Namaste JavaScript

Episode 1- Introduction

- Everything in a JS happens inside an ***execution context***.
- ***Execution context:*** like a big box or container where whole the whole JS code is executed.
  It looks like the following: it has 2 components; memory component- ***Variable Environment*** (variables and functions are stored in key-value pairs) and Code component -***Thread of execution*** (where the codes are executed line by line)

Memory (*Variable Environment)*	**Code (*Thread of execution)***
Key: value	
a :10 fn1:{……}	
…… **…….**	

So, JS is a ***synchronous*** and ***single-threaded*** language.

Codes are executed sequentially and JS's compiler will move to the next line one by one once the current line has been executed.

Episode 2 – What happens when you run JS code?

When you run a program in JS, an execution context is created.
Steps during execution of a JS program;

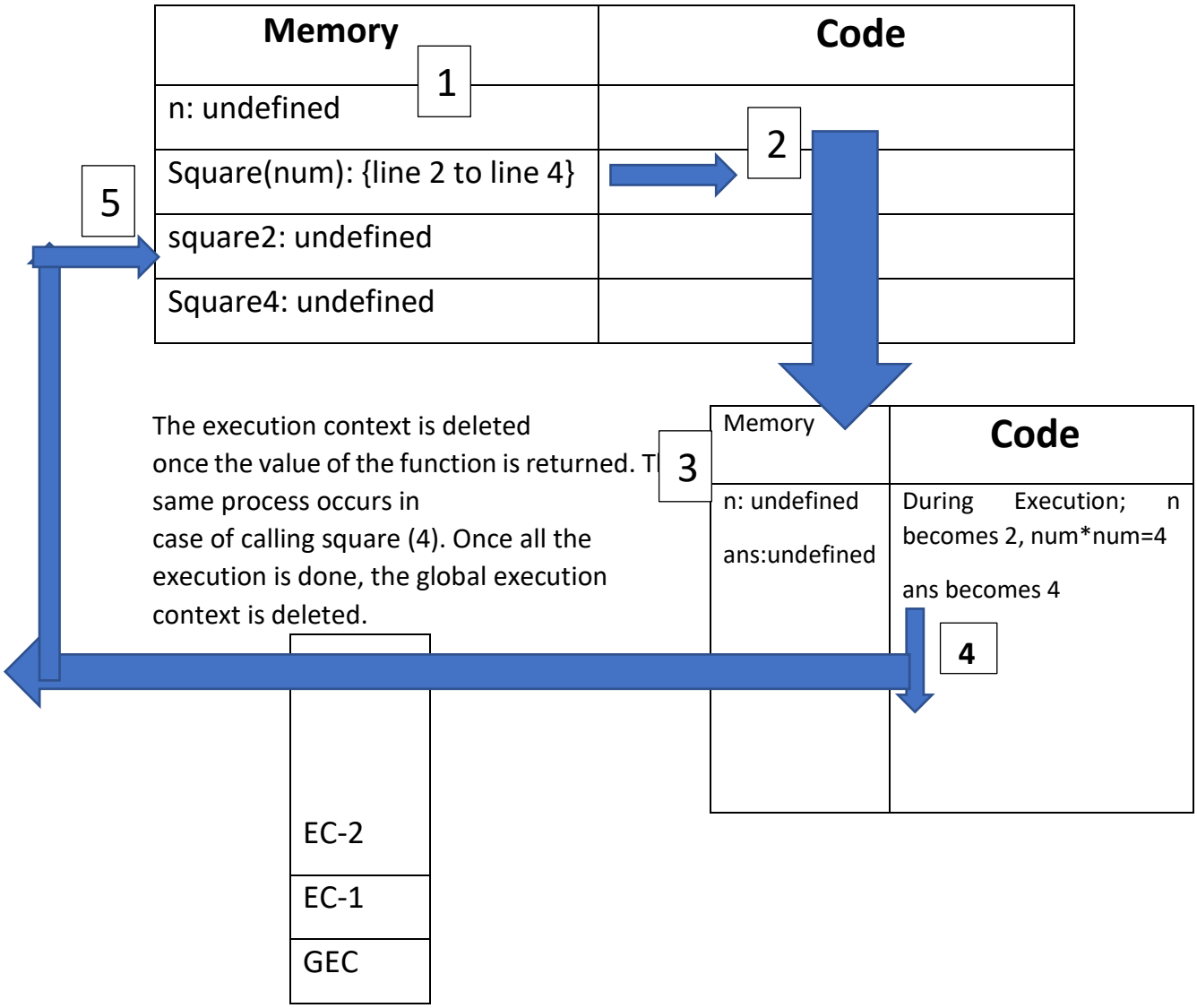
Step 1 – Memory Allocation Phase
Step 2 – Code Execution Phase

Ex. Of a JS code →

```
1   var n=2;
2   function square(num){
3       var ans = num*num;
4       return ans;
```

```
5   }
6    var square2 = square(n);
7   var square4 = square(4);
```

Step 1: JS skims through the whole program and allocates memory to each variable and function. It assigns ***Undefined*** to variables; placeholder, a special keyword in JS and in case of functions, it copies the whole code within its body.



In case of many functions within a function, JS does it very beautifully. It does it by a **call stack** which holds GEC (Global Execution Context) at the bottom of it. Once the work of E1 is over, it is popped out of the stack and the control moves to the GEC.

Call stack is created for managing (creation and deletion) the execution context. The call stack gets empty, once the entire execution is done. So, ***Call stack maintains the order of execution of the execution contexts.***

Call Stack is also known **as**

1. **Execution context stack**
2. Program Stack

3. Control Stack
4. Runtime Stack
5. Machine Stack

Episode 3 – Hoisting in JS (variables and functions)

Hoisting is a concept in JS where you can access the values of variables and functions even before initialising them. There'll be no error in this case. [this can be referred to Ep.1 for reference regarding the concept of memory.]

If the value of a variable is not given at all, during the execution JS will throw a reference Error.

A function behaves as a variable in the following examples and initially;

1. var getName = () => {

   }

2. var gerName2 = function () {

   }

Episode 4 -Introduction – Function Invocation and variable environment in JS

```
1. var x = 1;
2.  a();
3.  b();
4. Console.log(x);
5. Function a()  {
6. var x = 10;
7. console.log(x);
8.   }
9. Function b()  {
10.  var x =100;
11. console.log(x);
12.    }
```

Output: 10 100 1 [follow the context of execution]

Episode 5 - Introduction to the shortest JS program; Window and this keyword

Shortest JS program → empty file

this → global object(window) [this ===window; output: true]

Window is a global object which is created along with the Global Execution context along with 'this'. All JS engines create the window object. In case of browser, it is known as window.

Global Space → variable not inside any function [By default, JS compiler looks for variables in global space first]

So, variables and functions in global space are objects of window.

```
1. var a = 10;
2. function b() {
3.    var x = 10;
4.   }
5. Console.log(window.a);
6. Console.log(a);
7. Console.log(this.a);
```

output: 10

10

10

Episode 6 – Undefined Vs not defined in JS

JS is a loosely-typed (weakly typed) language. A variable has flexibility of storing values of different data-type.

```
1. var a;
2. console.log(a);
3. a =10;
4. console.log(a);
5. a = "Hello World!";
6. console.log(a);
```

Output:
Undefined
10
Hello World

It's not advisable to assign "Undefined" to any variable.

Episode 7 – The scope chain, scope and Lexical Environment

```
1.function a()  {
2.console.log(b);
3.}
4.Var b=10;
5.a();
```

output - 10

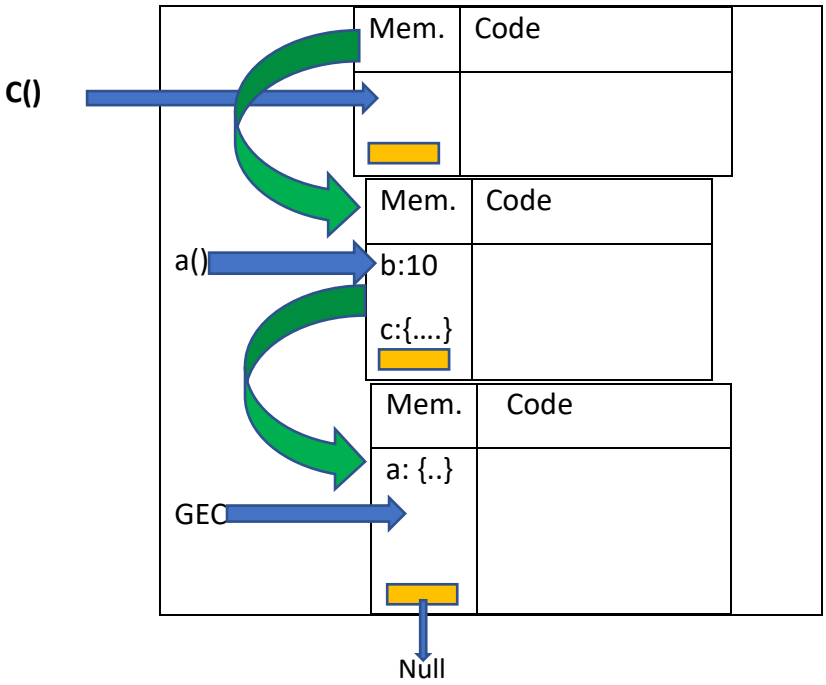
```
1.  function a()  {
2.   c() ;
3.      function c() {
4.           console.log(b);
5.       }
6.               }
7.Var b=10;
8.a();
```

Output :10

Whenever an Execution context is created, a ***lexical environment*** is also created.

Lexical Environment = local memory + lexical environment of its parent

Lexical means in hierarchy, in a sequence. In the above code, function c() is lexically present within function a() and a() is lexically present in the global scope.



Scope Chain: The way of finding variable in previous lexical environments is called ***scope chain***.

Episode 8 – Let and const in JS and Temporal Dead Zone

Let and const. declarations are hoisted but their way of hoisting is very different than of the declaration of var.

Eg.1

1. console.log(b);
2. console.log(a);
3. let a = 10;
4. var b =100;

output:
undefined
Uncaught Reference Error: cannot access ‘a’ before initialisation

Eg.2
let a = 10;
console.log(a);
output: 10

Unlike in var, let and const. are assigned memory by the compiler initially (they are now storing ‘undefined’) but they are not global objects. They are not stored in the scope of GEC but their scope (called script) is different. So, they cannot be accessed before initialisation.

Temporal dead zone is the period of time between hoisting a variable with let data type and assigning value to it. Refer to Eg.1, when the control goes from line 2 to line 3, the variable 'a' loses its temporal dead zone. The variable with data type const. also goes through temporal deadzone.

Whenever we try to access a variable in ***temporal dead zone***, it gives ***Reference Error***. This type of error also occurs in the case when the we try to access a variable nowhere defined in the program.

Any variable in the global scope (with data type var) can be accessed through this command; ***window.object*** or ***this.b.***

Assigning value to an already initialised variable gives ***syntax error***.

Eg. 3

console.log ("An example of syntax error")

let a = 100;

let a=10;

output: Uncaught Syntax Error; identifier 'a' has already been declared.

JS Engine does not even execute the first line of the code.

Eg. 4

let a = 10;

var a =100;

output: Syntax Error

(because can't redeclare an already exiting variable.)

Eg. 5

var a = 10;

var a =100;

No error

Eg. 5

const b;

const b = 1000;

In the console: **Syntax Error;** missing initializer in const declaration

Eg. 6

const b = 100;

const b = 1000;

In the console: **Type Error:** Assignment to a constant variable

Const variable should be assigned a value during the time of declaration, otherwise the compiler will throw a ***Syntax Error***. If a const. variable is reassigned the value(case duplicate declaration even with variable having let datatype), then the compiler throws a ***Type Error***. Accessing variables even before their declaration(in case of let data type) , JS engine throws a ***Reference Error.*** This error also occurs in the case when the variable is nowhere declared in the program.

Use const. whenever you want to assign value to a variable which will not change its value. Const. data type is the most-strict of other data types used in JS. Then the preferred data type is Let.

To avoid unnecessary errors, it is advisable to put initialisation and declaration at the top of the program. So, it helps in shrinking of the temporal dead zone.

Level of strictness: var < let< const.

Summary:

S.No	Data Type	Re-declare (in same scope)	Re-initialise	Temporal Deadzone	Memory
1	var	Yes	Yes	No	Global
2	let	No(Syntax Err.)	Yes	Yes	Separate(Ref. Error)
3	const	No(Syntax Err.)	No(Type Error)	Yes	Separate (Ref. Error)

When trying to access them before initialization,

Episode 9 – Block scope and shadowing in JS

Blocks are area where codes are enclosed in curly {} brackets. They are also called ***compound statements.*** Its purpose is to group multiple statements at one place where JS engine expects one statement.

Block Scope means the variables and functions that can be accessed inside a block. It also follows lexical scope.

Eg. 1:

```
{var = 10;

let b =20;

const c = 30;

}
```

In the above example,

Scope	Variables
Global	a: undefined
Block	b, c: undefined

So, let and const are block scoped as they can't be accessed outside of the block where they are declared.

Eg. 2:

```
1.{var = 10;

2.let b =20;

3.const c = 30;

4.}

5.console.log(a);

6.console.log(b);

7.console.log(c);
```

Output: 10

Reference Error

The execution stops here as the compiler encounters an error in line 6.

Eg.3:

```
var a=100;

{var a = 10;

let b =20;

const c = 30;

console.log(a);

}
```

```
console.log(a);
```

output: 10

10

In case of variables with ***same name*** and ***data type***, when we write a print statement inside/outside the block, the variable present in the block shadows(and modifies) its global counterpart and prints the value present in block. This concept is called ***shadowing***. This occurs because variable 'a' is present in the global scope not in the block scope.

Eg.4

```
var b=100;                    global scope
const c = 120;                separate memory (script)
{var a = 10;
let b =20;                    block scope
const c = 30;
console.log(b);
console.log(c);
}
console.log(b);
console.log(c);
```

output: 20

30

20

120

As soon as the control goes out of the block scope, its memory allocation in call stack is destroyed and the values either in global scope / script space ***shadow*** the values in block scope. This concept is still valid in case of functions.

Eg.5 (a)

```
let a = 10;
```

(b)

```
let a =10;
```

```
{                                                    function x() {
var a=20;                                                  var a =20;
                                                            }
                                                       console.log(a);
}
console.log(a);
```

Output: a) syntax error , b) no error

For a) This is called ***illegal shadowing***.

Eg.6

```
var a = 10;
{
let a=20;
}
console.log(a);
```

This is valid!!

Eg.7

```
const a = 10;
{
const a= 200;
{
console.log(a);
}
}
```

Output: 200

As a is not present in the scope, the compiler will search for it in its immediate adjacent scope and on finding it, prints the required value.

Episode 10 – Closures in JS

Eg.1

```
function x() {
        var a =7;
          function y()
           {console.log(a);}
        y();
}
x();
```

output: 7

Closure: Function with its lexical environment or function bundled with its lexical scope.

In JS, you can pass a function as argument to another function. You can also return a function out of another function.

When you return a function, closure is returned i.e. function with its lexical scope is returned.

Whenever function is returned, even if it's vanished in execution context but still it remembers the reference it was pointing to. It's not just that function alone it returns but the entire closure

Each and every variable or function in JS has an access to its outer lexical environment/environment of its parents. If a function is executed in another scope, it still remembers its lexical environment or scope of parents.

Eg.2

```
function x() {
        var a =7;
          function y()
           {console.log(a);}
        y();
}
var z = x();
console.log(z);
z();
```

Uses of closure:

1. Module Design Pattern
2. Currying
3. Function like once
4. Memoize
5. Maintaining state in async world
6. setTimeouts
7. Iterators
8. And many more……

Episode 11 – setTimeout+closure interview Qs

Eg.1:

```
1. function x() {
2.    var i =1;
3.    setTimeout(function x() {
4.    console.log(i);
5. },1000);
6. }
7. x ();
```

output: 1 (after 1 second)

Eg.2:

```
1. function x() {
2.    var i =1;
3.    setTimeout(function x() {
4.    console.log(i);
5. },3000);
6. console.log("Namaste JS");
7. }
8. x ();
```

Output: Namaste JS (instantly)

1 (after 3000ms)

Function x() has a closure which has a reference to i. It recalls the scope of i. JS Engine stores the function x() in a different location and continues with the Timer.

Eg.3: To print 1 to 5 with a delay of 1second among each

```
1. function x() {
2.    for (var=i ; i<=5 ; i++)
3.    { setTimeout (function()  {
4.    console.log(i);
5. }, i* 1000);
6. console.log("Namaste JS");
7. }
8. x ();
```

Time delay (in ms)

Output: Namaste JS

6

6

6

6

6

[Prints 6 for 5 times with a delay of 1 second each]

Even after the function is executed inside the block, because of its closure it remembers the reference to 'i' not the value of 'i'. When the loop runs for the first time, it creates a copy of the function, attaches the timer and remembers the reference to 'i' and this process is repeated for 5 times and each time it is referred to the same variable 'i'. All these 5 copies point to same reference of i.

JS Engine does not wait for the setTimer to get expired. Instead of that, it prints Namaste JS and other values after 'i' has been updated to 6 in last 5 iterations.

In order to fix this issue, use 'let' in place of 'var'. 'Let' has a loop scope. So, in every iteration, 'i' appears as a new variable. Every time the function is called, setTimer function has a new copy of 'i'.

Modification in above program;

```
1. function x() {
2.    for (let i =1; i<=5; i++)
3.    {setTimeout (function()  {
4.    console.log(i);
5. }, i* 1000);
6. console.log("Namaste JS");
7. }
8. x ();
```

Time delay (in ms)

so, ‘let’ creates a new copy every time the loop is executed.

Modification with ‘var’; create a function to provide closure to ‘i’ so that every time a new copy of ‘i’ can be used.

```
1.function x() {
3.     for (var i=1; i<=5; i++)
4.     {  function close(x) {
5.          setTimeout (function()  {
6.     console.log(x);
7.  }, x* 1000);
8.  }
9.  close(i);
10. }
11. console.log(“Namaste JS”);
12. }
13. x ();
```

Time delay (in ms)

Episode 12 – JS interview feat. Closures

Example of closure()

```
1.  function outer() {
2.    var a =10;
3.     function inner() {
4.     console.log(a);
5.       }
6.    return inner();
7.  }
8.  outer() () ;
```

output :10

function inner() has access to its outer environment. So, the function inner() along with its outer environment /environment of its parent is called closure.

Syntax → outer () (); used for calling the inner function

Suppose we write outer() ; → it returns the inner() , so on writing outer() () → inner is called in a single line

Alternative method;

var close = outer();

close(); -→ calling the inner function

CASE 1: if the outer() is nested inside another function , inner will still have access to that function because of closure.

Advantages of closure:

i) It is used in module pattern, function currying.
ii) Used in higher order functions like Memois
iii) Helps in data hiding and encapsulation

Example of data hiding or privacy using closure;

```
1. var count=0;
2. function incrementCounter () {
3. count++;
4. }
```

Problem with the above code is that anybody can access count and modify it.

So, the solution is to wrap up the above codes inside a function, and anybody accessing it from the outside will have to face an error. So, count has become private here with the help of the function.

```
1. function counter() {
2. var count=0;
3. function incrementCounter () {
4. count++;
5.  }
6. }
7. console.log(count); → Reference Error
```

To access count variable;

var counter1 = counter();

counter1(); → output: 1

counter1(); → output: 2

var counter2 = counter(); → another counter which will start incrementing from 0.

counter2() ; → output:1

For increment and decrement of counter, constructor can be used.

This can be done by

```
1. function counter() {
2. var count=0;
3. this.incrementCounter= function() {
4. count++;
5. console.log(count);
6.   }
7. this.decrementCounter= function() {
8. count--;
9. console.log(count);
10.  }
11. }
12. var counter1= new counter();  -→ create a new counter with the constructor function
13. counter1.incrementCounter();  -→ 1
14. counter1.incrementCounter();  → 2
15. counter1.decrementCounter(); → 1
```

Disadvantages of closure:
i)There can be ***over consumption of memory*** because those variables are not garbage collected when the program expires.
ii) If not handled properly, it can lead to ***memory leaks***.

Garbage Collector is a program in the browser or JS Engine which frees up the unutilised memory.

Episode 13– First Class Function feat. Anonymous Function

Function statement/Function declaration;

Eg.1:

```
function a() {
  console.log(“a is called);
}
a();
```

output: a is called

Function expression;

Eg. 2:

```
var b = function  () {
```

```
  console.log(“b is called);
}
b();
```

output: b is called

If you use let or const. in place of var, it will behave exactly in the way any variable of let or const. behave.

Function statement and function expression are used to create a function. The difference between these two is hoisting. Now, we want to check the behaviour of these functions in hoisting;

Eg. 1.a

```
a();
function a() {
  console.log(“a is called);
}
```

output: a is called

Eg.2.a:

```
b();
var b = function () {
  console.log(“b is called);
}
```

output: type error

In this case; b is assigned “Undefined” initially until it hits the function.

Anonymous Function:

Eg.3 :

```
Function  ()  {


}
```

A function without a name is called an anonymous function. Eg.3 will give syntax error as it does not have a name as it looks like a function statement/declaration.

Anonymous functions are used where they are used as values. So, it has be used in function expression.

Named Function Expression:

Eg. 4:

```
var b = function xyz() {
  console.log("b is called);
}
xyz();
```

output: Reference Error

Eg. 4(a)

```
var b = function xyz(){
      console.log(xyz());
        }
output:  xyz() {
 console.log(xyz()); }
```

When you give name to an anonymous function, it becomes named function expression.

In the above program it gives error because xyz() has not been defined in the outer scope.

In 4(a), xyz() can be accessed inside a local scope.

Difference between parameter and arguments:

```
var b = function xyz(param1 , param 2) {          // Parameters [ in function expression]
  console.log("b is called);
}

b(1,2) ;                                          // Arguments
```

First class functions (Aka, First class citizen)

The ability of the functions to be used as values when it is passed as an argument to another function is the concept of first-class function. You can return a function in another function.

Episode 14– Callback functions in JS

Callback functions

Functions are passed as arguments to another function. These are called the first-hand functions.

The function which is passed into another function is called a call back function.

Eg.1

```
setTimeout(function() {
  console.log("timer");},5000);
Function x(y){
console.log("x");
y();
}
X(function y() {
 Console.log("y");s
……})      // y() is a callback function
```

Output –

x

y

Timer(after 5000 milli seconds)

Using APIs and callback functions, asynchronous operations can be achieved.

Advantages of callback functions

- It gives access to whole synchronous world from a synchronous threaded language.

Any operation blocking the call stack is known as blocking the main thread.

```
Document.getElementById("Click Me").addEventListener("Click" , function xyz() {

  console.log("Button clicked");
} )
```

When the event occurs, it will call the callback function (which is stored somewhere else,) and the callback function appears in the call stack.

Function to show the number of times the pushbutton has been pressed on.

```
Function attachEventListeners(){

}
let count =0;
Document.getElementById("Click Me").addEventListener("Click" , function xyz() {

  console.log("Button clicked",++count);
```

```
} )

attachEventListener ();
```

In this case, the callback function forms a closure with the outside environment.

Why is the need to remove EventListeners?

Ans: EventListeners are heavy. They take memory and form closure with the functions

even when the call stack is empty. So, we can not free up the extra memory. So, Event Listeners should be removed when they're not in use.

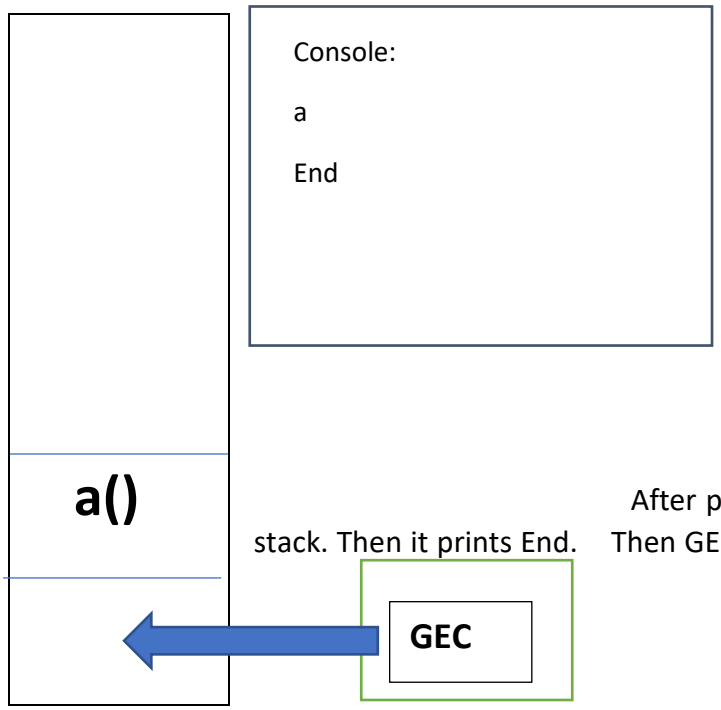
On removing the EventListeners, all the values/functions held by it will be garbage collected.

Episode 15– Asynchronous in JS and Event Loop from scratch

JS is a synchrounous ,single threaded language as it has a call stack and it can do one thing at a time.

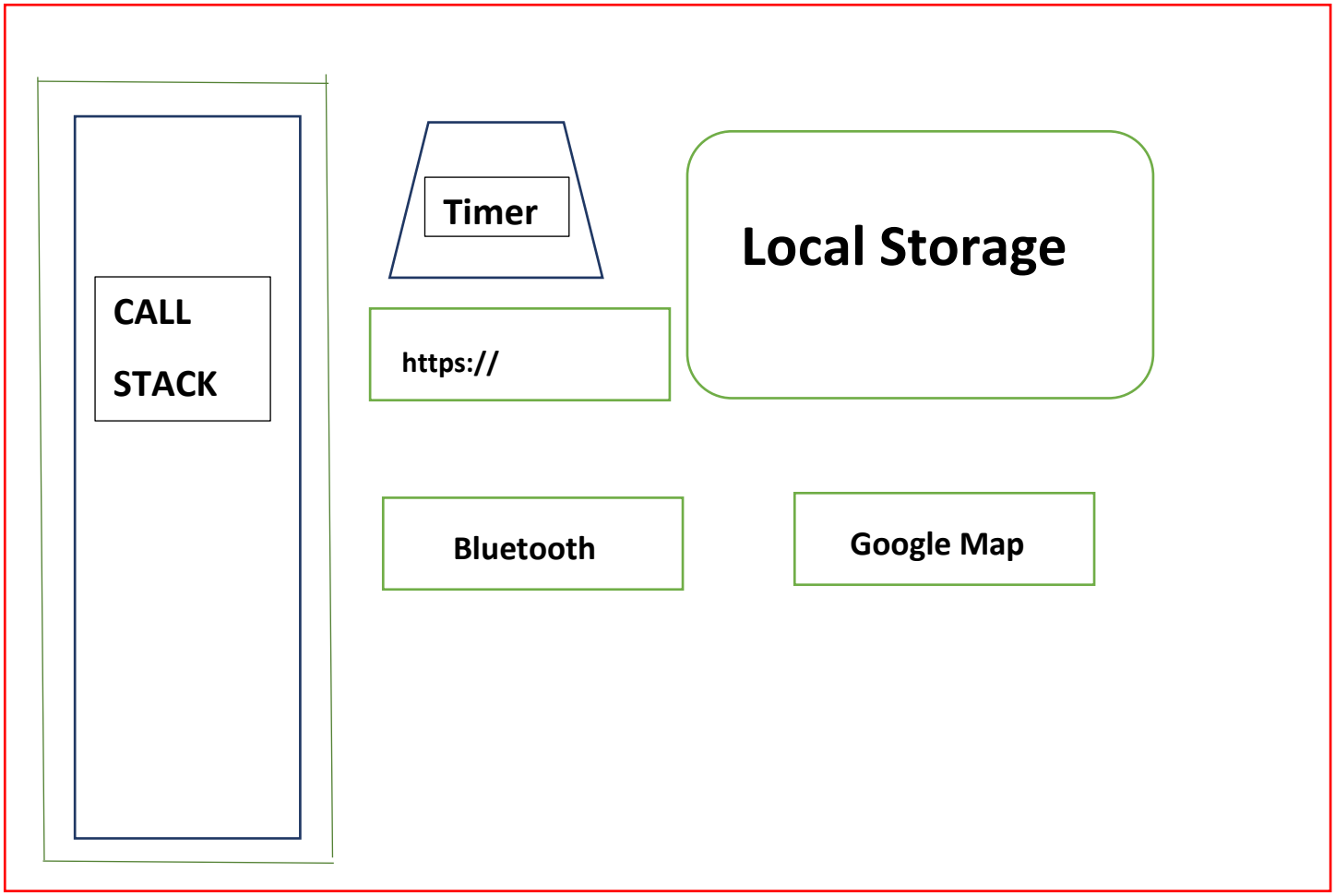
Eg.1

```
Function a() {

Function.log("a");

}

a();

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

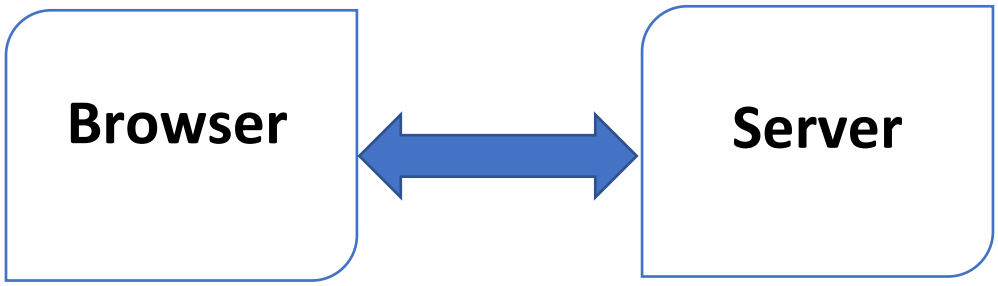


After printing a, it deletes it from the call stack. Then it prints End. Then GEC is popped up from the call stack.

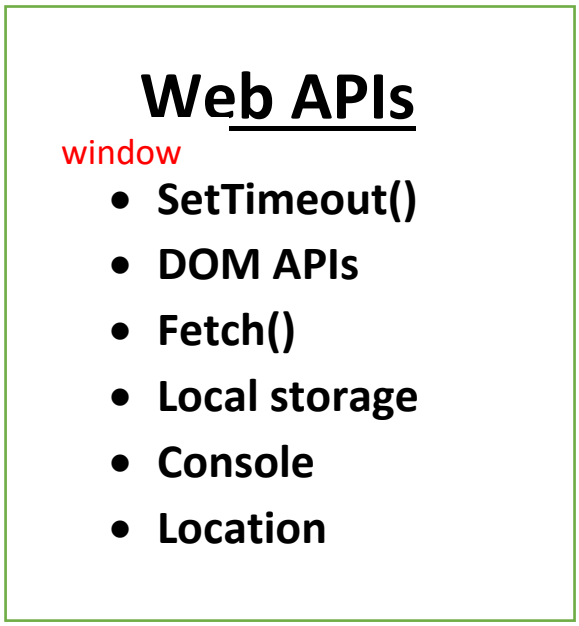
As soon as anything gets into the call stack, it quickly executes it without waiting for anything as it does not have a timer.



Red Box: Browser, Green Box: JS Engine, Blue Box: Call stack



To access all the functionalities of a server, JS Engine needs Web APIs.



Web APIs are parts of browsers not JS Engine. Browser gives access to all these functionalities inside JS Engine through **Window** which is a global object.

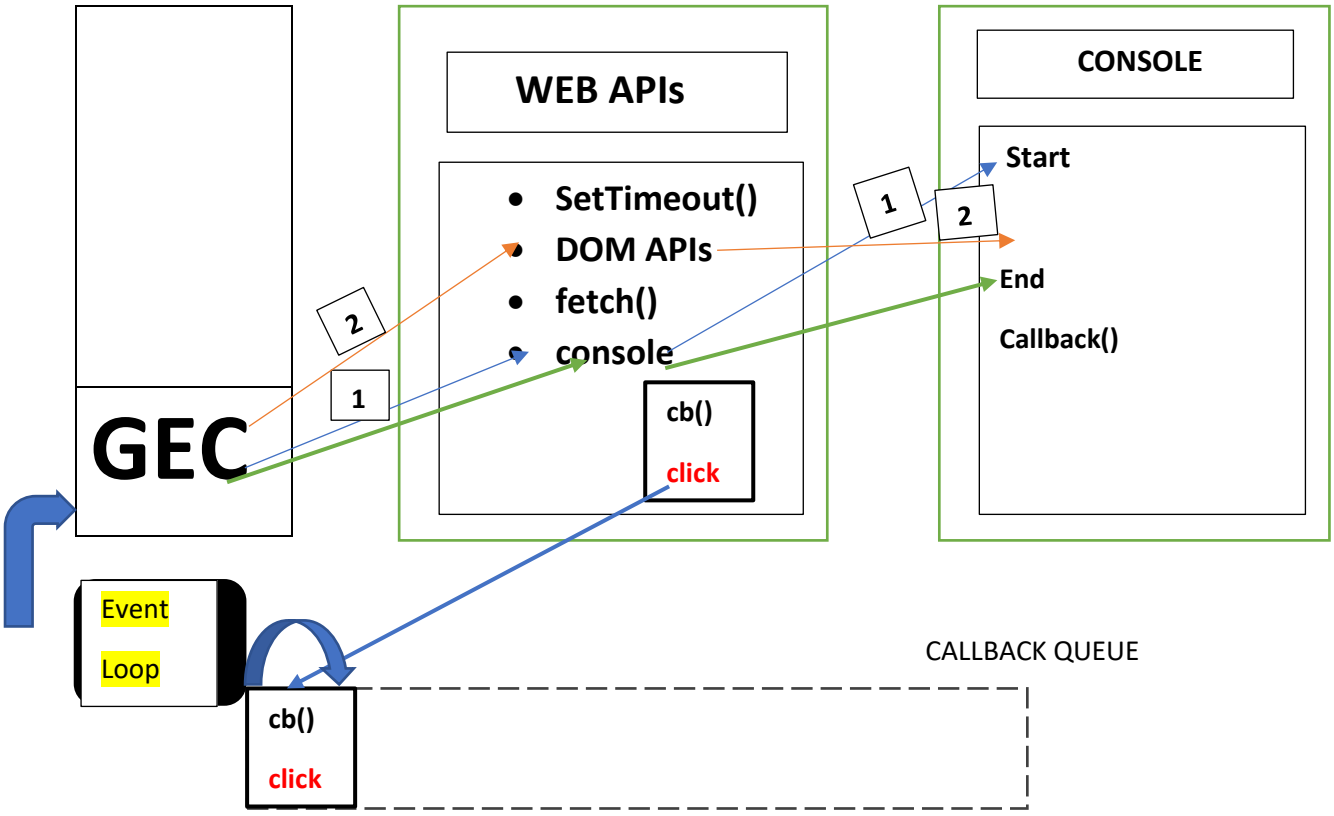
To access all these functionalities inside the JS code, write in window. format but it's not mandatory. Format -> For Example, window.setTimeout() , etc.

Call back function, after the event completion, gets into the callback queue. Event loop checks the call back queue and puts the call back function into the call stack.

How Event listeners work in JS?

1. Console.log("Start");
2. document.getElementById("btn").addEventListener("click",function cb() { console.log("callback")});
3. console.log("End");

Whenever any program is executed in JS, GEC is formed and it is pushed into the call stack.



In line-2, .getElementById() calls the DOM APIs in browser (which is like lines of HTML codes) and .addEventListeners registers a callback on button (clicking) event inside web APIs and the event click is attached to it. This addEventListener() stays in the Web APIs as long as the it is not explicitly removed in the program or the browser is not closed.

Registering a callback – Whenever addEventListener() is encountered in a program, it registers a callback() function in the environment of Web APIs and attaches an event to it.

After line 3, there's nothing to execute, so GEC is popped out from the call stack.

When the user clicks on the button, cb() goes into the callback queue and waits for its turn to get executed.

Role of the event loop is to continuously monitor the callstack and the callback queue. If the callstack is empty and the event loop sees a function present in the callback queue waiting to be executed, it transfers it to the callstack , then the cb() is quickly executed. After it to put in the console, it is popped out of the callstack.

Why is a callback queue needed when the event loop can directly pick up the callback () function from the web APIs?

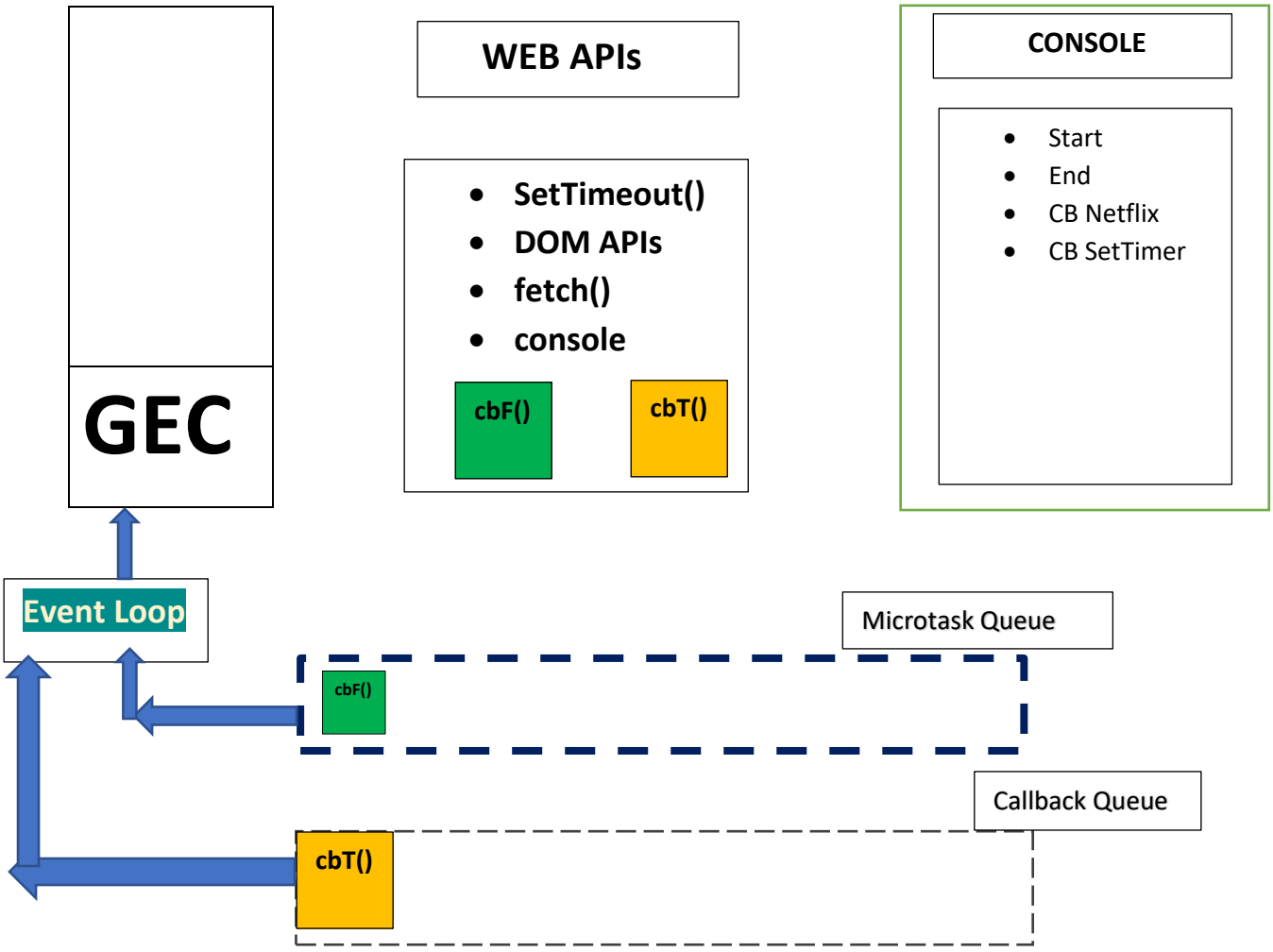
Note: These operations are valid for asynchronous callback functions.

Suppose a user clicks on the button, multiple times. In that case, the cb() is pushed into the callback queue multiple times and queued up to be executed. The event loop continuously checks the status of call stack, if the callstack is empty, it pushes the cb() to callstack to be executed.

How fetch() works?

1. Console.log("Start");
2. setTimeout(function cbT() {
   console.log("Cb SetTimeout");},500);
3. fetch(https://api.netflix.com).then(function cbF() {console.log("CB Netflix");})
4. console.log("End");

   Fetch() requests for an API call. Fetch() function returns a promise. So, we have to pass a callback function which will be executed once this promise is resolved.



When the control reaches line-2, it calls the timer in web API and puts cbT() in web APIs. It will then be pushed to the callback queue where it stays for next 5s. The control then moves to the next line where it encounters a fetch() function which sends a promise to the server for Eg. Netflix, until a request is received from the Netflix(server), it is then pushed to the Microtask Queue by the event loop. Microtask queue has a higher priority than callback queue.

Event loop keeps checking the status of callstack, when the call stack becomes empty, it pushes these callback functions to callstack and then they are executed.

All the cb-functions which go through promises as well as mutation observer, all of them occupy the microtask Queue. The event loop gives chance to the cbfunctions from callback queue to get shifted to the call stack only when all cbfunctions in Microtask queue have been executed.

Callback Queue => Task Queue

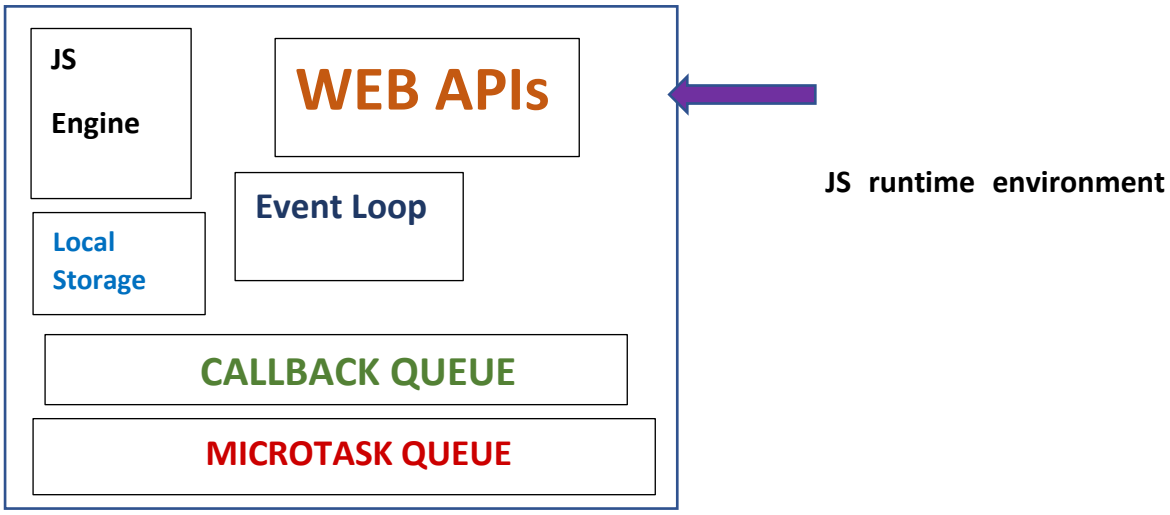
STARVATION(of tasks in the callback queue)-

It is the condition in which a cbfunction in microtask queue produces another microtask during its execution as a result of which the cbfunctions in callback queue(being lower in the priority than microtask queue) might not get a chance to get executed.

Episode 16– JS Engine exposed

JS runtime environment consists of JS Engine, Web APIs, Event Loop, Callback Queue, microtask queue. JS Engine is the heart of JS runtime environment. Browser can only execute the JS code because it has JS runtime environment. **Node.js** has JS runtime environment.

Some APIs such as setTimeout(),console.log are both present in Node.js and browser in the JS runtime environment. Even though these look alike but they may have a different implementation.



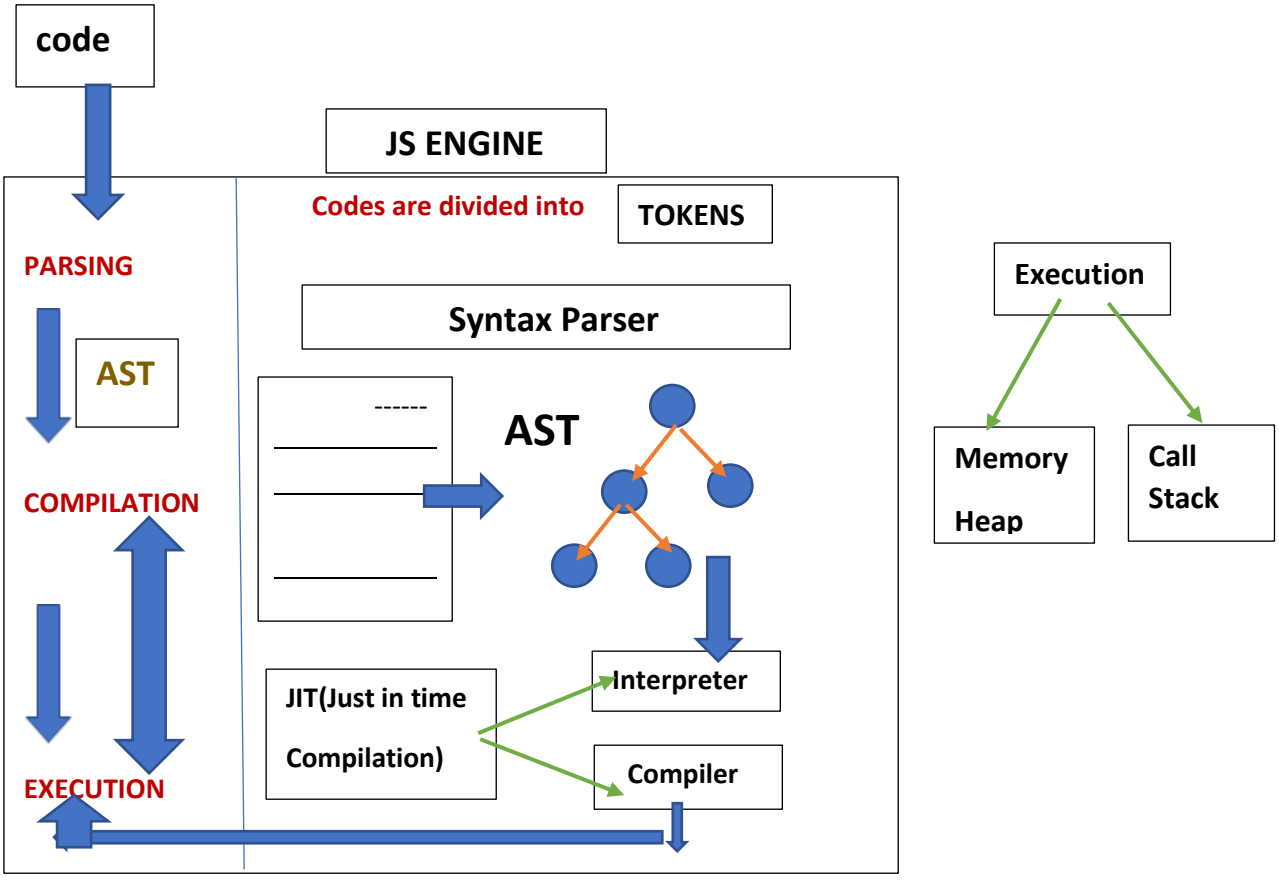
Different browsers have different JS Engine. For example Microsoft Edge: Chakra, Mozzila Firefox: Spidermonkey , **Google Chrome:** V8 according to the ECMA script.

The first JS Engine was spidermonkey which was breated by creator of JS.

JS Engine

JS Engine is not a machine!

JS Engine Architecture



Syntax parser generates AST(Abstract Syntax Tree) which is produced after parsing level.

Reference for AST: astexplorer.net

Interpreter	**Compiler**
• Reads line by line and executes • faster	• Reads entire code and create an optimized form of it even before execution. • More efficient

JS can behave as an interpreted or a compiled language depending upon the JS Engine. It was written initially to be used as an interpreter as it is mostly used in web browsers but later it is used as both. So, it is called a **JIT(Just-in-time)** language.

After the parsing, the compilation and execution go hand-in-hand. During compilation phase, AST is interpreted and during this time it is also converted to the optimized code by the compiler. So, the role of the compiler to optimize the code as soon as possible during the run time.

Memory heap is a space where all variables and functions are assigned memory. It is in constantly sync with the call stack and the garbage collector.

Garbage Collector frees up the memory heap whenever required. It uses **Mark & Sweep** algorithm to sweep off unnecessary stuffs from the code.

Mark-and-sweep algorithm

This algorithm reduces the definition of "an object is no longer needed" to "an object is unreachable".

This algorithm assumes the knowledge of a set of objects called ***roots***. In JavaScript, the root is the global object. Periodically, the garbage collector will start from these roots, find all objects that are referenced from these roots, then all objects referenced from these, etc. Starting from the roots, the garbage collector will thus find all *reachable* objects and collect all non-reachable objects.

This algorithm is an improvement over the previous one since an object having zero references is effectively unreachable. The opposite does not hold true as we have seen with circular references.

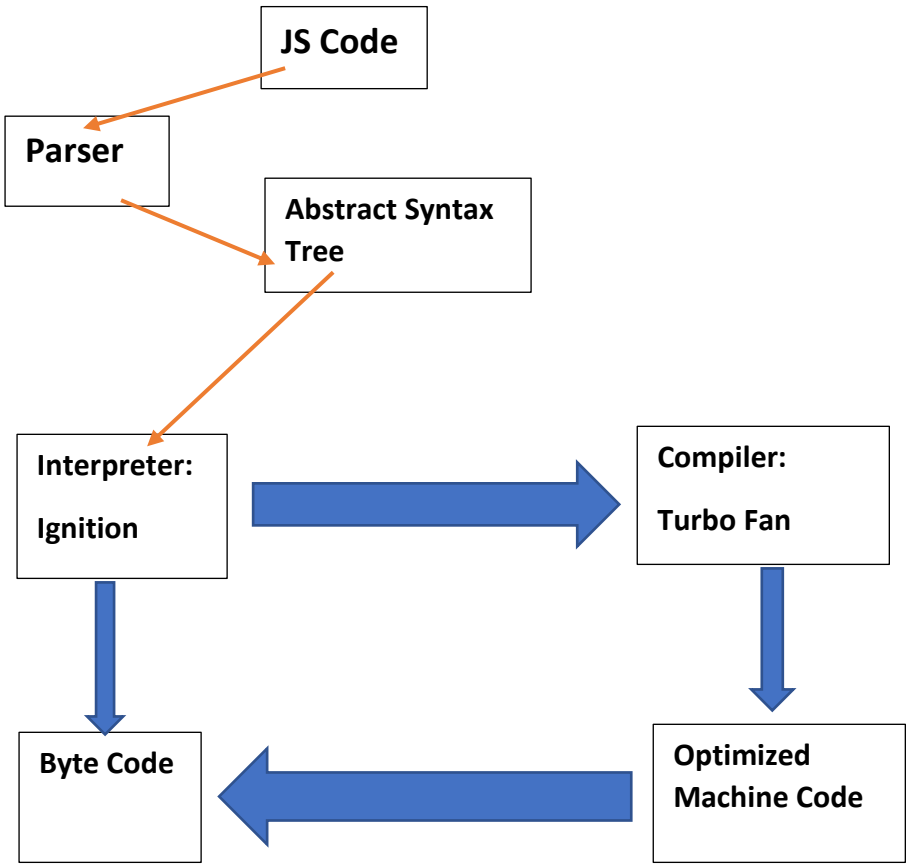
As of 2012, all modern browsers ship a mark-and-sweep garbage-collector. All improvements made in the field of JavaScript garbage collection (generational/incremental/concurrent/parallel garbage collection) over the last few years are implementation improvements of this algorithm, but not improvements over the garbage

collection algorithm itself nor its reduction of the definition of when "an object is no longer needed".

Compiler uses different ways to optimise the code such as inlining, copy elision and inline caching.

Google's V8 is the fastest of all JS Engines and its interpreter is ***Ignition*** and the optimizing compiler is known as ***Turbo Fan.*** Garbage collector of V8 is called ***Orinoco.***

V8 JS Engine:



Episode 17– Trust Issues with setTimeout()

setTimeout() function does not always print the result exactly after the expiration of the timer . There may be a case where there are a million lines of code and the GEC is still running. Even though the callback has already been passed into the callback queue but it will not be pushed to the call stack by the Event Loop as the call stack is not empty because GEC is still running. So, in this case, it may take more time than that of the timer to print. This is also called as the ***concurrency model*** of JS.

That's why it is advisable to keep the call stack empty or not to block the main thread.

Episode 18– Higher-order functions ft. Functional Programming

Higher-order function: A function that takes another function as an argument or returns a function from it is called a higher order function.

Eg.

```
1. function x() {
2. console.log("Namaste");
3. }
4. function y(x) {
5. x();
6. }
```

y → higher-order function, x→ callback function

Advantages of functional programming;

1) Reusability
2) Modularity
3) Passing a function into another function as an argument

_____END_____