

15 video

## ~~#~~ First Class Function ft. Anonymous functions

- Topics to cover :- function statement
- function Expression
- function Declaration
- <sup>Anonymous</sup> Function
- Named Function Expression
- Difference between Parameter & Argument
- First class function
- Arrow function

### (1) Function statement vs function Expression

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

O/P: a called                           O/P: b called

→ A way to create → A way to create  
function function

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

O/P: a called                           O/P: Type Error: b is  
not a function.

\* Differ in hoisting in memory creation phase  
of execution context exact copy of `a()` will  
be started as `()` is complete function  
while `b` will a variable and hence it will  
be undefined. If you try to run `a()` as a function  
throws an Error.

② Function Declaration is same as function statement.

function Declaration aka function statement

Anonymous Function

function without a name

function ( ) { }

← Invalid → Syntax Error:  
Syntax Requires function name.

(S, J) X

→ DO not have its own identity

→ It is invalid statement.

→ These are used ~~where we want to function~~

~~as value, to obtain return value~~

var b = function () { }

← function  
Expression

{ } ( ) { } { }

(S, J) X

④ Named Function Expression

var b = function x() { }

console.log('b called');

}

b(); → b called

x(); Reference Error: x is not defined.

because x() is located as local function only. If you try to access in out of scope it will give Reference Error.

④

## Argument vs parameter

inner var near function (param1, param2)

parameter

same is true

label identifier are called parameter

These are local value  
value passed are  
called argument

global variable ← local →  
function variable  
value.

x(1, 2)

⑤

First class function legg og spid  
passing function inside a function as  
an argument.

f() = d(x)

eg: var b = function (param) {  
console.log(param);  
}  
function xyz () {}  
b(xyz);

O/P: xyz()

f() = d(x)

"Hello" is printed

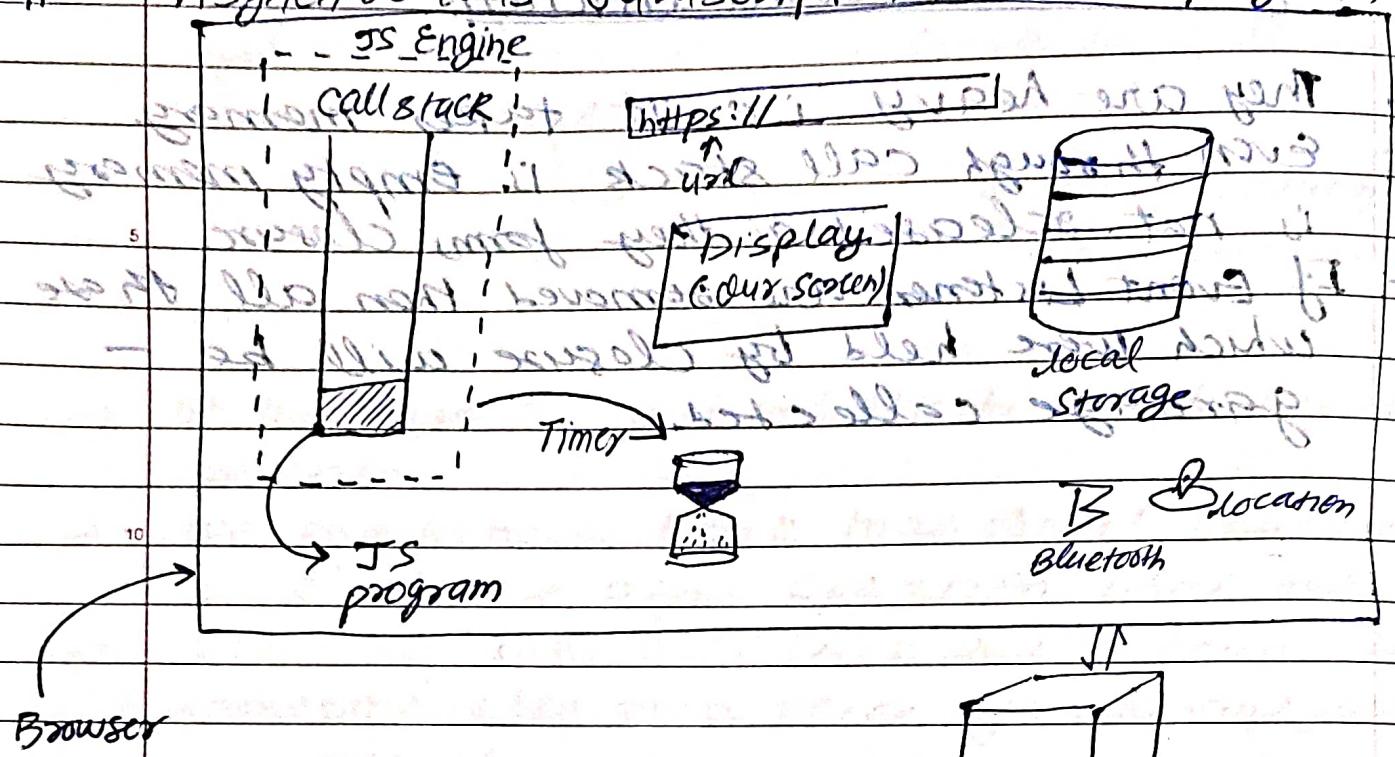
{}

as we can see hello d ← ; ()

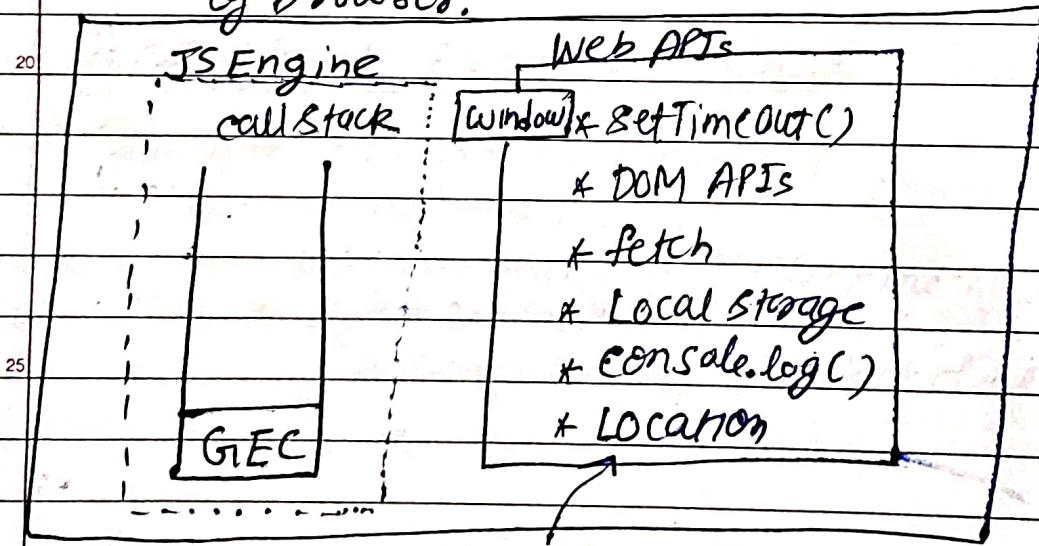
inner function x : local variable  
at first will run at inner level so hello is called if (x) is passed  
then similarly like this if (y) is passed to fun. in hello.

17 Video

## # Asynchronous Javascript & Event loop from scratch



- > The functionality related to access the website, timer, bluetooth, location etc... How can be accessed by JS Engine?
- we have web APIs to access all the element of browser.



And many more web APIs

**web APIs** are part of web browser  
it is not part of JS

- All web API's are for global variable.
- `window.console.log("Me.");` == `console.log("Me");`  
By default global.

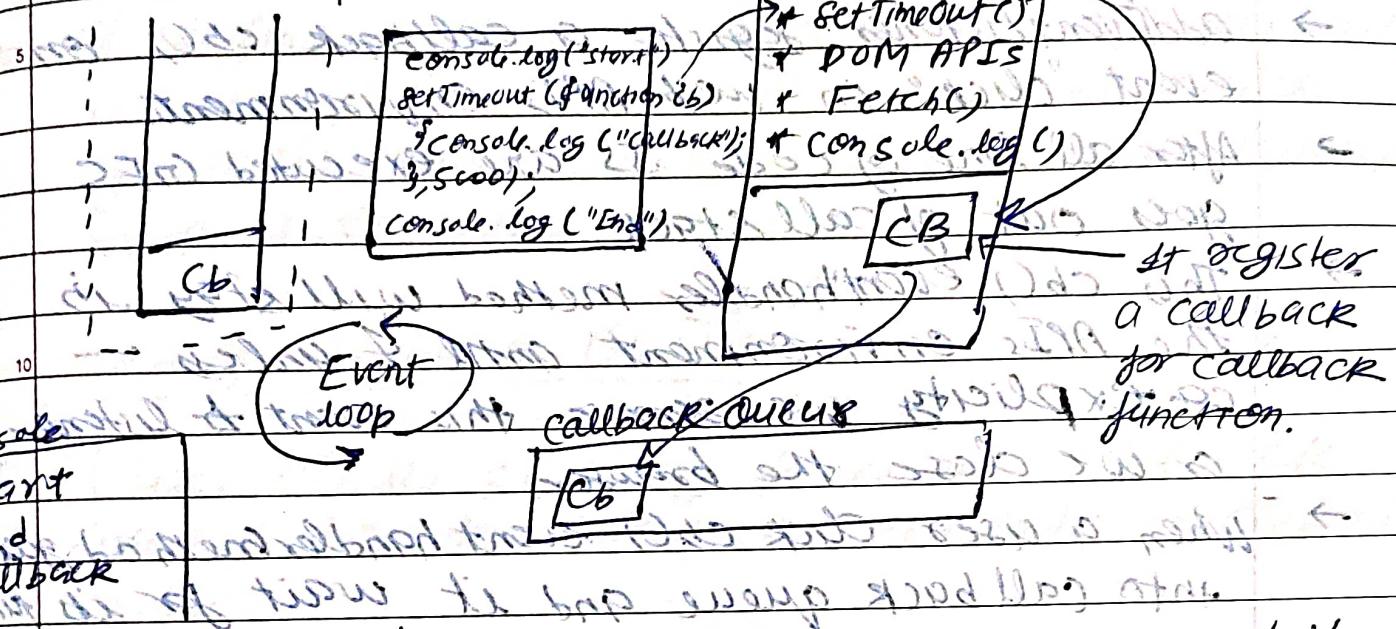
Browser wraps all its super power (like timer, screen access, local storage etc) into window object and makes it accessible to JS engine / Call stack through Web APIs

Camlin Page  
Date

Example 1

JS Engine  
Call Stack

Web API



consolidated dr. loop

Event

callback queue

st register  
a callback  
for callback  
function.

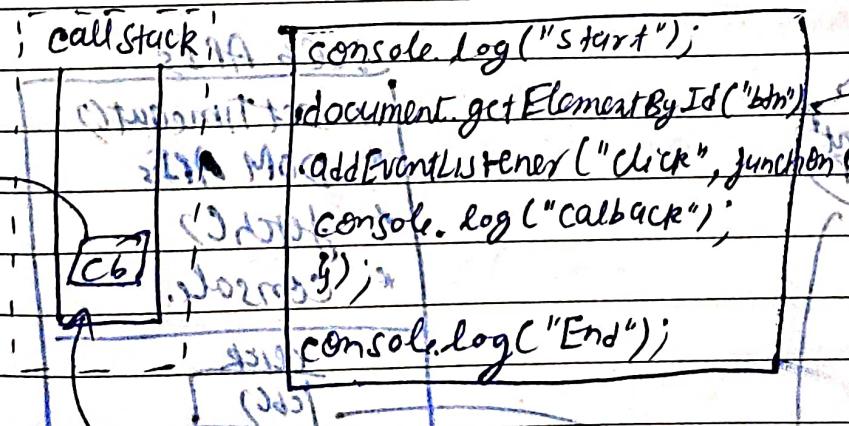
start

End

callback

- The JS code keeps on executing mean while timer keeps on running. When timer expire
- This cb() goes into callback queue
- Event loop keeps checking in the callback queue if there is something. When event loop find cb() in the queue it just pushes into call stack.

Example 2



pop out

Web APIs

\* SetTimeout  
\* DOM APIs  
\* fetch  
\* console

click  
CB()

callback queue

callback queue

console  
Start  
End  
callback

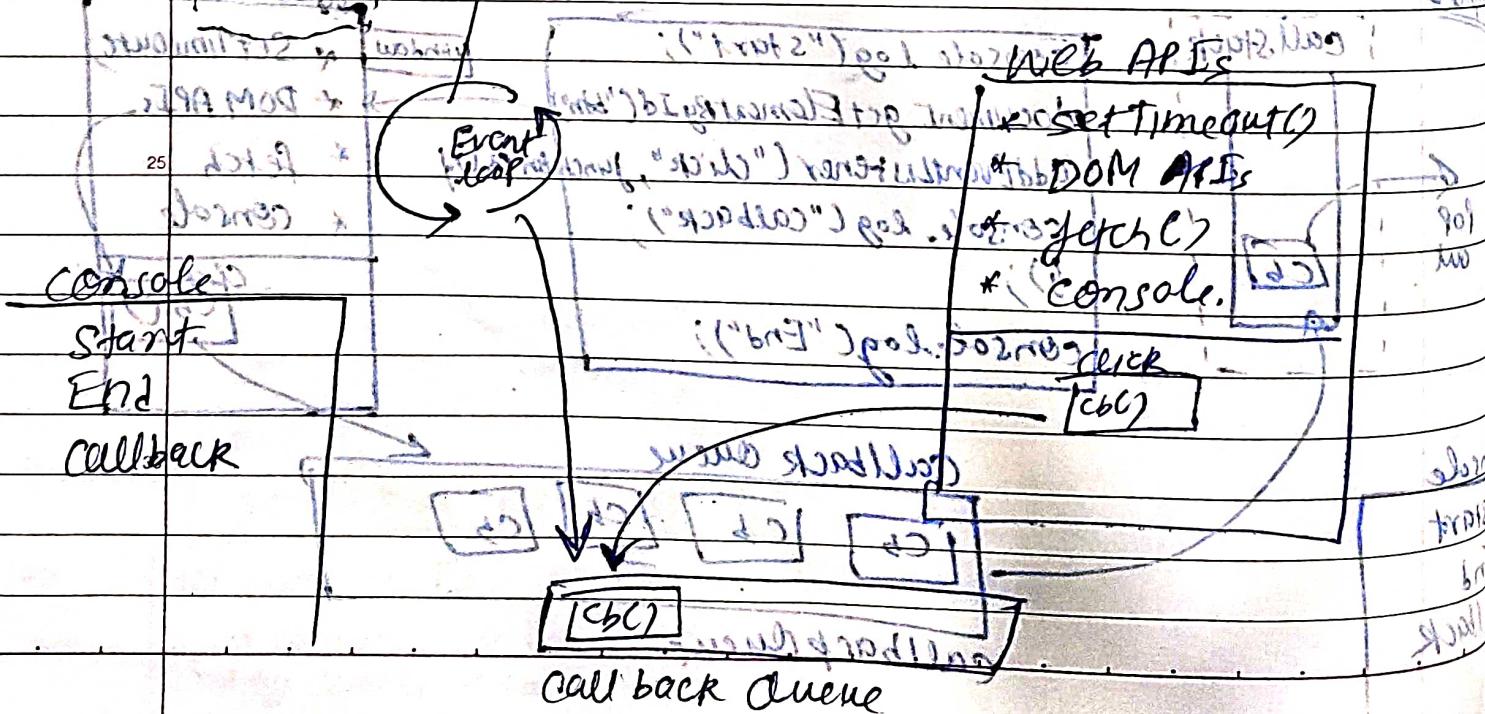
callback queue

DOM API

- DOM will act as html source file.
- When anyone access DOM API it basically access the html source file and tries to find the passed Id into DOM API.
- addEventListener registers a callback cb() on event "click" in web APIs environment.
- After all line of code JS are executed GFC goes out of call stack.
- This cb() event handler method will stay in the APIs environment until either explicitly we remove this event listener or we close the browser.
- When a user click cb() event handler method goes into call back queue and it wait for its turn.

```

    stack.push(cb);
    document.getElementById("bm").addEventListener("click", function cb() {
      console.log("Callback");
      console.log("End");
    });
  }
}
  
```



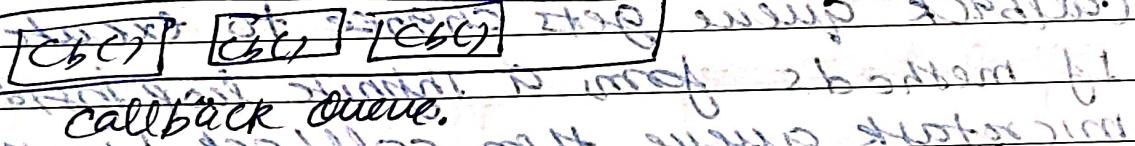
Event loop: Without setup since it's a lot.

- It has only one job
- continuously monitors the callstack & callback queue

If it finds a function in callback queue wait list it takes and pushes into callstack after callstack becomes empty & then callback method executed and if there is no

## Need of callback queue

If user clicks so many times then callback method will not execute in clicked order but will be pushed into callback queue.

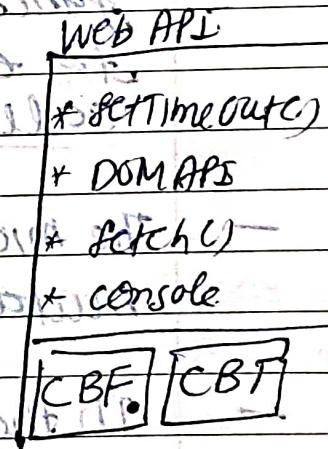


callback queue.

Example 3

```

    console.log("start");
    setTimeout(function() {
        console.log("CB setTimeout");
        fetch("https://api.netflix.com")
            .then(function() {
                console.log("CB Netflix");
            })
            .finally(function() {
                console.log("End");
            });
    });
  
```



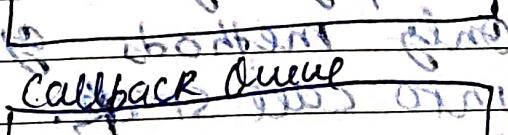
Start

End

CB Netflix

CB setTimeout

start if callback is microtask queue if it's not then it's promise mutation



Microtask Queue

Mutation

Callback Queue

Microtask Queue

Fetch APIs: It work quite differently. good. I know  
 fetch() method goes to make network call  
 and request API call.

- It return a promise.
- The callback function is executed once the promise is resolved.
- JS Engine does one task at a time but browser can do multiple task at a time.

### microtask Queue VS Callback Queue

should work with input from Q2 slides  
 basic It has higher priority than callback queue.

- All the method of this executed then only callback queue gets chance to execute.
- If methods form a infinite loop instead microtask queue then callback / method function will never get opportunity to execute. This stage is called callback queue starvation.

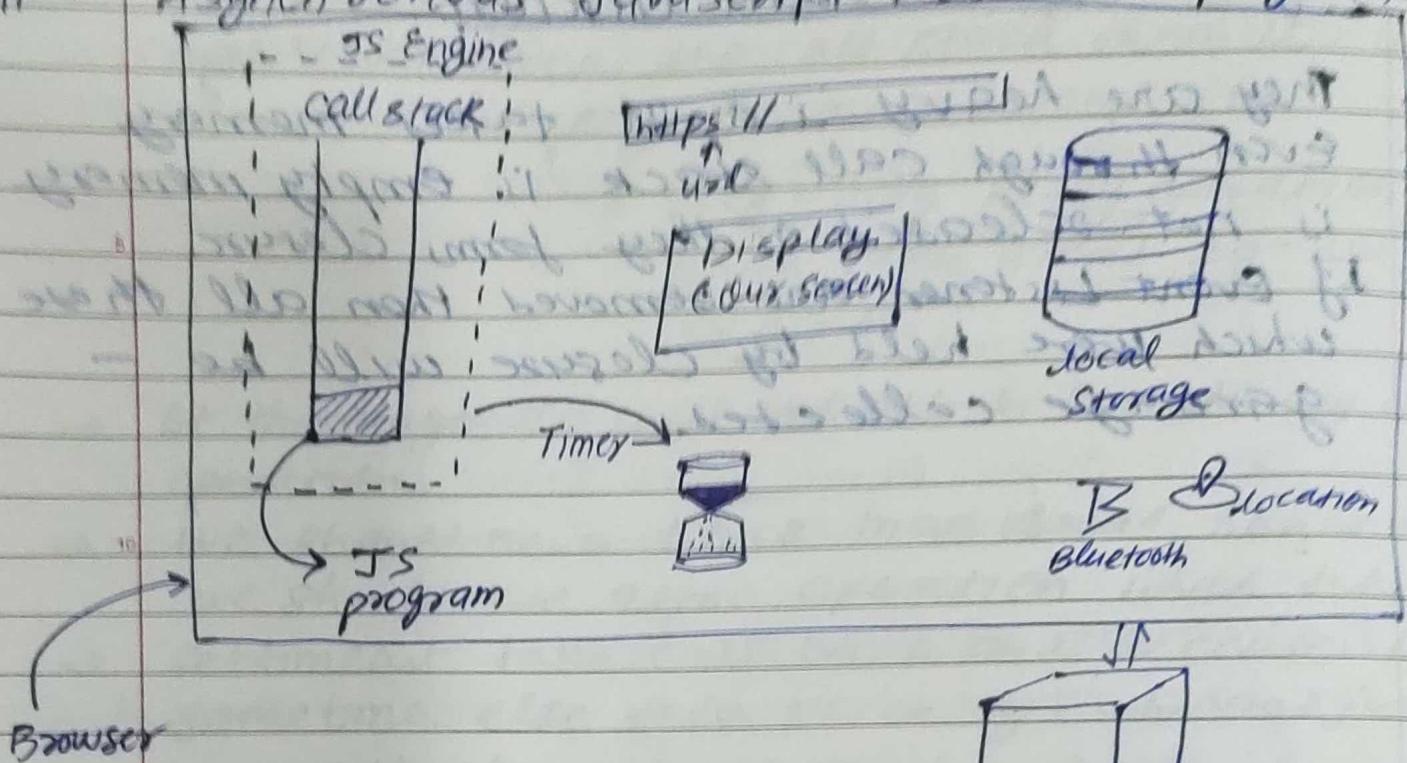
- The microtask queue contains only Promise fulfillment observes.
- In the callback queue setTimeout & DOM APIs are executed.

Event loop  
 Event loop keeps on checking if call stack is empty. If call stack is empty then only methods of microtask are pushed into call stack.

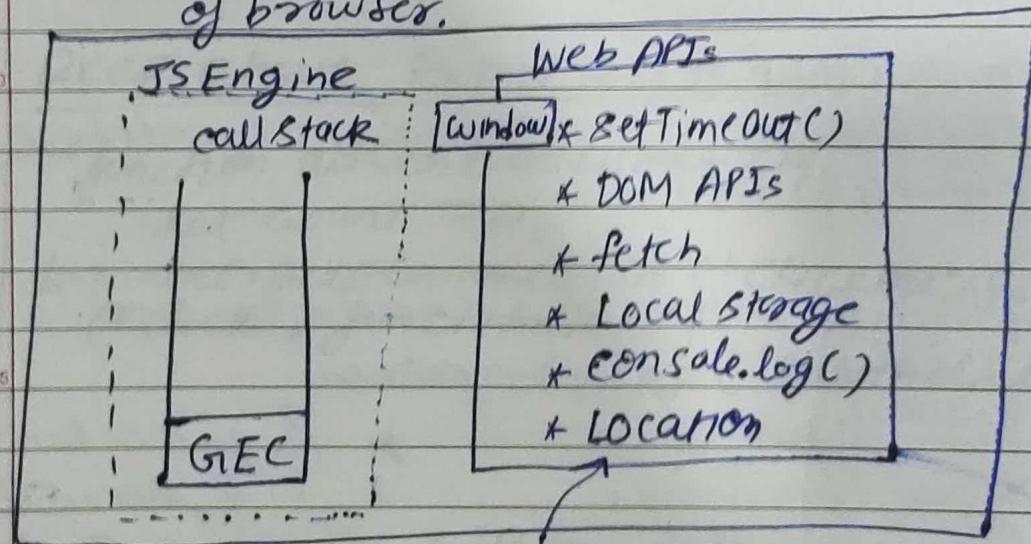
- When microtask queue become empty then only callback queue methods are pushed into call stack.

17 video

## # Asynchronous JavaScript & Event loop from Sam



- > The functionality related to access the website, timer, bluetooth, location etc... how can be accessed by JS Engine?
- we have web APIs to access all the elements of browser.



And many more web APIs

**web APIs**  
web APIs are part of web browser  
it is not part of JS

- All web API's are for global variable.  
→ `window.console.log("Me.");` == `console.log("Me.");`  
→ By default global.

Browser wraps all its super power (like timer, screen access, local storage etc) into window object and makes accessible to JS engine / Call stack through Web APIs

Example 1

JS Engine

Call Stack

Web API

\* setTimeout()  
\* DOM APIs  
\* Fetch()  
\* console.log()

```
console.log("start")
setTimeout(function cb() {
  console.log("callback");
  console.log("End");
}, 5000);
```

CB

It registers a callback for callback function.

Event loop

callback queue

CB

console  
start  
End  
callback

- The JS code keeps on executing mean while timer keeps on running. When timer expire
- This cb() goes into callback queue
- Event loop keeps checking in callback queue if there is something when event loop find cb() in the queue it just pushes into call stack.

Example 2

Call stack

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

```
document.getElementById("btn")
```

```
.addEventListener("click", function cb() {
```

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

```
});
```

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

pop out

CB

Web APIs

\* setTimeout()

\* DOM APIs

\* fetch

\* console

click

CB(C)

callback queue

CB CB CB CB

callback queue

console  
start  
End  
callback

DOM API

- DOM act as html source file.
- When anyone access the DOM API it basically access the html source file and tries to find the passed Id into DOM API.
- addEventListener registers a callback cb() on event "click" in web APIs environment.
- After all line of code JS are executed GC goes out of call stack.
- This cb() event handler method will stay in the APIs environment until & unless we explicitly remove this event listener or we close the browser.
- When a user click cb() event handler method goes into callback queue and it wait for its turn.

script: console.log("start")

stack

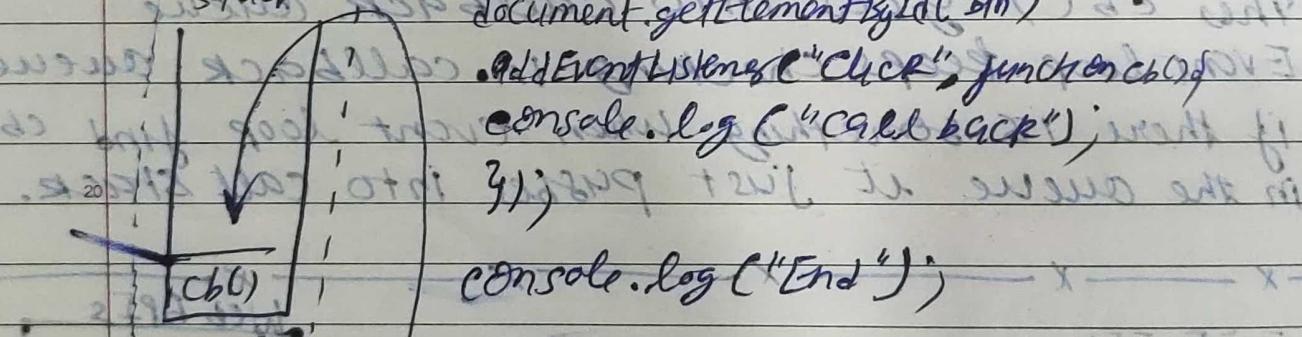
document.getElementById("bm")

.addEventListener("click", function cb()

console.log("callback")

3); // will run in inner site

console.log("End")



Console:

Start

End

callback

Event Loop

Web APIs

\* setTimeout()

\* DOM APIs

\* fetch()

\* console.

click

cb()

call back queue

Event loop

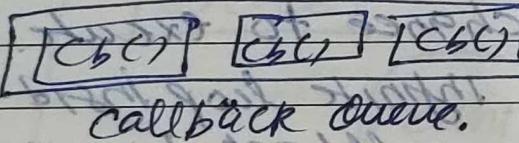
It has only one job

- continuously monitors the call stack & callback queue

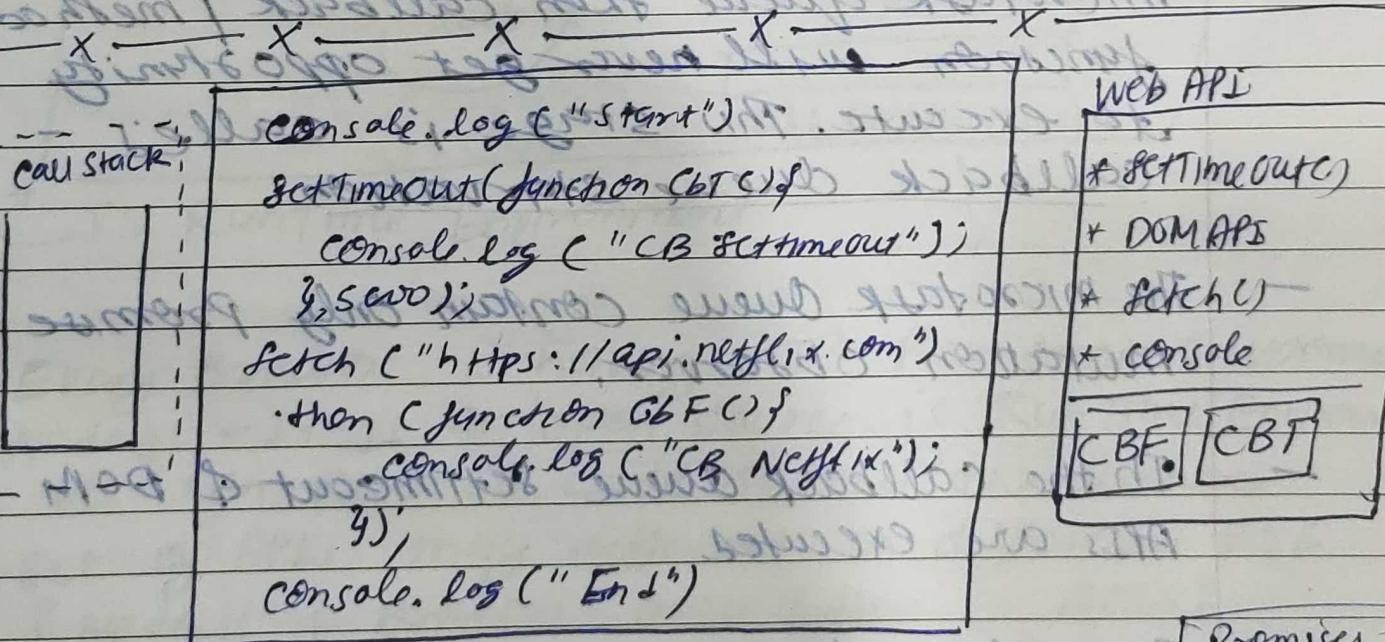
If it finds a function in callback queue wait list it takes and pushes into call stack automatically. Once call stack becomes empty then callback method executes.

Need of callback queue

If user clicks so many times then callback method has to be forced to execute in click order.



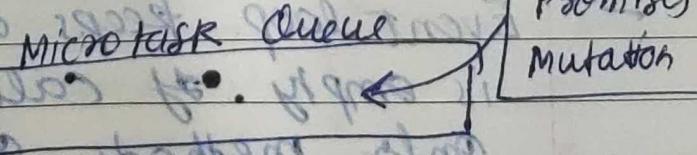
callback queue.



Start  
End

CB Netflix

CB setTimeout



Fetch APIs: It work quite differently. ~~work in browser~~  
fetch() method goes is used to make network call  
and request API call.

- It return a promise.
- The callback function is executed once the promise is resolved.
- JS Engine does a task at a time but browser can do multiple task at a time.

### microtask Queue vs callback Queue

It has higher priority than callback Queue.

- All the method of this executed then only callback queue gets chance to execute.
- If methods form a infinite loop in microtask queue then callback / method function will never get opportunity to execute. This stage is called callback queue starvation.

- The microtask queue contain only Promise & mutation observer.
- In the callback queue ~~settimeout & DOM~~ APIs are executed.

Event loop keeps on checking if call stack is empty. If call stack is empty then only methods of microtask are pushed into call stack.

- When microtask queue become empty then only callback Queue methods are pushes into call stack.