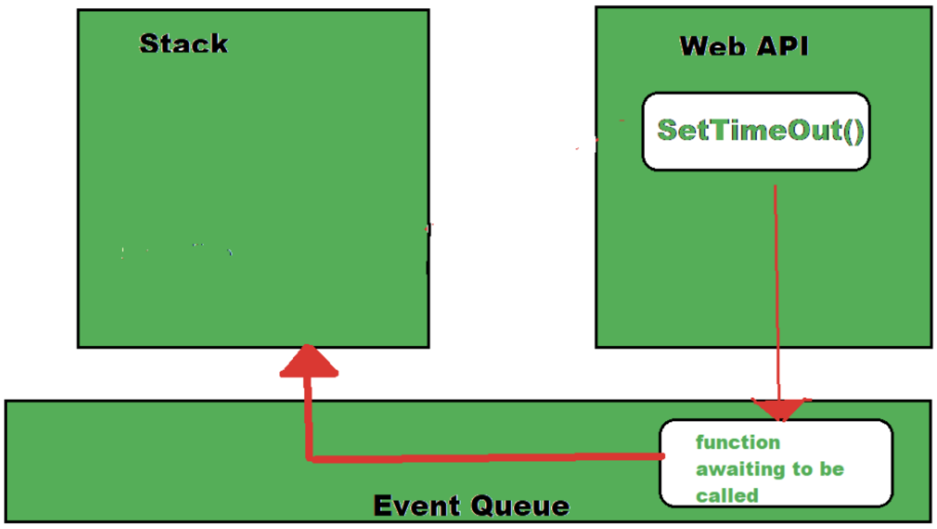
Javascript: synchronous single threaded language
js runs in one line at a time manner and there is no possibility of running code in parallel.

Function call stack: The function stack is a function which keeps track of all other functions executed in run time

Event loop: An event loop is something that pulls stuff out of the queue and places it onto the function execution stack whenever the function stack becomes empty.

Microtask queue : execute after event queue



Execution Context: : 1. memory: variable env(key-value)
2. Code : thread of execution

→ when a function is invoked then a brand new execution context is created inside code section of parent execution context

Before execution of js code memory is allocated to each and every function and variable
Hoisting : access function, variable before initialising

```
getName();
console.log(x);
var x=7;
function getName(){
console.log("namaste")}
```

Output
namaste
undefined

Lexical env : local memory along with lexical env of parent.
Lexical Environment : it's the internal js engine construct that holds **identifier-variable mapping**. (here **identifier** refers to the name of variables/functions, and **variable** is the reference to actual object [including function type object] or primitive value). A lexical environment also holds a reference to a **parent lexical environment**.
And every lexical environment tracks its parent lexical environment (that of parent execution context). As a result, every function has a chain of lexical environments attached to it.

```
function a(){
console.log(a)}

var a =10;
a();

//output 10
```

Map filter Reduce

Map: help in transformation
eg:

```
 arr=[1,2,3,4,5];
function double(x){
return x *2 ;
}
const output =arr.map(double)
// arr.map((x)=> x*2);
```

Filter : to filter values

```
arr=[1,2,3,4,5];

function isOdd(x){
return x%2!=0;

}
const output =arr.filter(isOdd);
```

//output [1,3,5];

Reduce: take all element of array and come out a single output value

```
 arr=[1,2,3,4,5];

const output = arr.reduce(function(acu, curr){

acu= acu+curr;
return arr;
}, 0);
```

// second argument is initial value of a acu
// first argument is function
// curr represent each element of array

// output : 15

Temporal Dead Zone: Time diff since when let variable was declared and initialised

→ variable cannot access in TDZ

eg:

```
console.log(a);
let a= 10;    //output reference error
```

→ In const with declaration we have to initialization

→ let and const are block scope while var is global scope
→ call stack reside JS engine.
→ JS engine reside browser

Web api — Console, set timeout, dom tree, local storage , location . these reside in browser which can be used in js code(call stack). these webapi is wrap up in global object Window

Call Back fn:A callback is a function passed as an argument to another function

eg:

```
setTimeout(function(){
console.log("timer)}, 5000);
```

```
function x(y){
console.log("x");
y();
}

x(function y(){
console.log("y");
});
```

```
output:  x
         y
         timer
```

eg :2

```
document.getElementById("btn").addEventListener("click", function xyz(){
console.log("button clicked")});
```

//here xyz is call back function. when button is clicked this fn which reside in waiting queue is pushed to call stack and then execute.

First-Class Function: A programming language is said to have First-class functions if functions in that language are treated like other variables. So the functions can be assigned to any other variable or passed as an argument or can be returned by another function.

Promises are used to handle asynchronous operations in JavaScript. They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code.

```
var promise = new Promise(function(resolve, reject) {
const x = "geeksforgeeks";
const y = "geeksforgeeks"
if(x === y) {
     resolve();
} else {
     reject();
}
});
promise.then(function () {
          console.log('Success, You are a GEEK');
     }).
     catch(function () {
          console.log('Some error has occurred');
     });
```

1. then()

then() is invoked when a promise is either resolved or rejected. It may also be defined as a career which takes data from promise and further executes it successfully.

Parameters:

then() method takes two functions as parameters.

1. First function is executed if promise is resolved and a result is received.
2. Second function is executed if promise is rejected and an error is received. (It is optional and there is a better way to handle error using *.catch() method*

```
.then(function(result){
        //handle success
    }, function(error){
        //handle error
    })
```

catch() is invoked when a promise is either rejected or some error has occurred in execution. It is used as an Error Handler whenever at any step there is a chance of getting an error.

Parameters:

catch() method takes one function as parameter.

1. Function to handle errors or promise rejections.(.catch() method internally calls .then(null, errorHandler), i.e. .catch() is just a shorthand for .then(null, errorHandler))

var → function'
let, const —> block

Arrow function

```
const square = number => number *number;
```

Spread operator

```
const first =[1,2,3]
const second=[4,5,6]
const combined = [...first, ...second, 7]
```

Whenever we have constructor in child class then we need to call constructor of parent class by super()

Async/await

The async and await keywords enable asynchronous, promise-based behaviour to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

The word “async” before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically.

The keyword `await` makes JavaScript wait until that promise settles and returns its result.

```
async function harry(){
console.log("inside");
const response = await fetch("'https://api.github.com/users");
console.log("before response");
const users =await response.json();
console.log("user resolved");
return  users;
}

console.log("before harry");
let a =harry();
console.log("after harry");
console.log(a);
a.then(data=> console.log(data));
console.log("last line of code");
```

output
before harry
inside
after harry
promise {<pending>}
last line of code
before response
user resolved

{ "tanu", "anu".......}

Slice Method

The `slice()` method returns selected elements in an array, as a new array.

The `slice()` method selects from a given *start*, up to a (not inclusive) given *end*.

The `slice()` method does not change the original array.

Data changw without mutation

```
var        = {score: 1, name: 'Jeff'};

 var newPlayer = {...player, score: 2};
```