papergrid
Date: / /

JavaScript

* Everything in Javascript happens inside an "Execution context"

Execution context

Memory (or variable environment)	code (or Thread of execution)
key : value	
a : 10	
fn : {...}	

→ Thread of execution is a executing code line by line at time.

* Javascript is a synchronous single threaded language.

* code : [square]

```
var n = 2;
function square(num)
{
  var ans = num*num;
  return ans;
}
var square2 = square(2);
var square3 = square(4);
```

Invoking the fn

Execution context

Memory	code
n : undefined → 2	n : 2
square : fn { copy square fun stored in memory }	square2
square2 : undefined → 4	memory: num : undef → 2, ans : undef → 4
square3 : undefined → 16	square3
	memory: num : undef → 4, ans : undef → 16; code: 4*4 = 16
	GEC (Global Execution Context)

* executing the code, it will create new brand execution.

* when new brand execution will create, it will allocate memory for parameters & local variable.

* The once execution is completed, execution context will delete.

[Note:- Everything completed, GEC will be delete.]

JavaScript

* Everything in Javascript happens inside an "Execution context"

Execution context

Memory	code
1st variable environment	2nd Thread of Execution.
key : value	
a : 10	
fn : { ..... }	

→ Thread of execution is a executing code line by line at time.

* Javascript is a synchronous single threaded language.

* code: [square]

```
var n = 2;
function square(num)
{
var ans = num * num;
return ans;
}
var square2 = square(2);
var square3 = square(4);
```

involking the fn

Execution context

Memory	code
1st Asc n: undefined → 2	n: 2
2nd n: 2	square 2
square: fn { copy square fun. placed in memory }	memory: num: undef → 2, ans: undef → 4
square2: undefined → 2nd square2: 4	square 3
square3: undefined → 2nd square3: 16	memory: num: undef → 4, ans: undef → 16 (4*4 = 16)
	GEC (Global Execution Context)

* executing the code, it will create new brand execution.

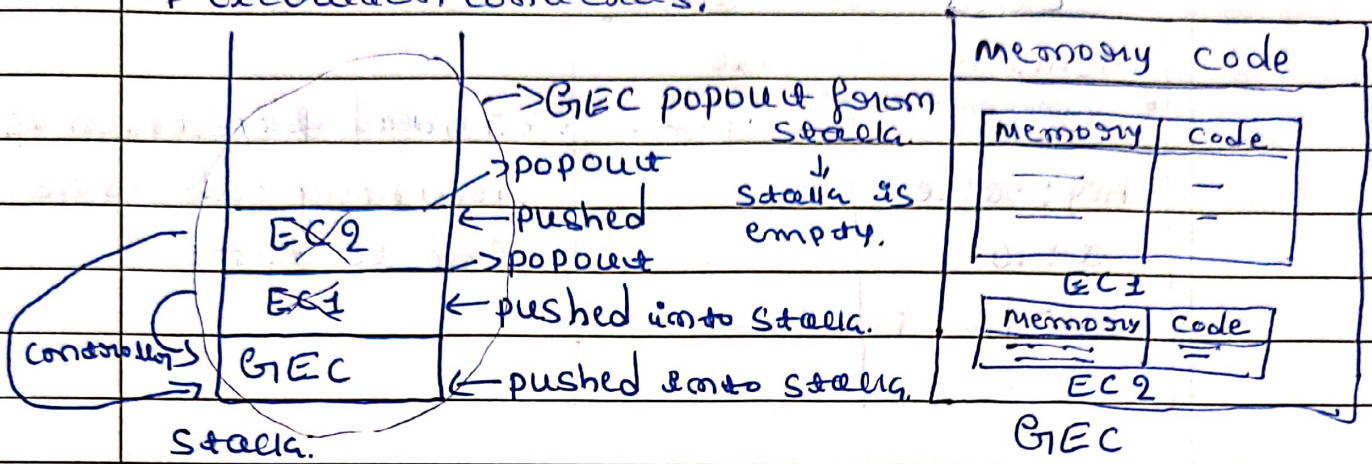
* when new brand execution will start, it will allocate memory for parameter & local variable.

* The once execution is completed, execution context will delete.

[Note:- Everything completed, GEC will be delete.]

=> **Call Stack:**

Call Stack maintains the order of execution of execution contexts.



* When function is invoked or called, then EC will be pushed into GEC stack.
* Once execution is completed, EC will popout from stack. And controll goes to GEC.

=> **Variables & functions:**

Note:-

Even starting the code, Memory will be allocating each & everytime.

* **Hoisting:**

We can access, Even before initilizing without any Error, Anywhere in the file

* For variable -> It's giving undefined.
* For function -> It's for itself.

Ex:- getName()

```
console.log(x); // undefined
x=10;
fun getName(){ function itself
{ console.log(x); }
```

⇒ **How function's works in JS:**

```
var x = 1;
a();
b();
console.log(x); ——→ Global variable.
function a()
{
  var x = 10;
  console.log(x);
}
function b()
{
  var x = 100;
}
```

GEC

memory	code.
x: ~~undef~~ 1	x = 1
a: fn {...}	

EC1

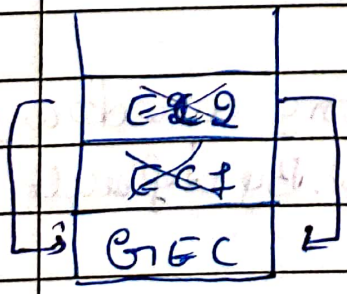
Memory	code
x: ~~undef~~ 10	x = 10
console.log(10);	

→ Local variable.

EC2.

memory	code
x: ~~undef~~ 100.	

→ Local variable.



controller moves GEC.

stack.

=> Shortest JS program

* Even if we will given empty file, then also JS engine will create GEC.

Window:

This is a big JS object given by JS engine. And we can access all variables & functions in entire file.

This:-

This keyword also provided by JS engine.

Note:

Chrome JS engine is V8.

Global space:

Anything is not inside a function called Global space.

Ex: var x = 10;] → Global space.
function b()
{
var x = 20; → Local space.
}

console.log(x) // 10;
console.log(window.x) // 10;
console.log(this.x) // 10;

If you not providing any window & this keyword infront of variable. By default it will take window object.

=> undefined vs not defined in JS:-

undefined:-

It's a placeholder, where can put some undefined placeholder. Before executing a single line of code.

But it's allocated memory & Temp placeholder

Ex:- consale.log(a); // undefined

var a=10;

consale.log(a); // 10

Note:-

* Javascript is a loosely type language. If we can assign value to variable.

Ex:- var x=123;

x="lucky";

console.log(x) // lucky.

=> Scope chain, Scope & Lexical Environment:-

Scope:-

Scope means, where we can access, a specific variable in a function.

```
function a()
{
  c();
function c()
{
  console.log(b); // 20;
}
}
var b=20;
a();
```

Lexical Environment:

Lexical Environment is a local memory, and along with LE of his parent.

Note:

First, it will check the local memory, If it's not there, then it will check the it's parent lexical environment.

Ex:
```
function a()
{
  var b = 10;
  c();
  function c()
  {
    console.log(b);
  }
}
a();
```

→ local memory & lexical scope of a().

=> Let & const in JS [Temporal Dead Zone]:

* Let & const declarations are Hoisted. But In temporal dead zone for time being.

Temporal dead zone:

Temporal dead zone is a time, since when let variable hoisted, and till initialize some value. That time b/w TDZ.

Reference Error:

RE is a, whenever you tries to access a variable inside a TDZ. It gives RError.

Ex: console.log(b) → RE
let b = 10;

Note:

The const & let variables are not storing global memory space.

If you tries to access from using the window.variable or this.variable. It will give undefined result.
name name

Ex: console.log(b); → RE. bcz this is hoisted in TDZ.
let b = 10;

console.log(a); → undefined.
var a = 200;

this.a // 200
window.a // 200
this.b or window.b // undefined.

=> Let & const Difference.

Let	const.
i) Let also hoisted but it's TDZ for time being.	i) const also a hoisted but it's TDZ for time being
ii) Using let, we can declare & initialize letter. let a; let a = 20;	ii) Using const, we have to declare & initialize at same time. const a = 100;
iii) It storing in diff memory space script:	iii) It's storing in a diff memory space. Script,

=> Error Types:

1) Syntax Error.

i) let a = 100;
let a = 2000;

ii) const a;

2) Reference Error:

If you tries to access before initialing. Bcz it's TDZ.

i) console.log(a)
let a = 1000;

ii) console.log(x)

3) Type Errors:

const a = 1000;
a = 2000;

=> Block Scope & Shadowing in JS :-

Block :- * If you want to combine multiple statement together, we need block. ~~scope~~.

Ex: If (true)
{ // Compound statement
var a = 10;
console.log(a);
}

Block Scope :-

what are the variables & functions access inside a block.

Ex: {
var a = 10; → This is Global scope.
let b = 20; } → These are available in
const c = 30; Block Scope
}

= { var a = 10;
let b = 20;
const c = 30;
console.log(a) // 10
" " (b) // 20
" " (c) // 30
}
console.log(a) // 10
" " (b) // Error, b is not defined in
" " (c) current scope.

⇓ let & const are block scope, so we can access only in current block. But not outside block.

=> Shadowing in JS:-

```
var a = 100;
{
  var a = 10;
  let b = 20;
  const c = 30;
  console.log(a); // 10
  "   "   (b); // 20
  "   "   (c); // 30
}
console.log(a); // 10;
```

Follows Lexical Scope.

```
let a = 100;
{
  let b = 10;
  let a = 20;
  "
  console.log(b) // 10
  "   "   (a) // 20
}
console.log(a) // 100
```

var changed to let

* Shadowing in function:-

```
const c = 100;
function ab()
{
  const c = 1000;
  console.log(c); // 1000
}ab();
console.log(c); // 100
```

{ let & const are block scope & pointing to diff memory location.

* Illegal Shadowing:-

```
let a = 10;
{
  var a = 20;
}
```

```
var a = 10;
{
  let a = 100;
}
```

OR we can shadow like below.

```
let a = 10;
{
  let a = 100;
}
```

```
var aa = 200;
{
  var aa = 2000;
}
```

Closures in JS :-

Closure is a function that combines bundles together with lexical scope (environment)

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

Ex 2:-
```
function x()
{
  var a = 7;
  function y()
  {
    console.log(a);
  }
  return y;
}
var z = x();
console.log(z); // function y()
console.log(z()); // 7
```

Closure uses in JS:

i) Module design pattern
ii) Currying.
iii) Functions like once.
iv) memoize
v) Maintaining state in async world.
vi) setTimeouts.
vii) Iterators.
And many more.

Ex: print after 1 sec values 1,2,3,4,5.

```
for (var i=1; i<=5; i++)
{
  setTimeout(function()
  {
    console.log(i);
  }, i*1000);
}
console.log("Namaste JS");
```

use let

O/P
Namaste JS
6, 6, 6, 6, 6.

∴ when we are using the var, it always refere to the same memory. And above example, it will until executing the i value.

∴ But when we are going to use let variable, it will create a always new copy of that variable. Thats the reason, It will print 1, 2, 3, 4, 5.

1. What is closures?

i) closure is a function, along with his lexical scope bundle together.

ii) In Javascript, each & every functions has access to it outer lexical environment.

iii) Even this function executed from other place, still its remember of lexical of his parents.

Ex:-

```
function outer()
{ var a=10;
  function inner()
  {
    console.log(a); // 10
  }
  return inner;
}
outer()();
```

Then, If you want access inner values, Still it remember. what was the value of a.

```
var close = outer();
close();
```

Note:-

i) closure are using for data hiding.

ii) First level lexical environment as calling a closure function.

=> Functions:
Functions are a heart of the javascript.

i) Function Statement:
```
function a() -> a();
{
  console.log("a called");
}
```
Def:- we can call before declaring the fn. Still, we can access the function.

ii) Function Expression:
```
var b = function() -> b();
{
  console.log("b called");
}
```
Def:- This will works as variable, & assigned value to b as undefined. So, getting the error.

iii) Function Declaration:
This is also called as Function statement. Both are same [Fn statement or Declaration].

iv) Anonymous Function:-
It's a look like Function statement but we don't have name.
```
function()
{
}
```
} Gives a syntax error.

Use:-
Anonymous functions are used when the functions are used as value.
It's means we can assign that value to variable.

v) Named Function Expression:
It's like anonymous function with named expression.

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

b(); → we can access the function.
xyz(); → we will get an error. Like xyz is not defined.
∴ Now xyz is a not Outer scope fn. Its local.
→ xyz() inside the function, we can get the value of the function.

vi) Diff b/w Parameters & Arguments?

```
function b(param1, param2)
{
  console.log("b called");
}
b(10, 20);
```

param1, param2 → parameters.
10, 20 → Arguments of b function.

Arguments:
The values are passed inside a function, are known as arguments.
invoking

parameters:
which will get those values are called parameters.

viii) First class functions:

The ability of functions to be used as values and can be passed as arguments to another function, can be return from the function's. This ability known as First class functions.

Ex: var b = function xyz(Param1)
{
return function xyz()
{
}
}
console.log(b());

Note:

First class citizens also called as First class Functions.

viii) Arrow Functions:

```
const sum = () =>
{
let a=10, b=20;
let sum;
sum = a+b;
}
```

Callback Functions in JS :-

1) What is callback function in Javascript?

* we can take a function and passed it as a argument in another function.
* It could provide a async execution.

Ex:
```
function x(y)
{
}
x(function y()
{
});
```
→ callback function.

Ex:
```
setTimeout(function()
{
console.log("timeout");
},5000);
function x(y)
{console.log("x");
y();
x(function y()
{console.log("y");
});
```

O/P
x
y
timeout → after 5 m second.

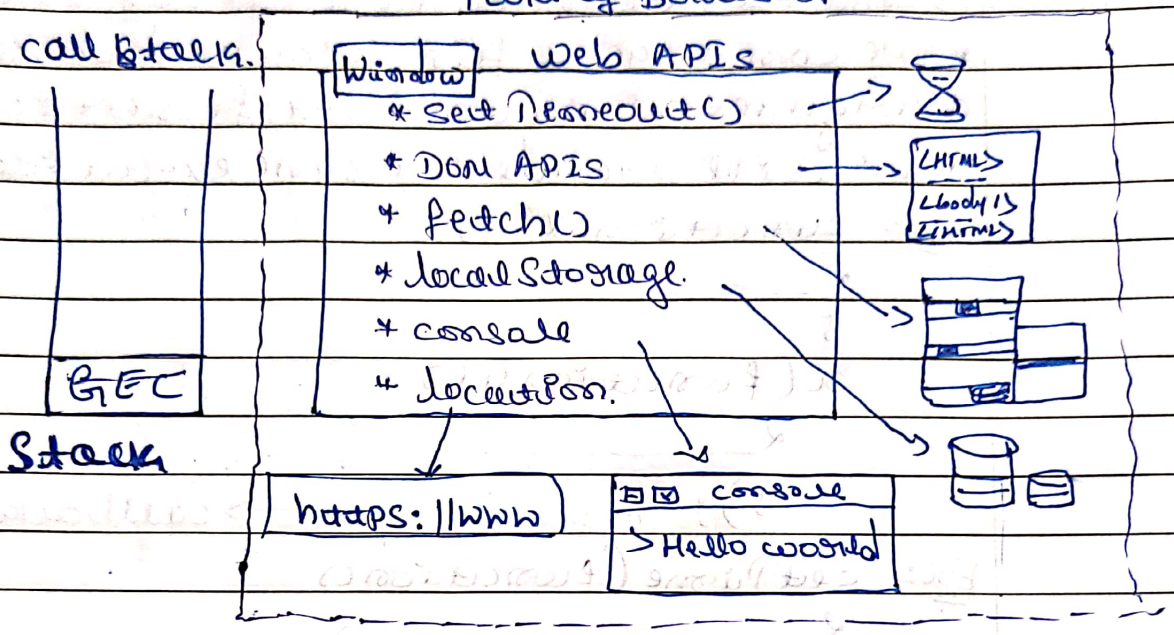
Note:

Javascript has only one callstack & one main thread.

After created event listener, once executed all events. So we need to remove Event listener for Garbage collection.

=> Event Loop & Asynchronous Javascript:

Part of Browser



* If you want access any web APIs inside your code, through global object window, we can access.

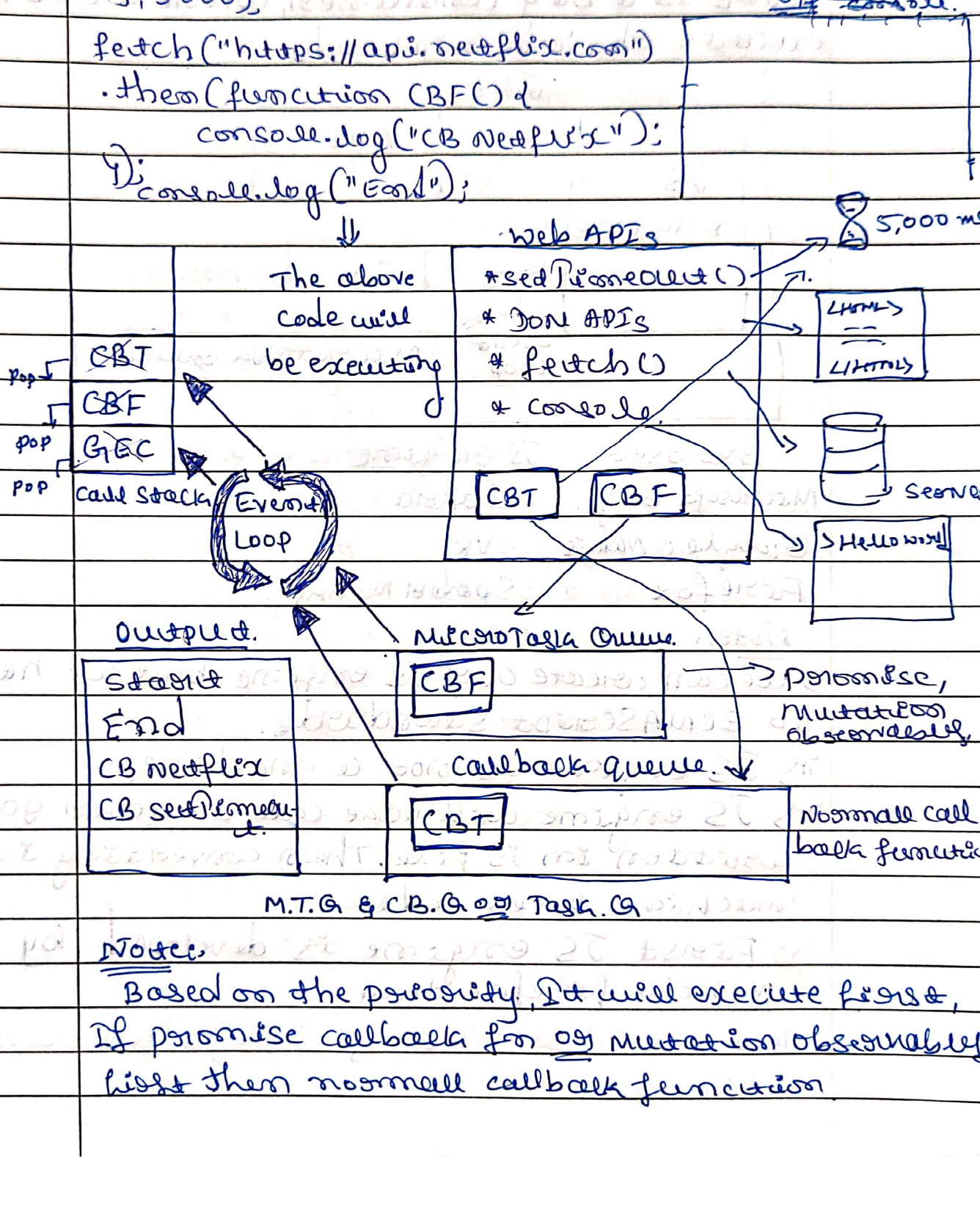
Ex: window.setTimeout().
window.console()
" .location().

* If your not mentioning window keyword, By default, it will take window. we can directly use like this.

setTimeout();
console.log(); , document().
location(), fetch(),

Ex: Event Loop example.

1st line of execution ⇒ console.log("start");
setTimeout(function CBT() {
console.log("CB setTimeout");
}, 5000);
fetch("https://api.netflix.com")
.then(function CBF() {
console.log("CB Netflix");
});
console.log("End");

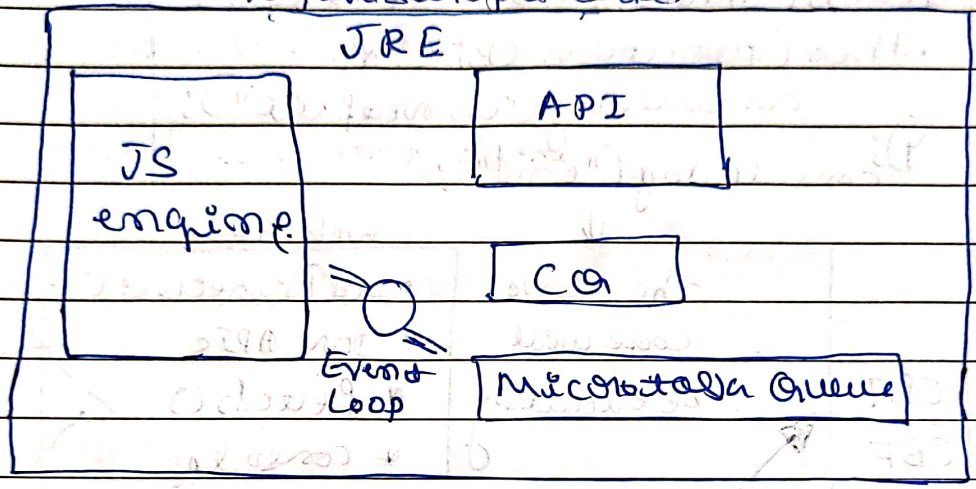


Note:
Based on the priority. It will execute first, If promise callback for or Mutation Observable first then normal callback function.

=> Javascript Engine:-

* If you want to run any line of code Javascript needs a Javascript engine.

JRE [Javascript Runtime Environment]:-
JRE is a big container, where we can execute the javascript code.



Browser	JS. Engine
Microsoft Edge.	Chakra
Google & Node.js	V8
Firefox	Spider Monkey.

Note:-

i) We can create our JS engine but we have to ECMAScript standards.

ii) JS engine is not a machine.

iii) JS engine will take code, whatever you written in JS file, then converting into machine level code.

iv) First JS engine is developed by Mozilla firefox team.

= Browser has JRE & node.js has own JRE.

JS Engine :-

Code

⇓

Parsing

⇓ → AST

Compilation — JIT compilation [Just In Time].

⇓ — Interpreter & Compiler

Execution → Memory Heap / call stack.

* i) Take the code as a input. Then code will be broken into tokens

let a = 7.

1 2 3 4

* ii) Take the ~~code~~ token. Then converts the AST [Abstract Syntax Tree].
* iii) Interpreter: [Fast]

It is execute's the code line by line. Even we don't know, what will be happening next.
* iv) Compiler: [More efficiency].

It will compile the entire code first then it will form the optimization of that code, then it will execute.

Note: JS having the both interpreter & compiler.

V8 Architecture:

Javascript source code.

↓

Parser

↓

AST [Abstract Syntax Tree]

↓

Interpreter → Compiler Turbo Fan

Ignition.

↓ ↓

Byte code. → optimized machine code.

=> Higher Order Functions :-

DRY Principle

Don't Repeat yourself.

* Higher Order Function is a function, which takes another function as a arguments, & which returns.

Ex: function x()
{
  console.log("function called");
}
function y(x)
{
  x();
}
y(x); // Function called.

Note:
map, filter & reduce are HOF.

⇒ **map:-**

* If you want to transfer an array, we need to use map.

i) i/p :- const arr = [2,4,6,8];

o/p :- double = [4,8,12,16];

Ex: function double(x)
{
return x*2;
}

const res = arr.map(double);
console.log(res); // [4,8,12,16]

OR

ii) i/p :- arr [5,1,3,2,6];

o/p :- arr ["101","1","11","10","110"]

~~function binary~~

const res = arr.map(function binary(x)
{
return x.toString(2);
}
console.log(res); // ["101","1","11","10","110"]

OR

iii) i/p :- arr [5,1,3,2,6];

o/p :- arr [15,3,9,6,18];

const res = arr.map((x) => x*3);
console.log(res); // [15,3,9,6,18]

=> **Filter:-**

Filter is a HOF, which calls filter the array based on some logic.

Ex: 1

```
const arr = [5, 1, 3, 2, 6];
Print ODD values only.
function oddValue(x)
{
  return x%2==1;
}
const res = arr.filter(oddvalue);
console.log(res);          //[5, 1, 3]
```

Ex: 2

```
i/p = [5, 1, 3, 2, 6];
o/p = [5, 6].
Print greater than 4 values only.

const res = arr.filter(function (x)
          {
            return x>4;
          });
console.log(res);          // [5, 6]
```

3: Even numbers:

```
const res = arr.filter((x)=>{
        return x%2==0;
  }
console.log(res);          // [2, 6]
```

=> **reduce:-**

reduce is a HOF. And which will return a single value.

Ex:-