){
    return a+b;
}
function sub(a,b){
    return a-b;
```

```
}
calculateArithmetic(1,2,”sum”)
```

Now we are not allowed to do if checks

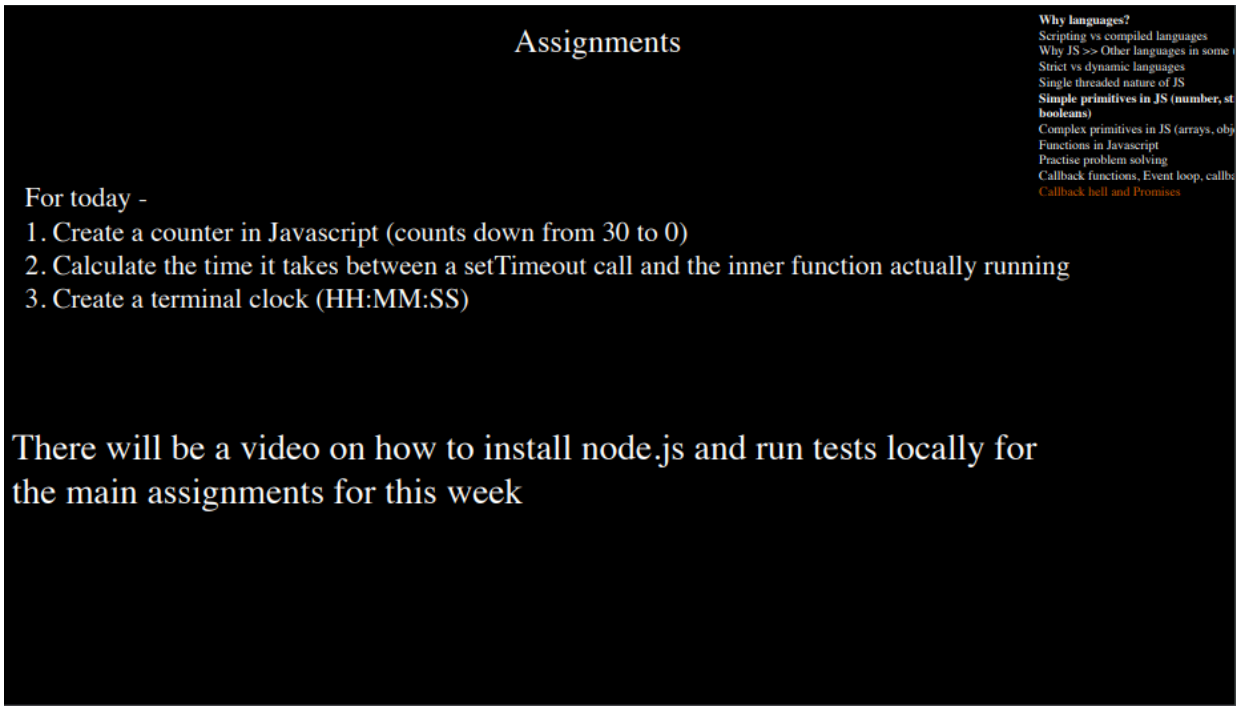
```
function calculateArithmetic(a,b,arithmeticFinalFunction){

    // function arithmeticFinalFunction(a,b){
// It is just like a function defined inside the function and we have
defined a new function called arithmeticFinalFunction
        // return a+b;
    // }
    const ans = arithmeticFinalFunction(a,b);
    return ans;
}

function sum(a,b){
    return a+b;
}
function sub(a,b){
    return a-b;
}
const value = calculateArithmetic(2,3,sum);
//We are passing the function as an argument
console.log(value);
```

Another example

```
function greet(){
    console.log("Hello World");
}
// setTimeout
// after how much time this function should be called in milliseconds
setTimeout(greet,1*1000)
// print after 1 sec
// this is also example of call back

// setInterval will repeat after the interval mentioned
setInterval(greet,3*1000)
```

```
// every 3 sec greet function is called

// function setTimeout(a,b){
//     // somelogic
// }
```

## Assignments




## Q 2:

```
setTimeout(a,1000);
function a(){
    // after how much time exact the control reaches here
}
```

## Solution:

Q1

My solution(with help of gpt lol)

```
let timer =30;
function count(){
    console.log(timer);
    timer--;
}
```

```
const intervalID = setInterval(count,1*950);
setTimeout(()=>{
    clearInterval(intervalID);

},30*1000)
```

Chat gpt

```
let count = 30; // Initial count value

function countdown() {
  const intervalID = setInterval(() => {
    console.clear(); // Clears the console for a cleaner display

    if (count === -1) {
      clearInterval(intervalID); // Stop the interval when count reaches
-1,meaning that it will print till 0,and then stops
      console.log('Countdown completed!');
    } else {
      console.log(count); // Display the current count in the console
      count--; // Decrement count
    }
  }, 1000); // Update the count every 1000 milliseconds (1 second)
}

// Start the countdown
countdown();
```

**Q2:**

```
function measureTime() {
    const startTime = performance.now(); // Record the start time

    setTimeout(() => {
      const endTime = performance.now(); // Record the end time when the
inner function runs
      const elapsedTime = endTime - startTime; // Calculate the elapsed
time
      console.log(`Elapsed time: ${elapsedTime.toFixed(2)} milliseconds`);
    }, 1000); // Set a timeout of 1000 milliseconds (1 second)
  }
```

```
measureTime(); // Call the function to start measuring time
```

In this example, performance.now() is used to capture the current time in milliseconds with high precision. It records the start time before the setTimeout() call and then captures the end time when the inner function inside setTimeout() runs. The difference between the start and end times gives you the elapsed time between the setTimeout() call and the inner function's execution.

**Output:**
Elapsed time: 1014.48 milliseconds

**Q3:**

```
function displayClock() {
    const now = new Date();
    const hours = String(now.getHours()).padStart(2, '0');
    const minutes = String(now.getMinutes()).padStart(2, '0');
    const seconds = String(now.getSeconds()).padStart(2, '0');

    console.clear();
    console.log(`${hours}:${minutes}:${seconds}`);
  }

  setInterval(displayClock, 1000);
```

This code uses setInterval() to repeatedly call the displayClock() function every second (1000 milliseconds).
Inside displayClock(), it retrieves the current time using new Date(), formats the hours, minutes, and seconds to ensure they have leading zeros if needed

(padStart()), clears the console (console.clear()), and then displays the current time in the HH:MM:SS format in the console.
When you run this code in a JavaScript environment (like a browser console or a Node.js environment), it will continuously update the displayed time every second.

**What is a single threaded non-blocking means?**
**What is metadata?**
(non-blocking is closely related to asynchronous call)
**We can write tests using Supertest framework**