# Class-6: JS Execution and Event loop

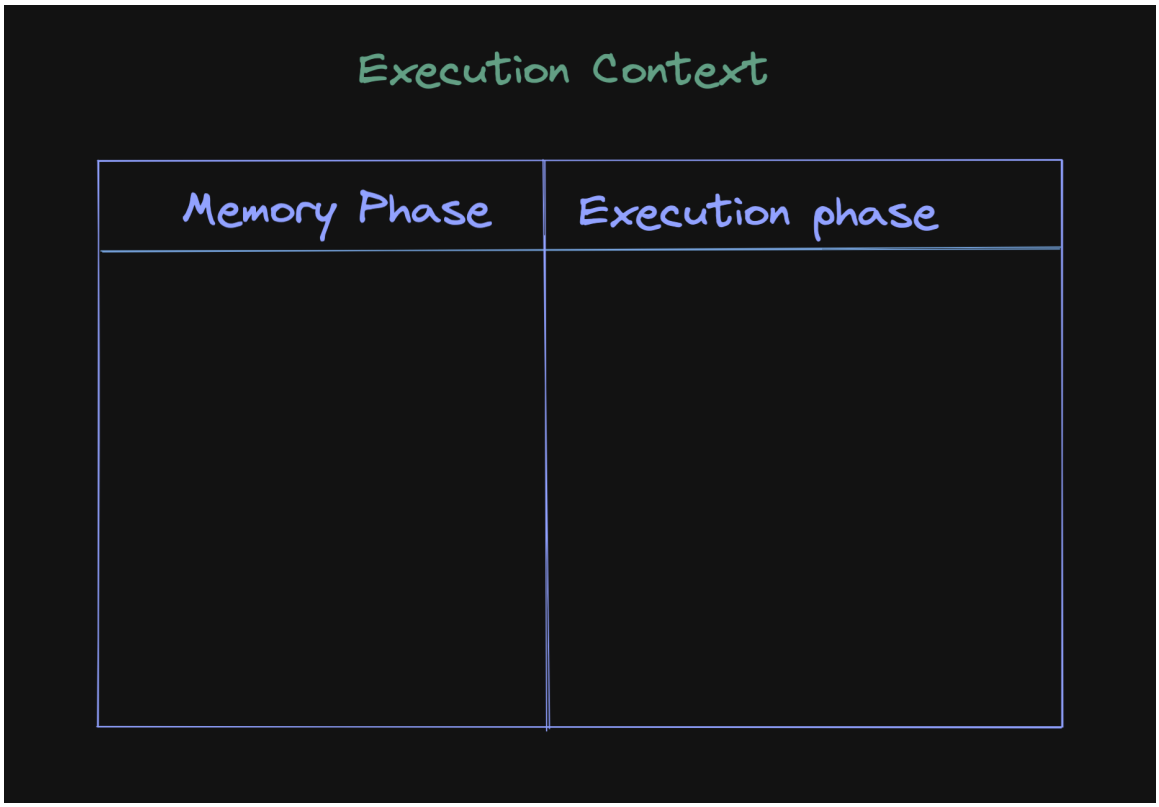
## How JS executes code:

- Lets take a basic code snippet.

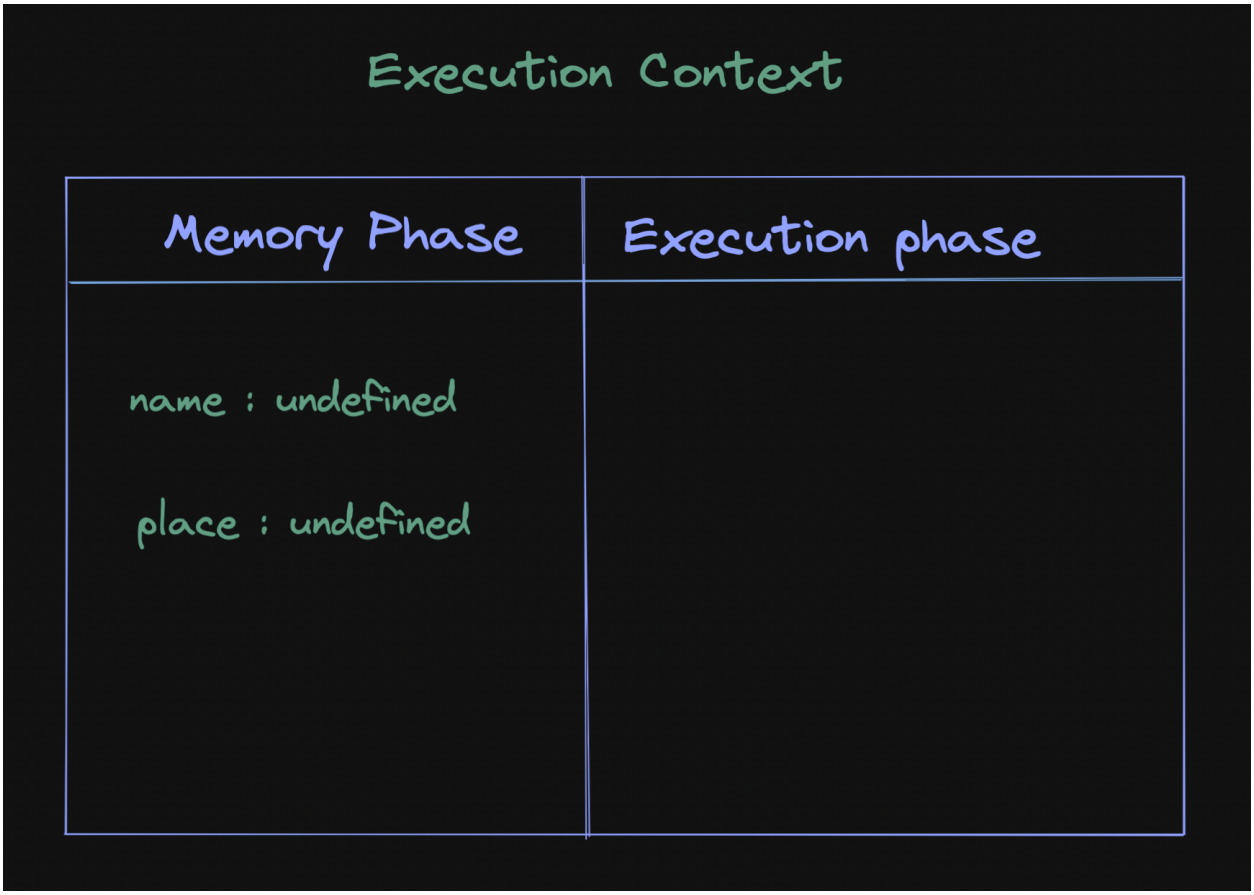
```
var name = "masai"
console.log(name)

var place="Banglore"
console.log(place)
```

```
var name = "masai"

function test(){
    var temp=1
}

test()
```

- Whenever code runs, a Global execution context gets created and it has 2 phases
    - Creation phase or memory phase
    - Execution phase

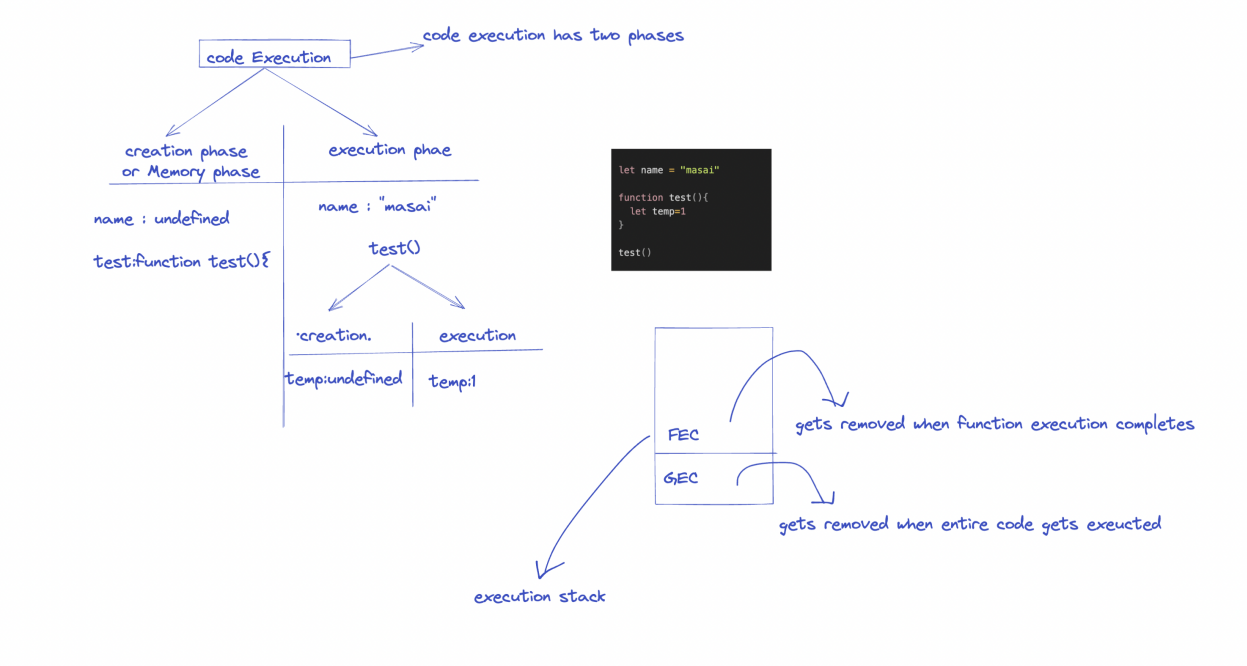



- **Creation phase or Memory phase**
  - During the creation phase of the execution context, JS allocates memory space for the functions and variables.
  - In case of functions, the whole function body is stored but in case of the variables, it is declared and assigned a default value `undefined`. This phenomena is called `Hoisting`
  - So in the above example, name and place will be stored in memory with a value of undefined




- Execution phase:
  - Since memory allocation is done, now it will go into execution phase .
  - In this phase, JS executes our code line by line and assigns the value to the variables.

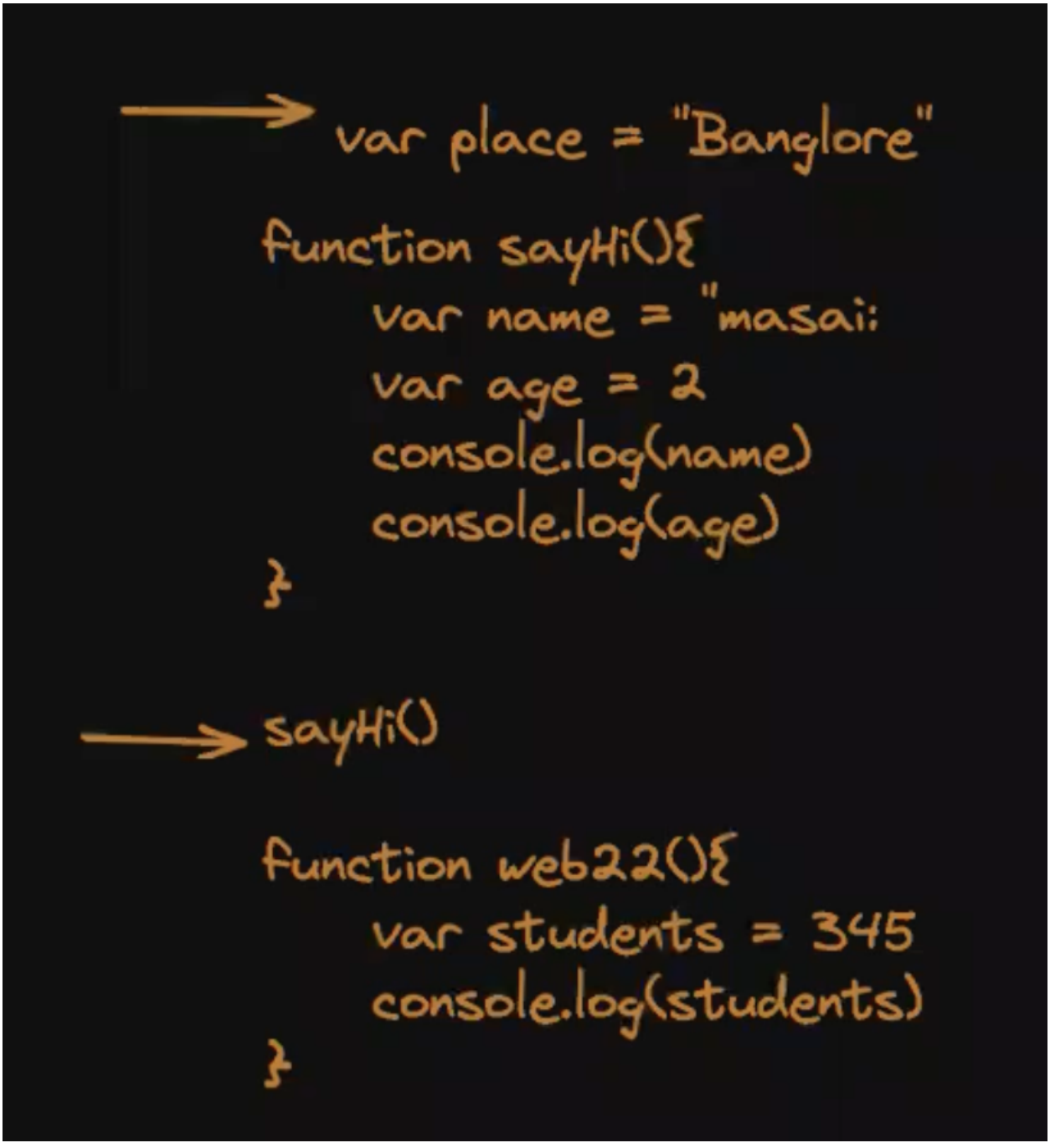
- Whatever we saw now is GEC (Global execution phase)
- There is one more thing called as functional execution phase
- Whenever a function gets invoked, a new execution context gets created known as functional execution context.
- It has two phases again
  - creation and execution phase
- Once function execution is done, functional execution context gets removed.

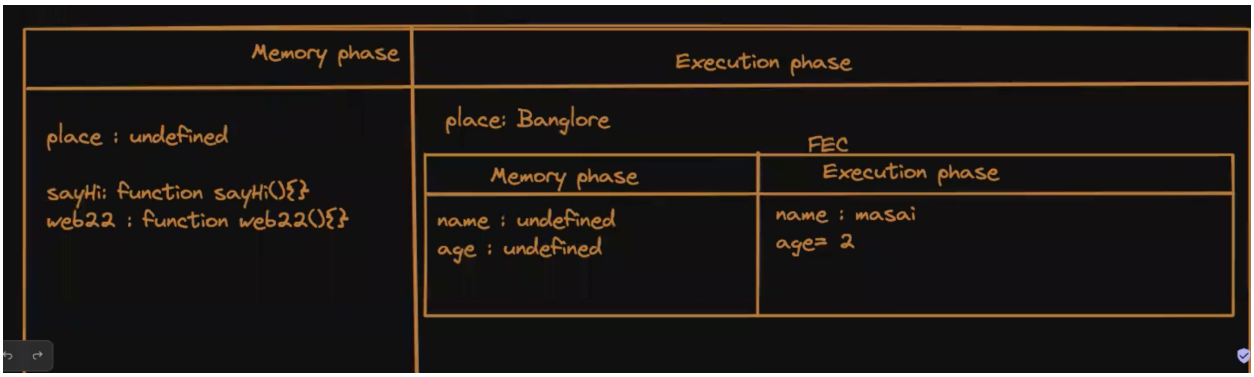


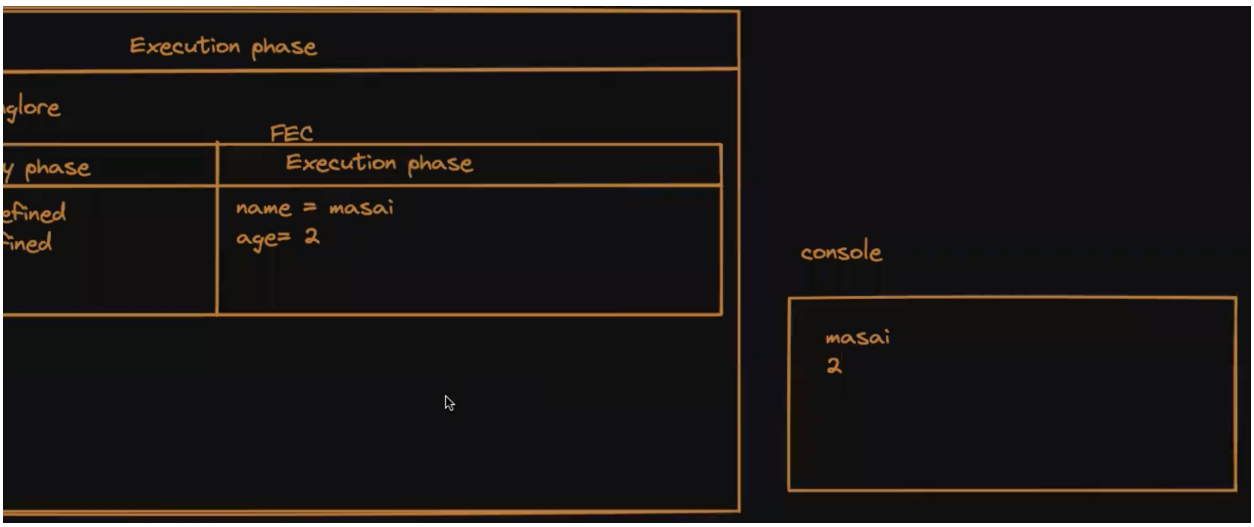

# Another example

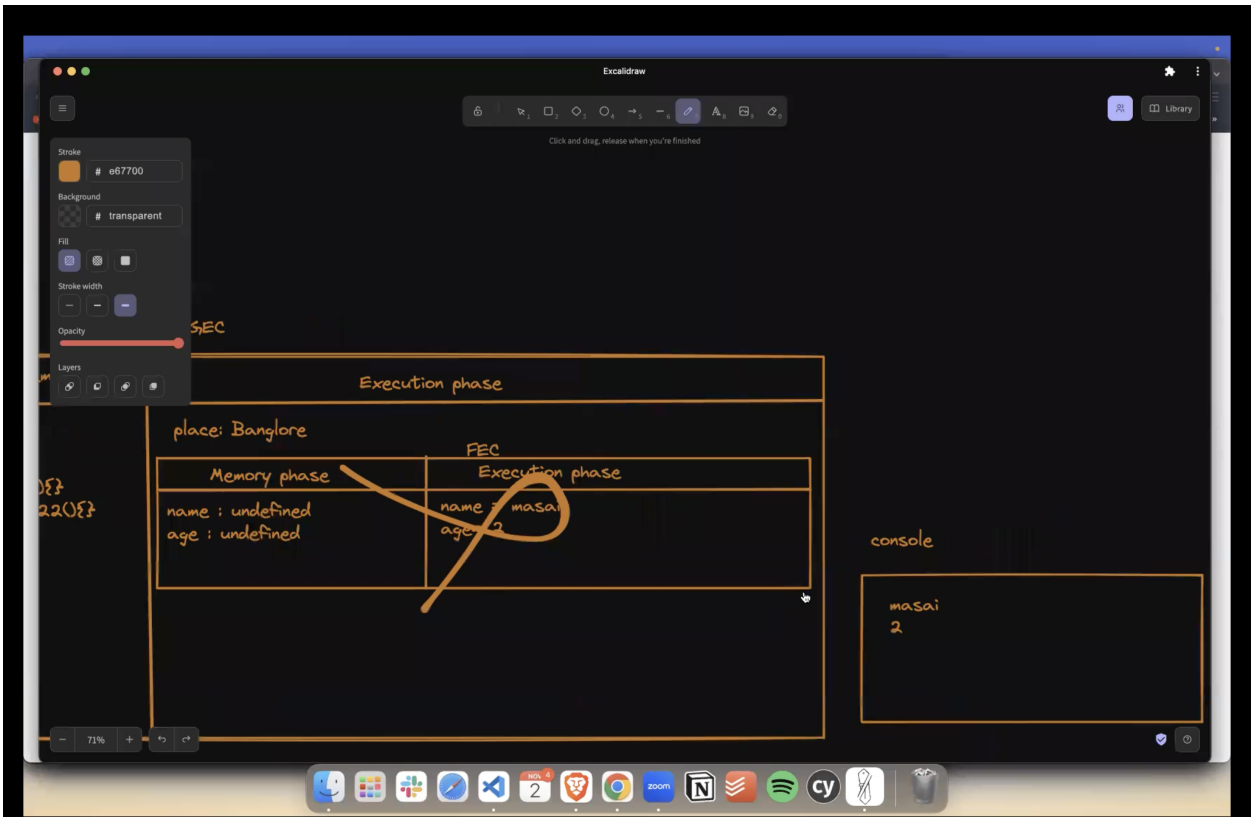
```
var place = "Banglore";

function sayHi(){
    var name = "masai";
    var age = 2;
    console.log(name);
    console.log(age);
}

sayHi();

function web22(){
    var students = 345;
    console.log(students);
}
```

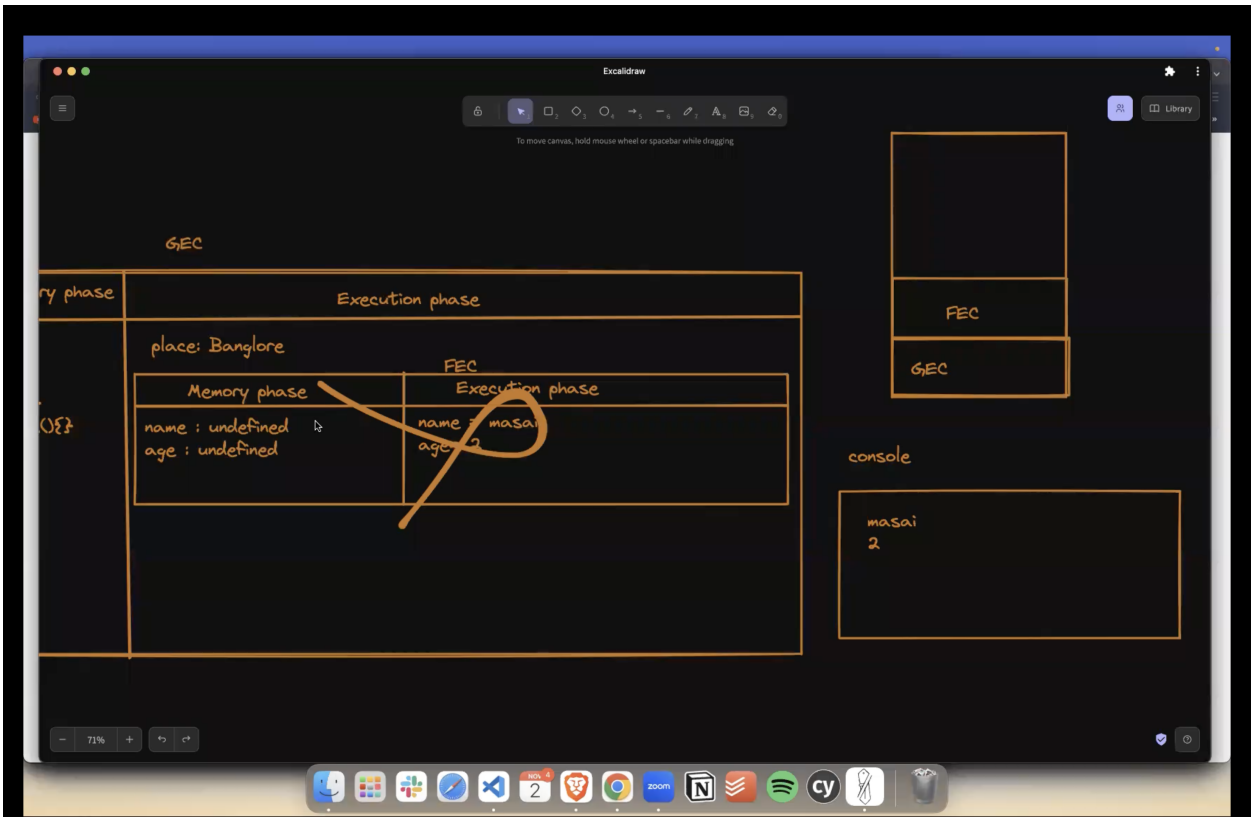
```
web22();
```


var place = "Banglore"
function sayHi(){
var name = "masai;
var age = 2
console.log(name)
console.log(age)
}
sayHi()
function web22(){
var students = 345
console.log(students)
}


Memory phase
Execution phase
place : undefined
sayHi: function sayHi(){}
web22 : function web22(){}
place: Banglore
FEC
Memory phase
Execution phase
name : undefined
age : undefined
name : masai
age= 2


Execution phase
glore
FEC
y phase
Execution phase
efined
ined
name = masai
age= 2
console
masai
2


Excalidraw
Click and drag, release when you're finished
Stroke
# e67700
Background
# transparent
Fill
Stroke width
Opacity
Layers
Library
GEC
Execution phase
place: Banglore
FEC
Memory phase
Execution phase
name : undefined
age : undefined
console
masai
2
71%


Excalidraw
To move canvas, hold mouse wheel or spacebar while dragging
Library
GEC
Execution phase
place: Banglore
FEC
Memory phase
Execution phase
name : undefined
age : undefined
FEC
GEC
console
masai
2
71%

Try these 2 example in python tutor

```
var user = "Arya";

function sayHello(){
  return `Hello ${user}`;
}

var userMsg = sayHello(user);
```

```
function sayHi() {
  var name = 'Lydia';
  var age = 21;
  console.log(name);
  console.log(age);
}

sayHi();
```

```
function sayName() {
console.log('Name: ', myVar);

  function sayAge() {
    myVar = 36;
    console.log('Age: ', myVar);
  }

  sayAge();
}

var myVar = 'Vivek';
sayName();
```

# Asynchronous Programming

## What and why?

- In its most basic form, Javascript is synchronous, blocking, single-threaded language.
- Lets discuss these three terms

## Synchronous:

- As the name suggests synchronous means to be in a sequence, i.e. every statement of the code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed.
- Let’s take some real life examples.
    - Think of Big-Bazaar counter scenario: You should wait until the person in front of you completes the billing.
    - You can directly jump above them and pay the bill right? 😀


## Student’s task:

- Ask students about some more real life examples of synchronous operations.

## Instructor Task

- Same applies for Javascript too, code execution will go line by line.
- In below example Task 1 will be printed first and so on till Task 4

```
console.log('Task 1')
console.log('Task 2')
console.log('Task 3')
console.log('Task 4')
```

## Blocking:

- Take same Big Bazaar scenerio, you should wait untill person in front of you completes billing, basically it is blocking untill previous operation is completed.
- In below example no matter how long for loop takes, after completing entire for loop only Task 3 will be printed

```
  console.log('Task 1')
console.log('Task 2')

  for(var i=0;i<1000000;i++){
      if(i===0) console.log("Loop Started");
  if(i===1000000) console.log("Loop Ended");
  }
console.log('Task 3')
console.log('Task 4')
```

**Output:**

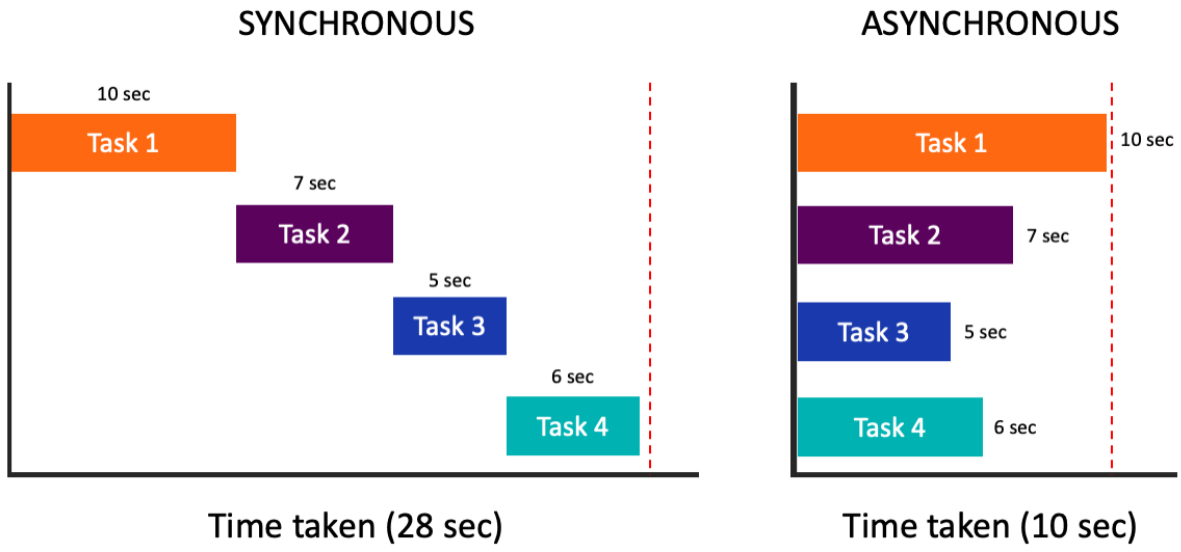
```
Task 1
Task 2
Loop Started
<A long Delay>
Loop Ended
Task 3
Task 4
```

## Single Threaded:

- A thread is simply a process that your javascript program can use to run a task
- Each thread can only do one task at a time
- JavaScript has just the one thread called the main thread for executing any code

## Synchronous vs Asynchronous:

- We have to wait for other process to get executed.
- From below example
  - Time taken to complete all tasks synchronously is 28 secs whereas
  - Time taken to complete all tasks asynchronously is 10 secs




- So we need a way to have asynchronous behaviour with JS.

## How?

- **Now lets take KFC or Dominos scenerio:**

- Whenever you go to KFC you won't wait untill your order is prepared, you will be pushed into waiting area to wait for your order. In the meanwhile the person next to you will make his order. This is asynchronous right?

Order queue

Waiting area

- For acheiving asynchronous behaviour, Just JavaScript is not enough
- We need new pieces which are outside of JavaScript to help us write asynchronous code which is where web browsers come into play
- Web browsers define functions and APIs that allow us to register functions that should not be executed synchronously, and should instead be invoked asynchronously when some kind of event occurs
- For example, that could be the passage of time ( setTimeout or setInterval), the user's interaction with the mouse (addEventListener), or the arrival of data over the network (callbacks, Promises, async-await)
- You can let your code do several things at the same time without stopping or blocking your main thread.

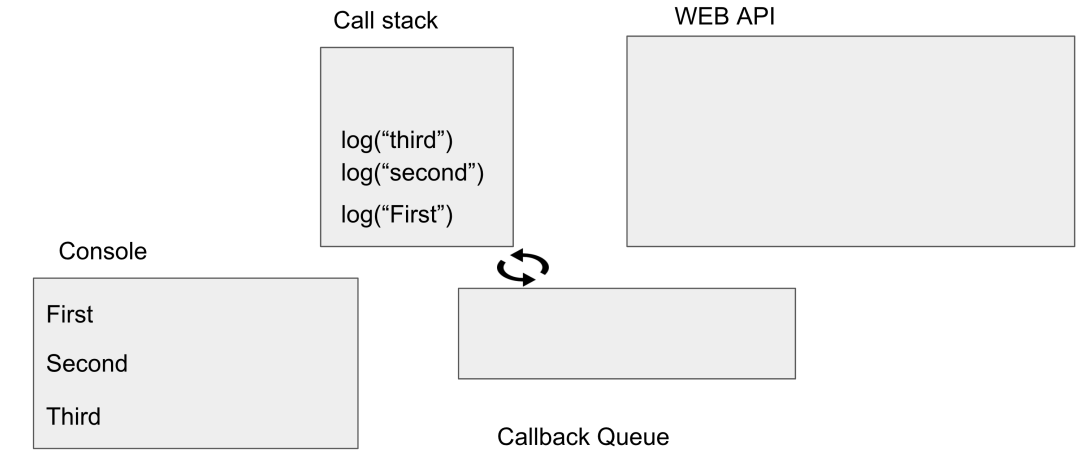
## Web APIs:

- To make async programming possible JS alone isn’t enough, we also need WEB broswer APIs

These Web APIs makes asynchronous programming possible

- setTimeout
- setInterval
- Local Storage
- fetch
- JS run time environment consists of following parts
    - Memory Heap: Stores all variables
    - Call stack: Whenever you execute code functions are pushed onto callstack and whenever functions returns it pops out of call stack
    - WEB APIs: SetTimeout, setInterval, Promise, Remember these APIs are provided by browser.
    - Call stack Queue: FIFO DS
    - Event loop: has only one job, check if callstack is empty, if yes push an item from queue into stack

```
console.log("one")
console.log("two")
console.log("three")
```




- Open http://latentflip.com/loupe/
- Open http://latentflip.com/loupe/

```
console.log("1")

setTimeout(function test(){
    console.log("2")
},2000)

setTimeout(function test2(){
    console.log("4")
},4000)
```

```
console.log("3")
```

## Call Stack

**Definition**: A stack data structure that keeps track of execution contexts in the program.

**Functionality**:

- **Push**: A new execution context is pushed onto the stack when a function is called.
- **Pop**: The execution context is popped off the stack when the function execution is complete.

**Example**: Show in this https://www.jsv9000.app/

```
javascriptCopy code
function firstFunction() {
    console.log("Inside firstFunction");
    secondFunction();
    console.log("Exiting firstFunction");
}

function secondFunction() {
    console.log("Inside secondFunction");
}

console.log("Script start");
firstFunction();
console.log("Script end");

// Output:
// Script start
// Inside firstFunction
```

```
// Inside secondFunction
// Exiting firstFunction
// Script end
```

## Callback Queue

**Definition**: A queue that holds callback functions ready to be executed after their associated asynchronous operations complete.

**Functionality**:

- Callbacks are placed in the queue after their associated asynchronous operations (e.g., `setTimeout`, `setInterval`, HTTP requests) complete.
- Follows the FIFO (First In, First Out) principle.

**Example**:

```
console.log("Script start");

setTimeout(() => {
    console.log("setTimeout callback");
}, 1000);

console.log("Script end");

// Output:
// Script start
// Script end
// (after 1 second) setTimeout callback
```

## Web APIs

**Definition**: Browser-provided interfaces for performing tasks asynchronously, such as `setTimeout`, DOM manipulation, and HTTP requests.

**Functionality**:

- Handle operations outside the main JavaScript thread.

- Upon completion, they pass callbacks to the callback queue.

**Example**:

```
console.log("Script start");

setTimeout(() => {
    console.log("setTimeout callback");
}, 1000);

console.log("Script end");

// Output:
// Script start
// Script end
// (after 1 second) setTimeout callback
```

# Event Loop

**Definition**: A mechanism that continuously checks the call stack and the callback queue.

**Functionality**:

- If the call stack is empty, the event loop dequeues a callback from the callback queue and pushes its context onto the call stack for execution.

**Example**:

```
javascriptCopy code
console.log("Script start");

setTimeout(() => {
    console.log("setTimeout callback");
}, 1000);

console.log("Script end");
```

```
// Output:
// Script start
// Script end
// (after 1 second) setTimeout callback
```

Here are the examples converted to traditional function expressions:

## Question 1

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

setTimeout(function() {
    console.log("Timeout 1");
}, 1000);

setTimeout(function() {
    console.log("Timeout 2");
}, 0);

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

## Question 2

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

setInterval(function() {
    console.log("Interval 1");
}, 1000);

setTimeout(function() {
    console.log("Timeout 1");
```

```
}, 2500);

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

## Question 3

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

setTimeout(function() {
    console.log("Timeout 1");
}, 2000);

setTimeout(function() {
    console.log("Timeout 2");
}, 1000);

setTimeout(function() {
    console.log("Timeout 3");
}, 0);

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

## Question 4

```
javascriptCopy code
let count = 0;

const interval = setInterval(function() {
    console.log("Interval count: " + ++count);
    if (count === 3) {
        clearInterval(interval);
    }
```

```
}, 1000);

setTimeout(function() {
    console.log("Timeout 1");
}, 2500);

setTimeout(function() {
    console.log("Timeout 2");
}, 4000);
```

## Question 5

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

setTimeout(function() {
    console.log("Timeout 1");
}, 500);

setInterval(function() {
    console.log("Interval 1");
}, 300);

setTimeout(function() {
    console.log("Timeout 2");
}, 900);

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

## Question 6

```
javascriptCopy code
let count = 0;
```

```
const interval = setInterval(function() {
    console.log("Interval count: " + ++count);
}, 1000);

setTimeout(function() {
    console.log("Timeout 1");
    clearInterval(interval);
}, 4500);
```

## Question 7

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

setTimeout(function() {
    console.log("Timeout 1");
}, 100);

setInterval(function() {
    console.log("Interval 1");
}, 50);

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

## Question 8

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

setTimeout(function() {
    console.log("Timeout 1");
}, 0);
```

```
setTimeout(function() {
    console.log("Timeout 2");
}, 10);

setTimeout(function() {
    console.log("Timeout 3");
}, 20);

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

## Question 9

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

setInterval(function() {
    console.log("Interval 1");
}, 1000);

setInterval(function() {
    console.log("Interval 2");
}, 1500);

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

## Question 10

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

setTimeout(function() {
    console.log("Timeout 1");
```

```
}, 300);

setInterval(function() {
    console.log("Interval 1");
}, 100);

setTimeout(function() {
    console.log("Timeout 2");
}, 600);

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

4o

## Slideshow app- https://codepen.io/abduljabbarpeer/pen/qBoZmVW

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-s
    <title>Document</title>
  </head>
  <body>
    <button onClick="start()">START</button>
    <button onClick="stop()">STOP</button>

    <div id="container">
      <!-- <img
```

```
        src="https://cdn.pixabay.com/photo/2019/05/29/14/30/cin
        alt=""
      /> -->
    </div>
  </body>
</html>

<script>
  var images = [
    "https://cdn.pixabay.com/photo/2019/05/29/14/30/cinque-terre
    "https://cdn.pixabay.com/photo/2022/01/28/21/10/honeybee-697
    "https://cdn.pixabay.com/photo/2021/10/16/05/43/love-671397
    "https://cdn.pixabay.com/photo/2022/01/25/12/58/conifer-6960
  ];

  let x;
  function start() {
    let slideImg = document.createElement("img");
    let i = 0;
    x = setInterval(function () {
      if (i == images.length) {
        i = 0;
      }
      slideImg.src = images[i++];
      document.querySelector("#container").append(slideImg);
    }, 2000);
  }

  function stop() {
    clearInterval(x);
  }
</script>
```

## Timer:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-s
    <title>DIGITAL CLOCK</title>
    <style>
      * {
        margin: 0;
        padding: 0;
      }

      #clock {
        display: flex;
        justify-content: center;
        align-items: center;
        height: 100vh;
        background: #b58022;
        /* direction,color1,color2...link--- */
      }

      #clock > div {
        padding: 30px;
        margin: 30px;
        border-radius: 20px;
        display: flex;
        justify-content: center;
        align-items: center;
        font-size: 50px;
        box-shadow: 5px 5px 23px #7b5717, -5px -5px 23px #ffc535
      }

      span {
```

```
        font-size: 40px;
      }
    </style>
  </head>
  <body>
    <!-- <div class="container">
      <div class="box">
        <div class="hours">
          <p id="hours">8</p>
        </div>
        <p>hours</p>
      </div>
      <div class="box">
        <div class="minutes">
          <p id="minutes">8</p>
        </div>
        <p>minutes</p>
      </div>
      <div class="box">
        <div class="seconds">
          <p id="seconds">8</p>
        </div>
        <p>seconds</p>
      </div>
      <div class="box">
        <div class="half">
          <p id="half">8</p>
        </div>
      </div>
    </div> -->
    <div id="clock">
      <div id="hours">13</div>
      <span>:</span>
      <div id="minutes">13</div>
      <span>:</span>
      <div id="seconds">13</div>
```

```
      <div id="half">PM</div>
    </div>
    <script>
      var clock = function () {
        var date = new Date();
        //console.log(date)
        var hours = date.getHours();
        console.log(hours)
        var minutes = date.getMinutes();
        var seconds = date.getSeconds();
        var half = "";
        if (hours >= 12) {
          half = "PM";
        } else {
          half = "AM";
        }
        document.getElementById("hours").innerHTML = hours;
        document.getElementById("minutes").innerHTML = minutes;
        document.getElementById("seconds").innerHTML = seconds;
        document.getElementById("half").innerHTML = half;
        setTimeout(clock, 1000);
      };

      clock();
      // here we used clock(); for calling the function again ar
    </script>
  </body>
</html>
```

- a