# JAVASCRIPT

## UNDER THE HOOD

ARIHARASUDHAN

# THIS BOOK

This book is written (typed) by Ari, who hails from the South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website, where you can reach out to him.
https://arihara-sudhan.github.io

# THREAD AND CALL-STACK

JavaScript is a single threaded language. It is a synchronous language with asynchronous abilities. It may sound like a sweet chilli! Can a chilli have sweetness? Can a synchronous language have asynchronous abilities? JavaScript is an exception! In a single threaded language, one operation has to be completed to perform the next operation.

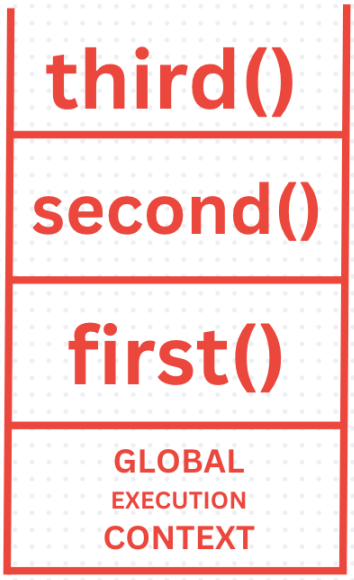
How can the exceptional JavaScript achieve Asynchronous stuffs being a single threaded language? EVENT LOOP is the answer for this question. We'll explore it later in this book. A thread has a stack and a heap. A Call-stack and a Memory-heap! If we have only one thread, YES! There is only one call-stack and a memory-heap. The Call-stack is a stack data-structure that follows LIFO Principle. It keeps track of our functions. It manages the so called, Execution Context, which we are about to explore in a while.

For the following snippet,

CALL-STACK

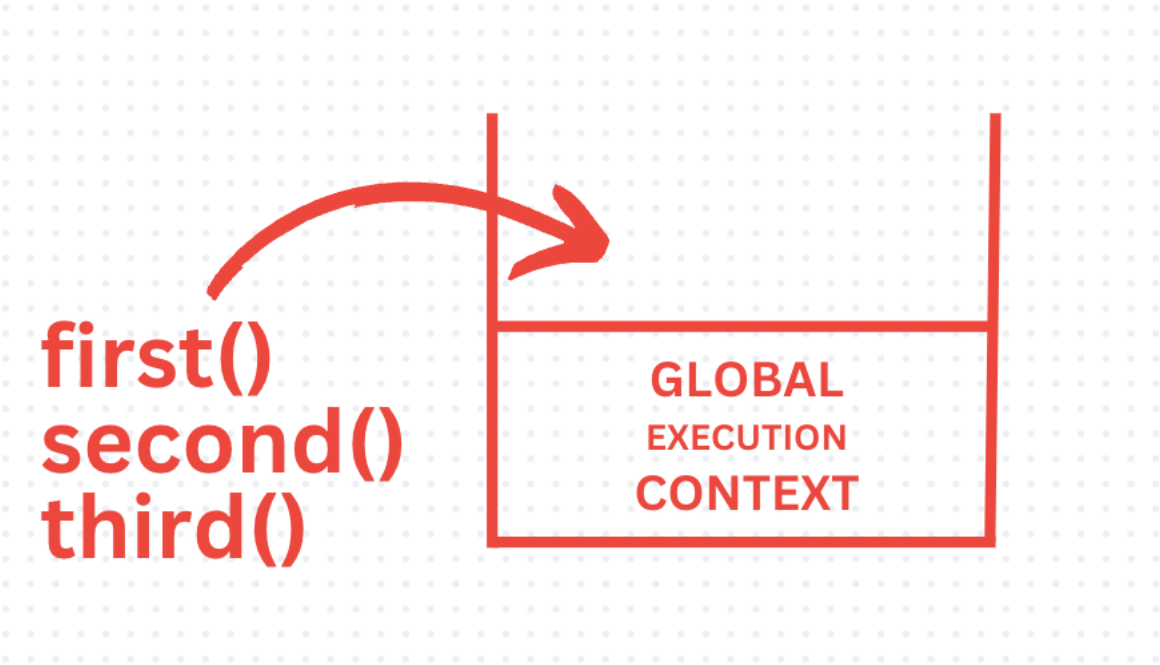
```
function first(){
    console.log("FIRST FUNCTION");
}

function second(){
    console.log("SECOND FUNCTION");
}

function third(){
    console.log("THIRD FUNCTION");
}

first();
second();
third();
```

We may expect a stack like,

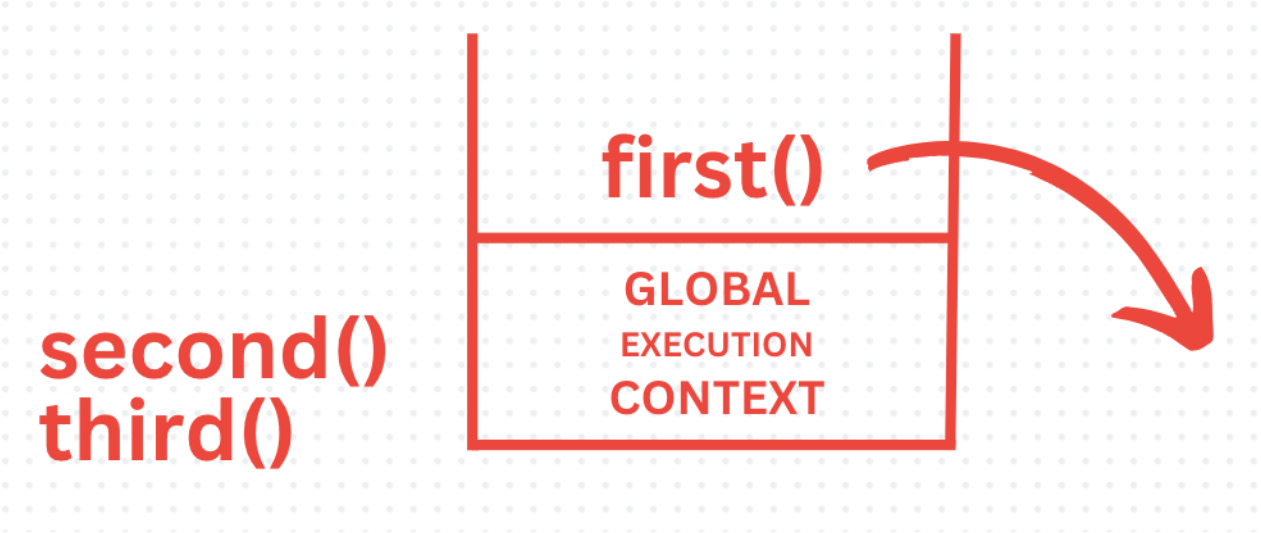



It doesn't mean that the stuffs in the stack should be stacked always as shown right here. In our code, no functions depend on others.

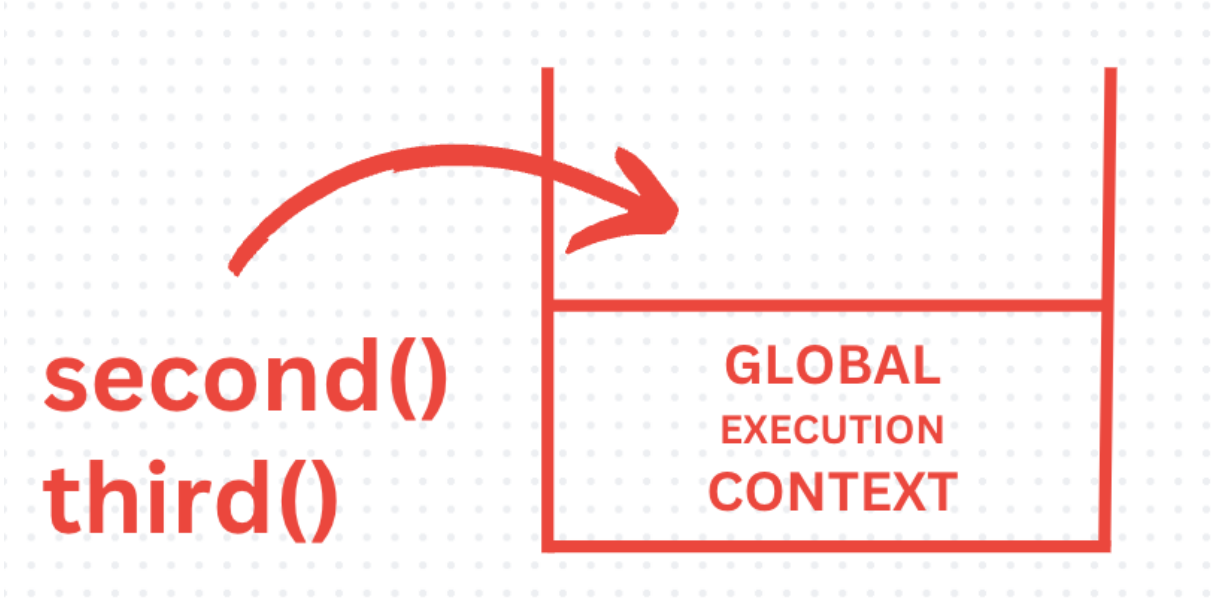
So, the call-stack will not hold all the functions stacked in at a time. Here, first() will be pushed into the stack.




On completion, it will be popped out and the next function will be pushed in.



The second() function is pushed into the stack for execution.

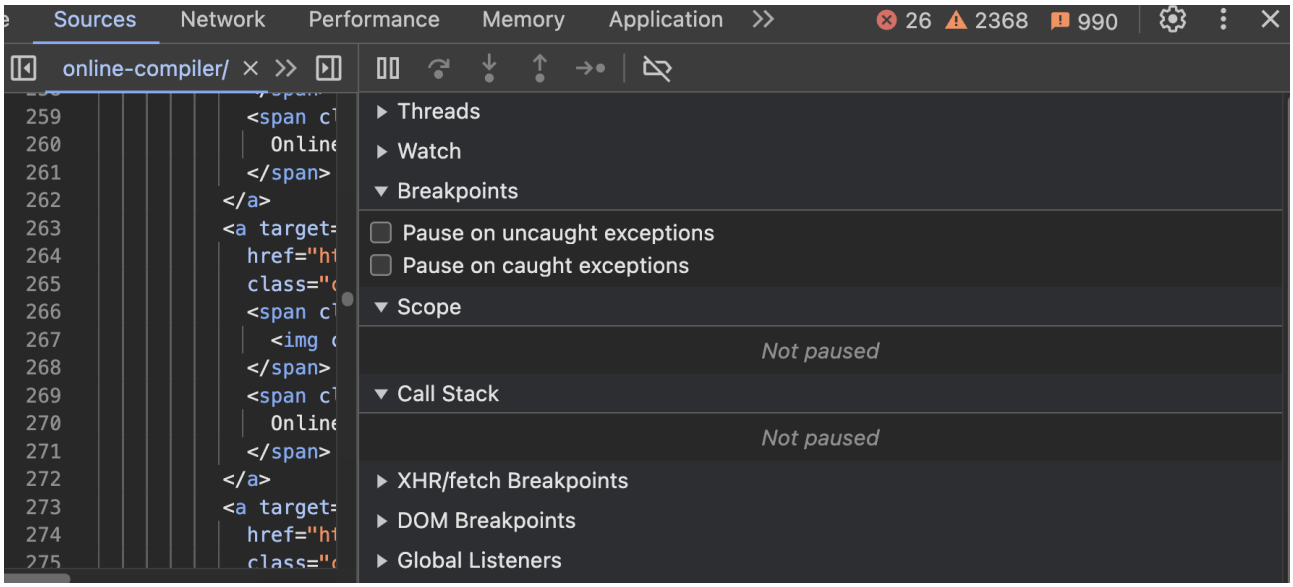



After it gets executed, it will be popped out as shown below.

Now, The third() function is pushed into the stack for execution.


After it gets executed, it will be popped out.

We can find the call-stack in under sources in developer mode.




We can put debuggers and check the call-stack. If we have a function that call another, we can see those functions stacked in.

CALL-STACK

```
function first(){
    second();
}

function second(){
    third();
}

function third(){
    console.log("THIRD FUNCTION");
}

first();
```

It will stack those functions in the call-stack as we discussed.


With debugger steps, we can also observe that these are being popped out in the reverse order (LIFO).


# EXECUTION CONTEXT

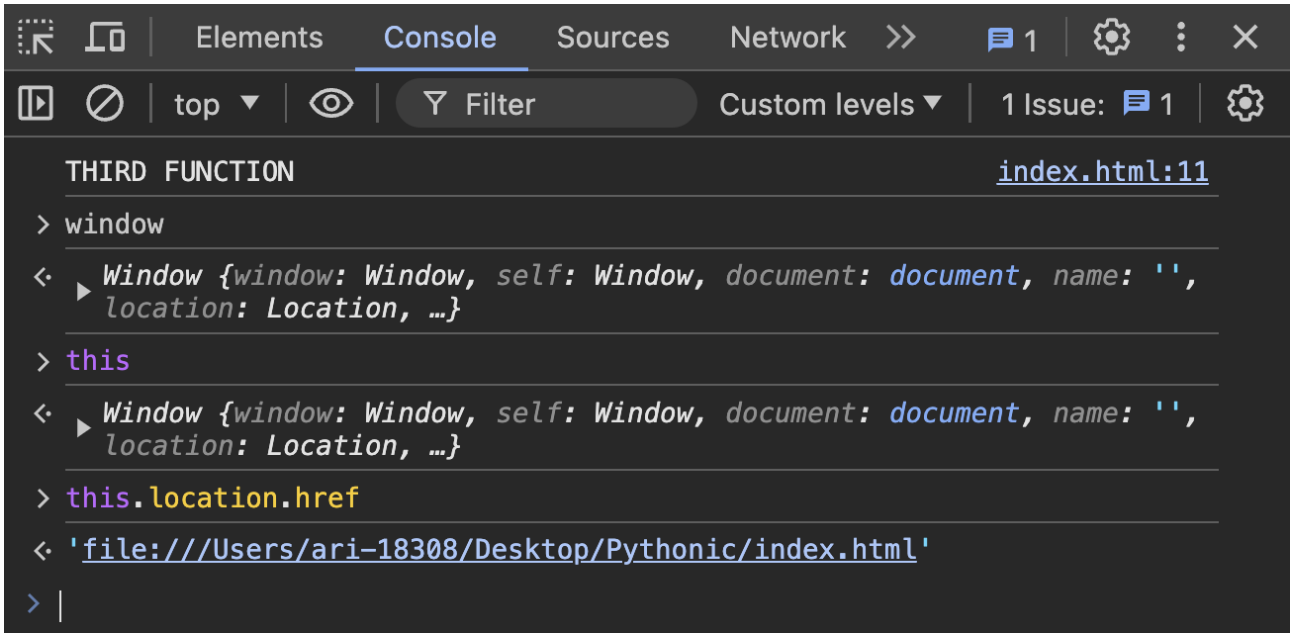
Whenever we run our JavaScript program, there will be an environment created for handling the transformation and the execution of our program. The execution context contains the current running program and everything required to aid the current running program. The environment may look like the following:

| MEMORY | EXECUTION |
| --- | --- |
| name: “Ari”<br>a: 4<br>b: 7<br><br>function1: {....}<br>function2: {....}<br><br>STORING VARIABLES AND FUNCTIONS AS KEY-VALUE PAIRS |  <br><br>EXECUTION HAPPENS HERE! EACH LINE OF CODE IS EXECUTED! |

Execution Context has two phases. Firstly, it's **Memory Creation Phase**. Secondly, it's **Execution Phase**. We can visualize it like parsing into our code twice. One time is for checking and creating memory for variables and functions. Another time is for executing them!

In The Creation Phase, these are the following actions happened.
>> Creating global Object (window in browser; global in nodejs)
>> Creating this Object and binding it to the global object
>> Setting Up memory-heaps for storing the variables and function references
>> Storing functions and variables in the global execution context and setting the variables to undefined.

We can check the global object, this object in the developer window as well.




During the memory creation phase, variables will be given undefined value. In The Execution Phase, there happens a line by line execution. For each function calls, there will be separate execution context created.

EXECUTION CONTEXT

```
var name = "Ariharasudhan";
var age = 21;

function findProduct(n1, n2){
    var product = n1*n2;
    return product;
}

var p1 = findProduct(2,3);
var p2 = findProduct(8,9);
```

For the snippet given above, following things happen in creation and execution phases.

## In Creation Phase

> name variable is allocated memory and stores undefined

> age variable is allocated memory and stores undefined

> findProduct() function is allocated memory and stores all the code in it

> p1 variable is allocated memory and stores undefined

> p2 variable is allocated memory and stores undefined

## In Execution Phase

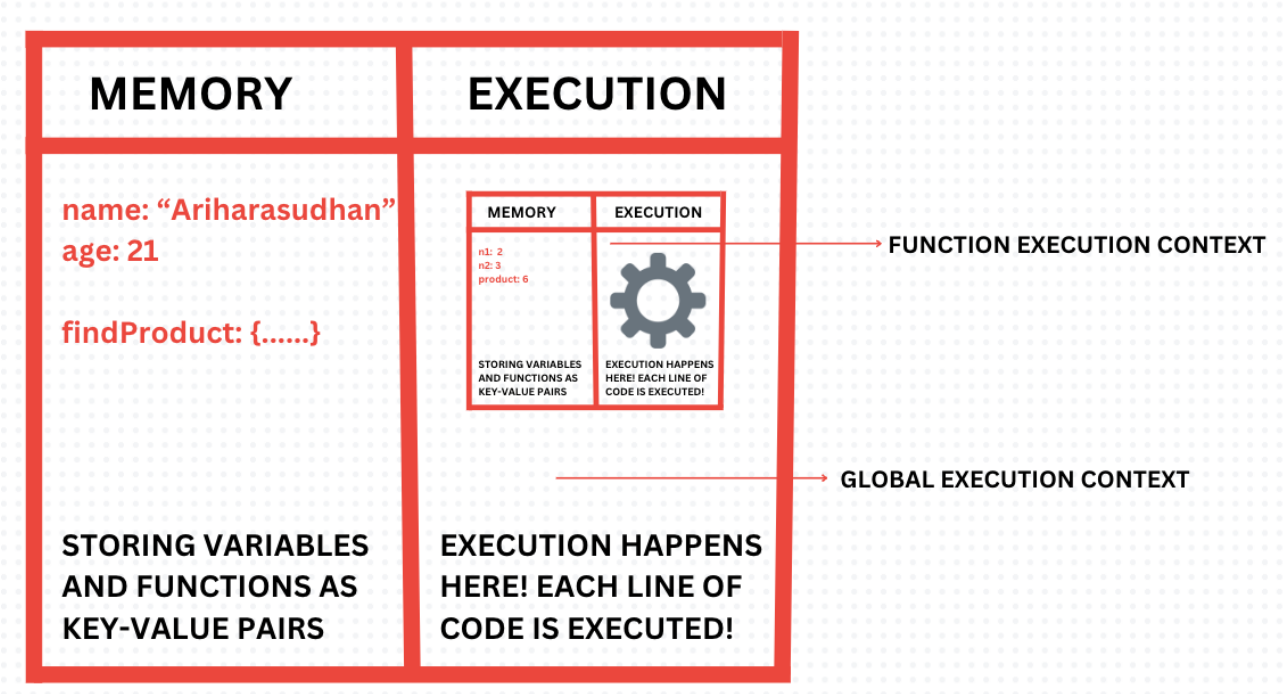
> Places the value, “Ariharasudhan” to name variable

> Places the value, 21 to age variable

> findProduct() is invoked and a new execution context is created for that

> Again, findProduct() is invoked and an execution context is created for that

We can think of it as the following. When there is a function call during the execution phase, it creates an execution context for it as shown below.




Now, we will have the usual Creation Phase and Execution Phase for the newly created Function Execution Context (FEC).

## In Function EC - Creation Phase

>n1 variable is allocated memory and stores undefined

>n2 variable is allocated memory and stores undefined

>product variable is allocated memory and stores undefined

## In Function EC - Execution Phase

> Places the passed value, 2 to n1 variable

> Places the passed value, 3 to n2 variable

> Places the computed value, 6 to product variable

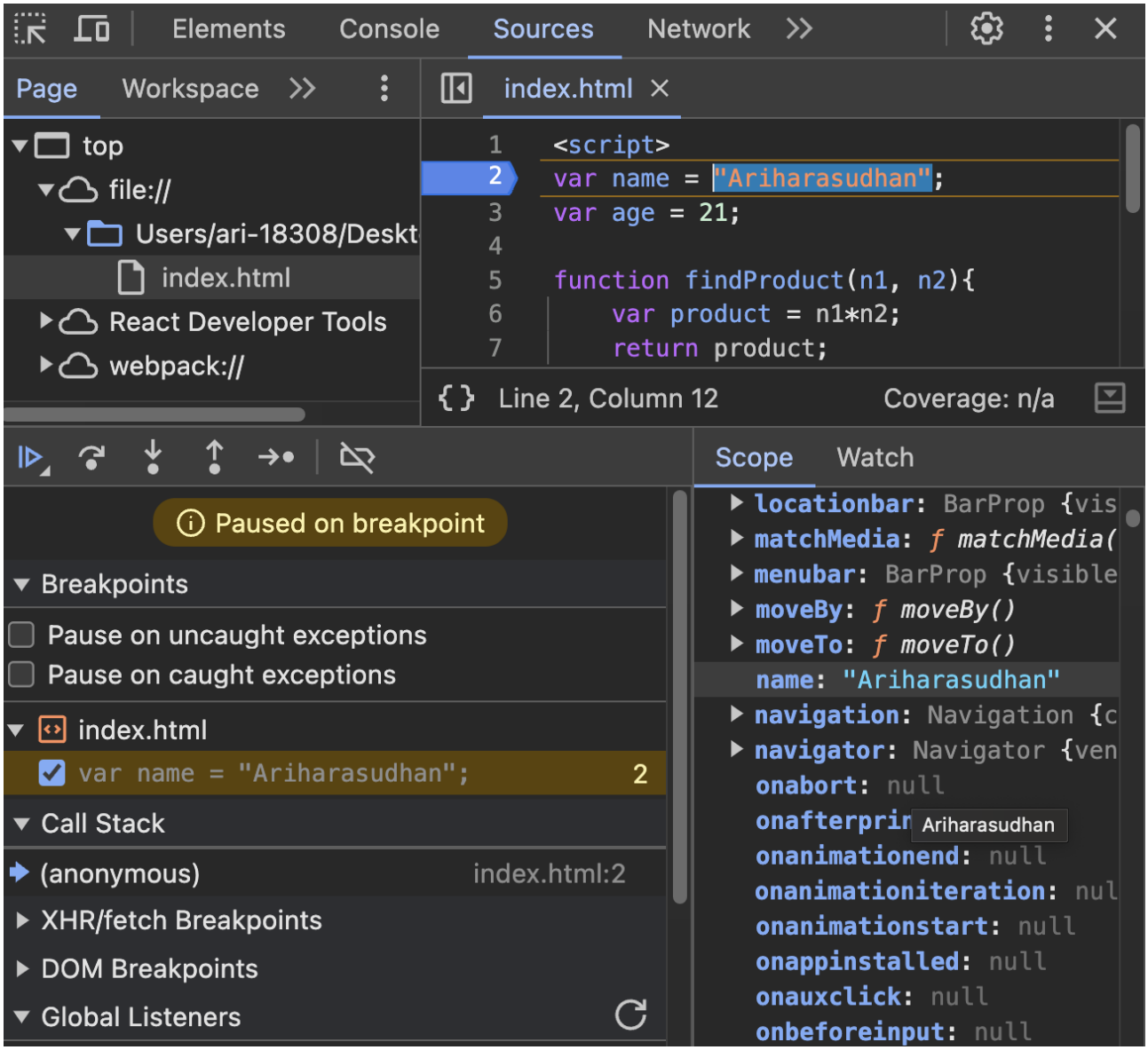
> Return tells the function EC to return to the global EC

> Returned value, 6 is placed to product1 variable

Don't forget that the function, findProduct is pushed into the call-stack which should be popped out right now!

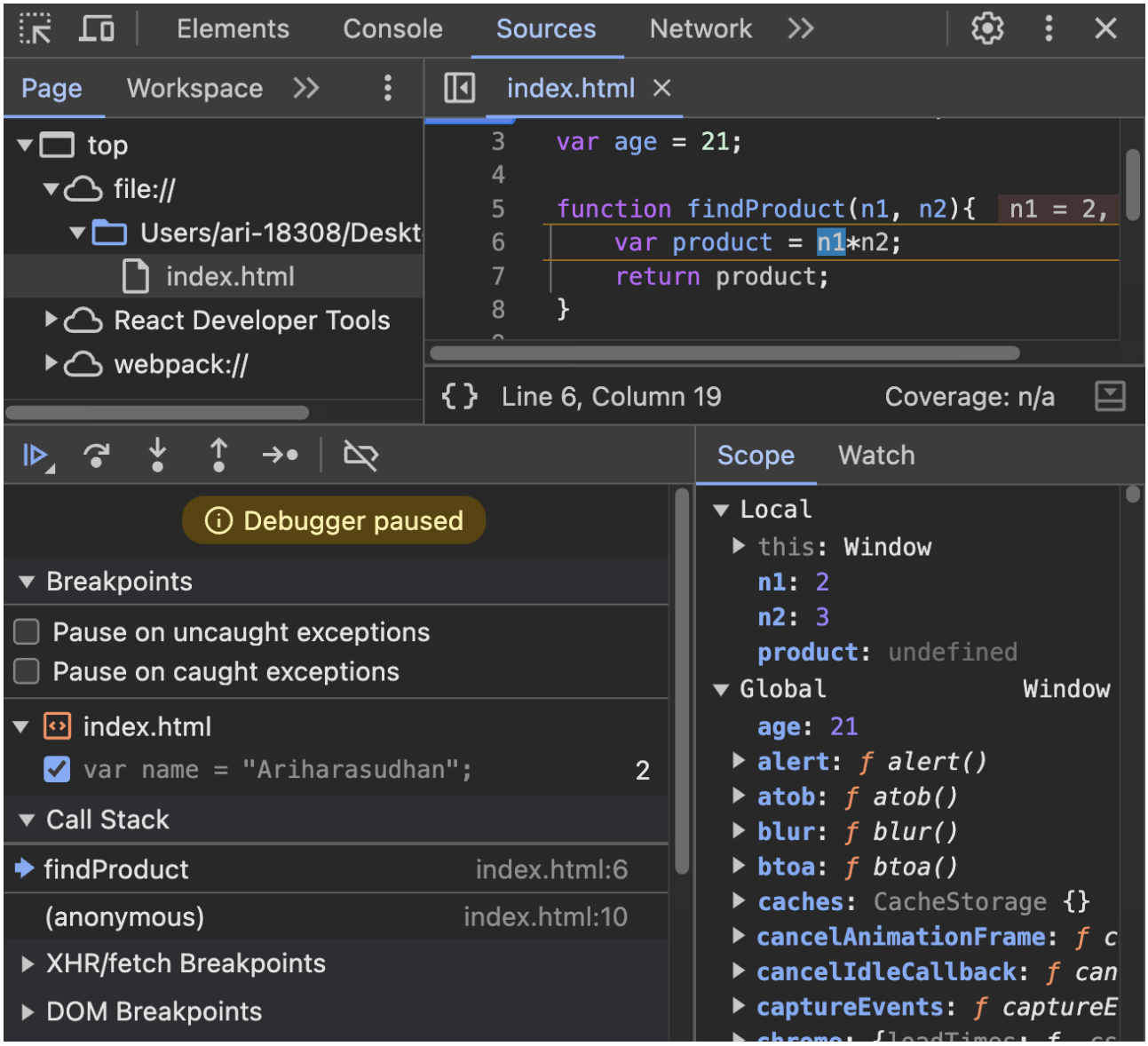

Similarly, the next line of finding product of 8 and 9 and assigning it to p2 is evaluated with creation of new function execution context!

If we execute this script with a breakpoint at first statement,




In the Global Scope, the name variable is given the value, “Ariharasudhan”. But, the age is “undefined” as it is yet to be executed in the line by line flow.

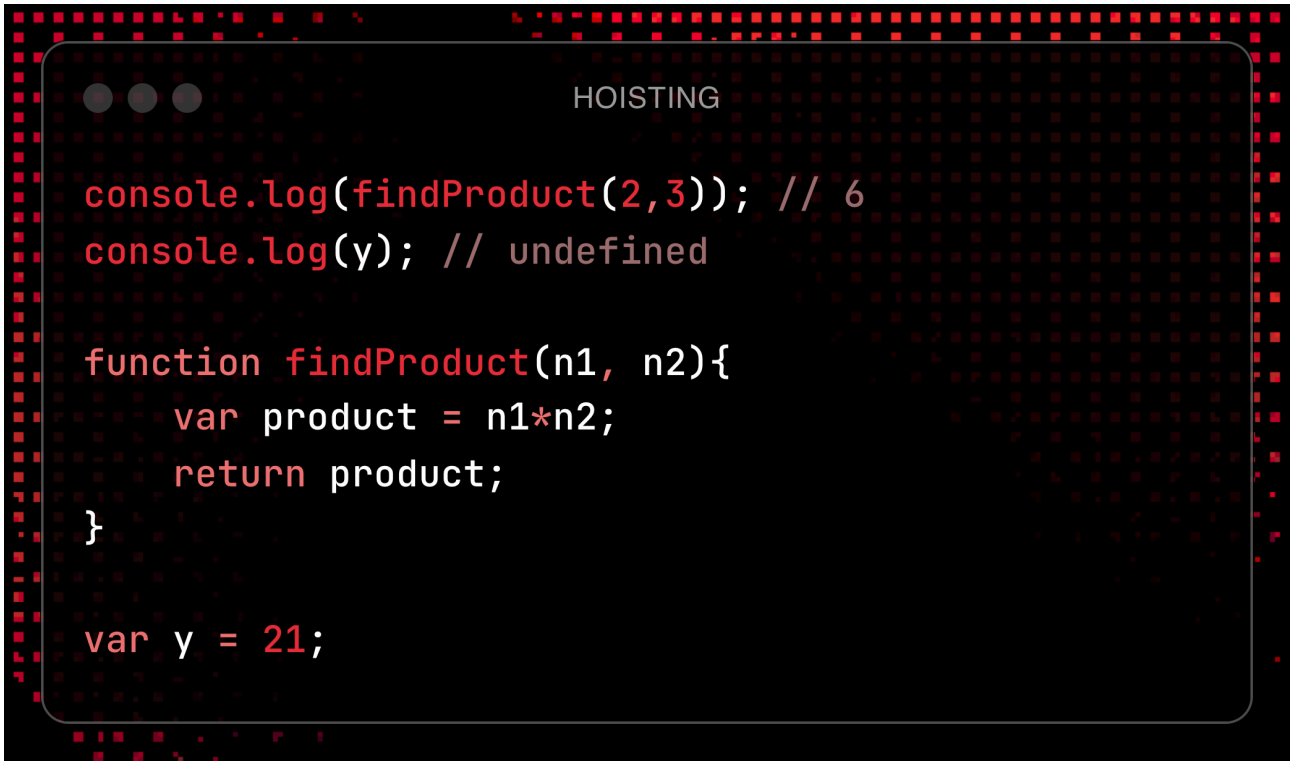
There will be local scope created when we our breakpoint hits the function in.




We can also notice the call-stack for the findProduct function being pushed in.

Now, we know the execution context clearly (Can I believe so?). Let's understand the hoisting clearly right now.

People often define hoisting as, “Interpreter basically, physically moves all the functions and variables to the top the script invisibly so that it can be accessed in lines above”. I meant the following:




Here, the function findProduct() and the variable y is available even before hitting. There is no wicked activities by the interpreter such as moving all of them invisibly to the top of the script. It is all because of the execution context. In Creation Phase, all the variables and functions are stored! In Execution Phase,

all are executed. Since we access the variable that is already stored with undefined value, we get undefined rather than “not defined”. The function is available as whole from the execution context! This is the simple mystery behind the hoisting! What about using const and let?

The hoisting happens also to const and let defined variables. But, they will not be accessible as they fall into the temporal dead-zone after the creation phase. During the execution phase, they will be assigned undefined followed by the value to be assigned. We can check where these let and const defined variables are put into in the elements tab. They are in a separate scope! (Not GLOBAL SCOPE for sure)

Scope Watch

▼ Script

age: <value unavailable>

name: "Ariharasudhan"

▼ Global Window

▶ alert: ƒ alert()

▶ atob: ƒ atob()

▶ blur: ƒ blur()

▶ btoa: ƒ btoa()

# ASYNCHRONOUS JS

Synchronous Execution evidently has the Blocking behaviour. One statement has to wait until the statement above it completes. In NodeJS, we do have blocking methods and non-blocking methods. Following is the blocking code that uses the readFileSync method.

BLOCKING CODE

```
const fs = require('fs');
const filepath = 'text.txt';
const data = fs.readFileSync(filepath, {encoding: 'utf8'});
console.log(data);

let sum = 0;
for(let i=1; i<=10; i++){
    sum = sum + i;
}
console.log('Sum: ', sum);
```

The output will be like,

Hello Hi (Content of File)

Sum: 55

To print the sum, we had to wait until the synchronous (blocking) file reading operation is completed.

Meanwhile, the following snippet describes the usage of a non-blocking file reading method.

NON-BLOCKING CODE

```
const fs = require('fs');
const filepath = 'text.txt';
fs.readFile(filepath, {encoding: 'utf8'}, (err, data)=>{
        console.log(data);
});

let sum = 0;
for(let i=1; i<=10; i++){
    sum = sum + i;
}
console.log('Sum: ', sum);
```

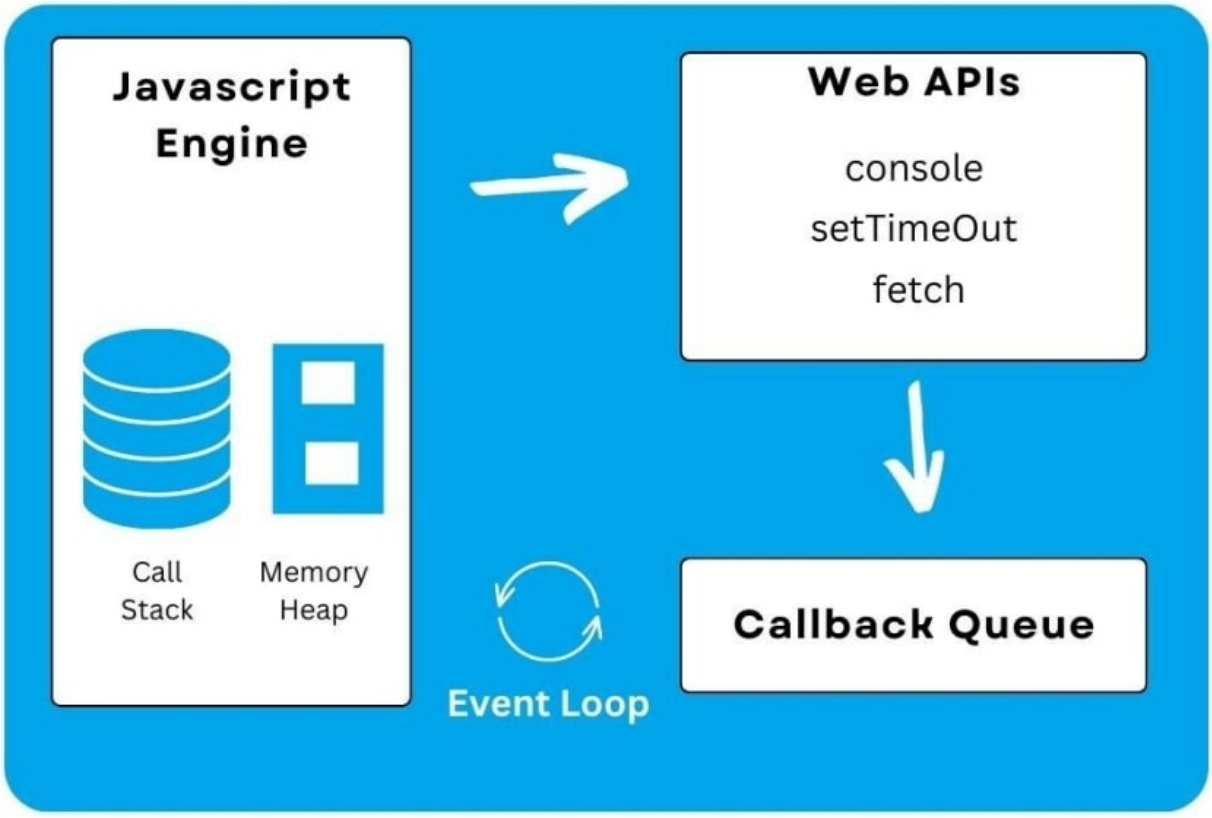
Here, the output will be like:

Sum: 55

Hello Hi (Content of File)

We had not waited until the file reading operation was completed. This is the usage of the non-blocking method, readFile. We can observe the callback function in the readFile. It will be executed when the readFile operation is finished. It happens without disturbing the main thread.

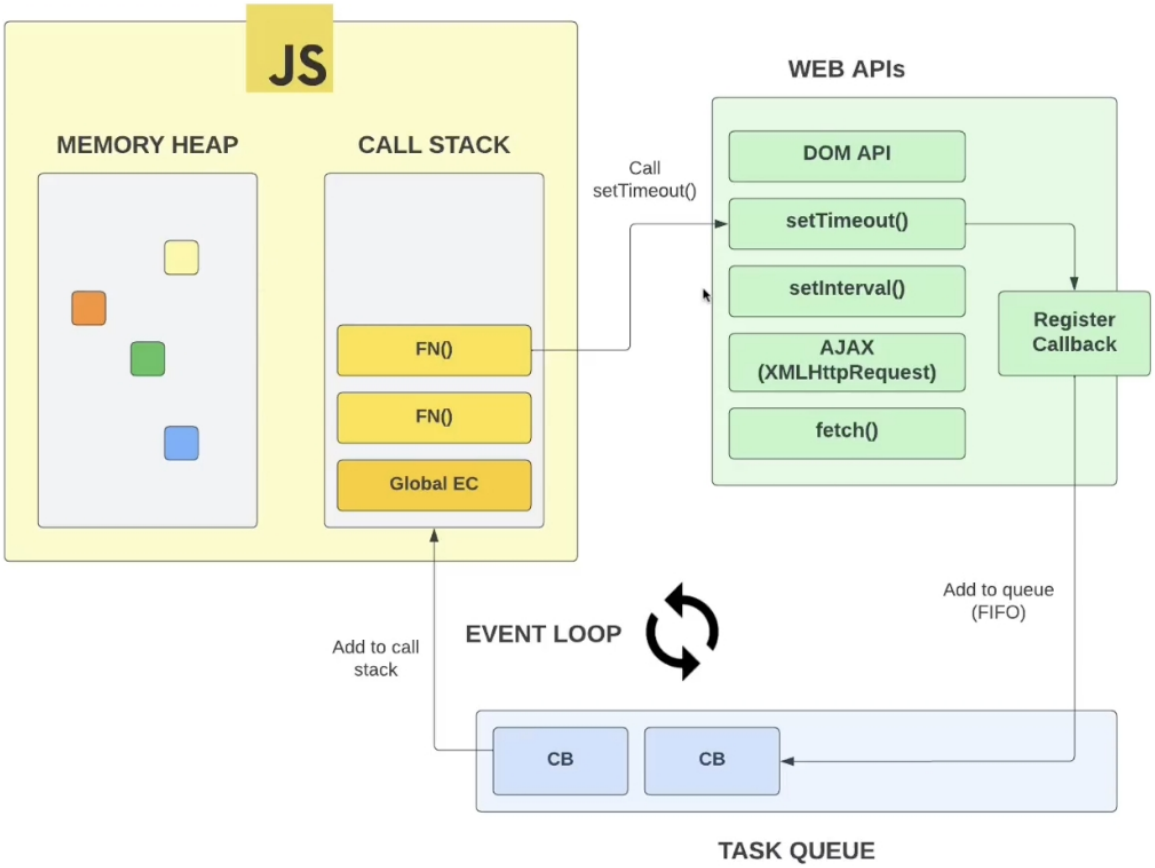
Such asynchronous, non-blocking methods are not available in the browser. But, we can make use of the Web APIs. Web APIs have lot of such non-blocking functions. Following is the something called as JavaScript Runtime Environment.




Javascript Runtime Environment

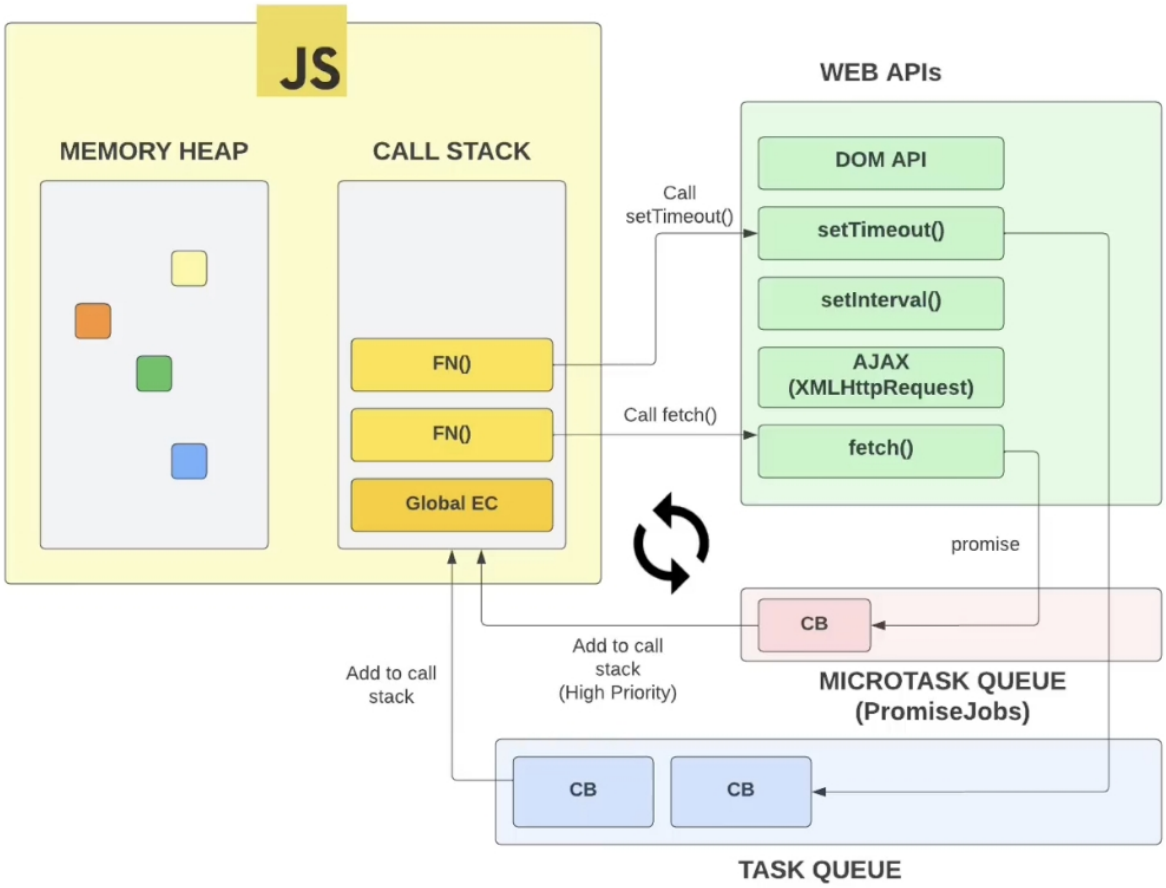
Call-stack and Memory-heap are there in the JavaScript Engine. Web APIs are listed there in there in the right. A Callback Queue is also there. An Event Loop is running!!!!

So, What is this?
We can see another detail image representation.




Callback Queue is also called as Task Queue. In our code, when we have an asynchronous function to be executed, it will be referred there in the Web API. Once it's hit, it will register the callback function we passed and put it into the task/callback queue. Queue follows FIFO principle as we know. The Event Loop continuously runs and checks the call-stack and callback queue.

It puts a callback from callback queue into the call-stack. It is the sole purpose of the event loop. It is case for all such non-blocking Web API powered functions. When we are dealing with Promises, we have to be aware of the MicroTask Queue, A Queue with Higher Priority than The Task Queue.




These Promise Jobs on resolved, will be pushed into the call-stack with priority.

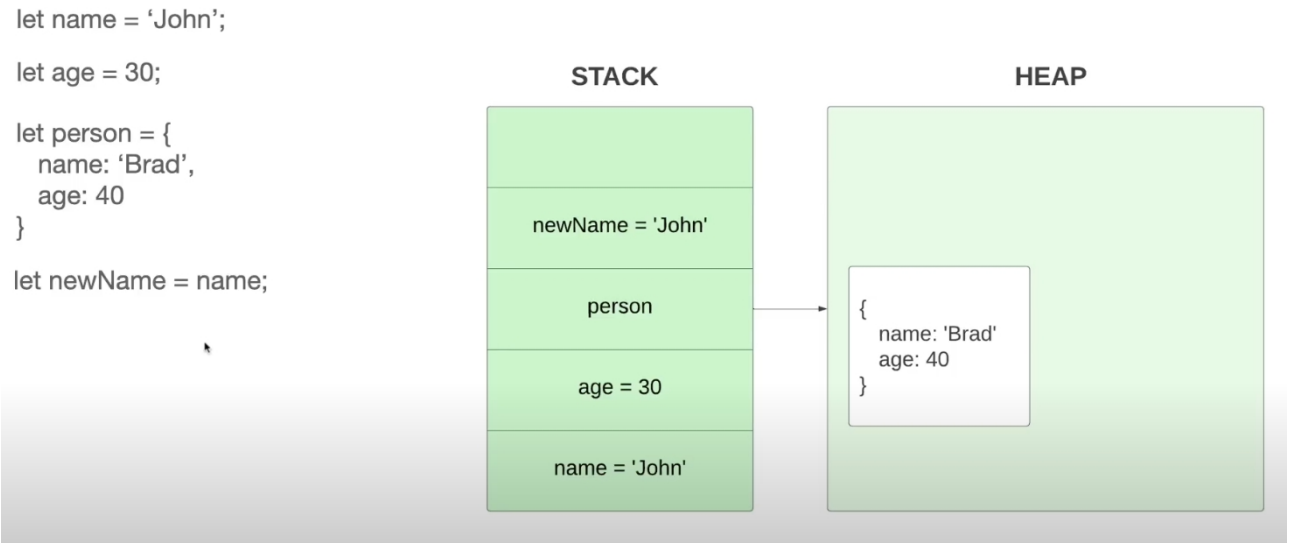
# MEMORY STORAGE

As we have already seen in The JavaScript Engine, there are two stuffs such as Memory-heap and Call Stack to store data. Data can be classified into two categories.

1.PRIMITIVE: String, Number, Boolean, Null, Undefined, Symbol, BigInt

These type of data are stored in the stack and accessed. These are static types. i.e., a fixed size will be allocated for a data of primitive datatype.

In the following figure, name, age, newName are of primitive/static types.



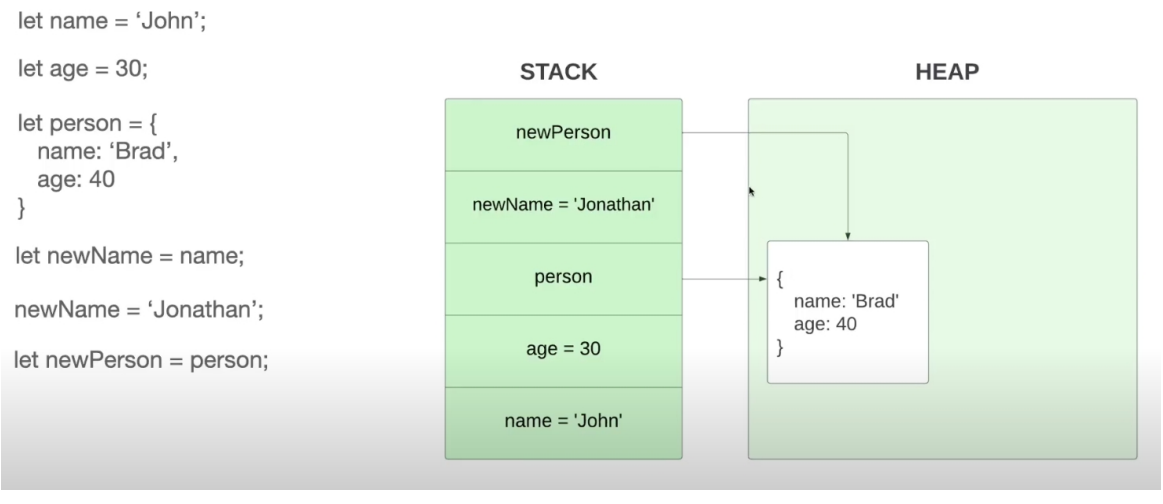

Assigning name to newName won't make both variables to access the same data.

PRIMITIVE TYPES

```
let a = 23;
let b = 34;
console.log(a); // 23
console.log(b); // 34
b = a;
console.log(b); // 23
b = 77;
console.log(a); // 23 UNTOUCHED
console.log(b); // 77 CHANGED
```

## 2.REFERENCE: Array, Function, Object

These are stored in the heap and accessed by reference. For data of these types, space beyond needed will be allocated. These are accessed through references. Here, if we assign a variable to another, it means both will refer the same data and changing one variable reflects in another.

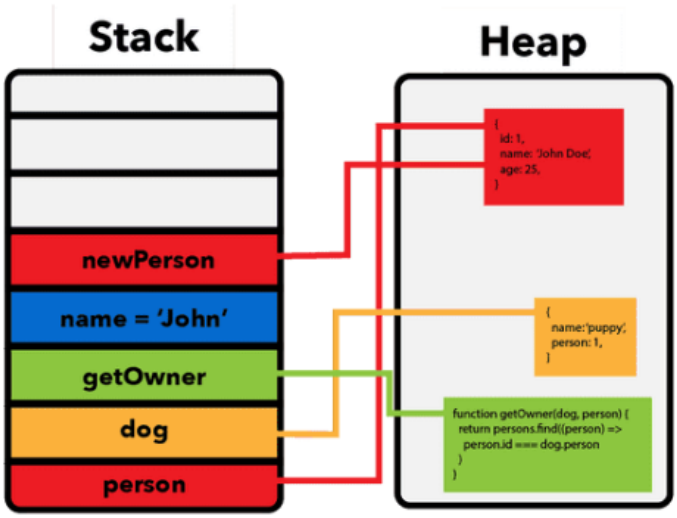


REFERENCE TYPES

```
let a = {
    name: "Ari",
    age: 21
}
let b = a;
b.name = "Aravind";
console.log(a); // { name: 'Aravind', age: 21 }
```

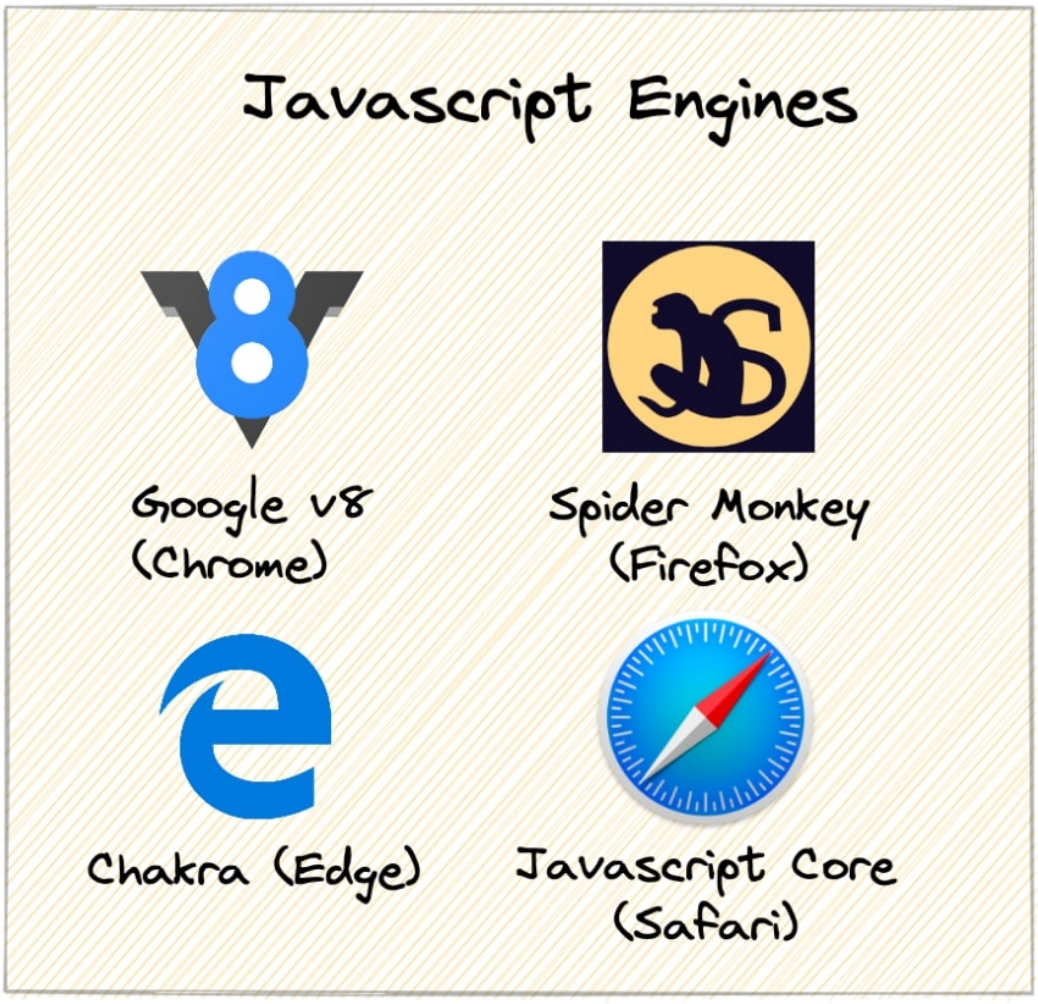
Here is a simple yet appropriate and colourful image of data in stack and heap.

```
const person = {
  id: 1,
  name: 'John',
  age: 25,
}

const dog = {
  name: 'puppy',
  personId: 1,
}

function getOwner(dog, persons) {
  return persons.find((person) =>
    person.id === dog.person
  )
}

const name = 'John';

const newPerson = person;
```

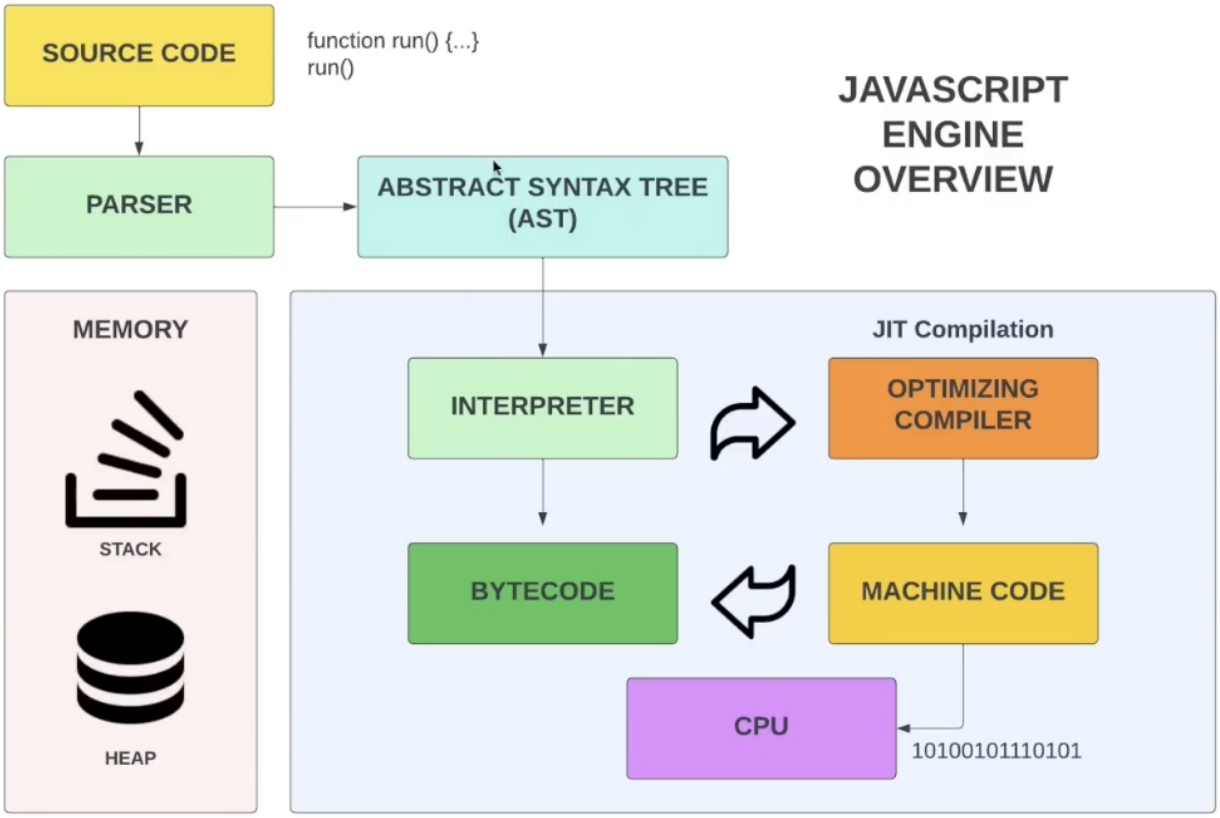


# JAVASCRIPT ENGINES

All browsers have a JavaScript Engine which is a software component in JavaScript Runtime Environment, that optimizes, interprets and executes the JavaScript code. Some modern engines are capable of doing JIT Compilation. Most of the engines are written in Languages like C, C++. V8 Engine is the one used by Chrome, NodeJs. Firefox uses SpiderMonkey.



JavaScript Engine can be defined abstractly as an interpreter of JavaScript.




It follows the usual transpilation flow of execution.

MERCI