Course On JavaScript->Jonas

Course On JavaScript

Notes is collection of concepts---> referance from Kyle Simpsons Book series and Articles

Notes and Full learning material: everything about javascript

Let, const and Var
Just remember that let is block scoped and the var is function scope. Const is way to define the constants

```
//define a arrow function
let i=true;
const aFunct=()=>{
    var p='Michal';
    if(i===true){
       var p='Jonas';
    }
    console.log(p);
}

aFunct();

o/p-> Jonas
```

with this example the var or variable p is accessable in the if block also so the var p have that whole functional scope here. any where we can able to update that variable p. And this kind of functionality proposed a inconsistency in coding and confusion so avoid using var when declaring and initialising a variable. User let and const.

```
//same code try with let and cheked what will we get
//define a arrow function
let i=true;
const aFunct=()=>{
    let p='Michal';
    if(i===true){
        let p='Jonas';
    }
    console.log(p);
}
```

```
aFunct();
o/p-> Michal
```

here the value of p defined inside of the if block is different and inside of function is different, so that is why let is block scope the p defined inside of the if block not manupulating the value of p that is defined inside the function.

Js have functionality of Dynamic Typing so one not have to define the type of variable. Just like Python

String and Template Literal

```
const firstName='saurabh';
const Job='Eng';
//string concatenation

const Jonas="I,m" + firstName + job;

const str=`I.m ${firstName}`;
console.log(str);

//new line

console.log('string with  \n\  multiple \n\ Lines');
//o/p-->
//string with
//multiple
//Lines

//using backticks it's way to more easy

console.log(`string
with
Lines
saparated`);
//that's all it will work fine
```

Conversion and the coersion

the conversion is manual conversion of type and in coersion js itself change the types and that is abstract from us totally.

```
const inputYear='1999';
console.log(inputyear + 18);
```

```
//Here it will concatenate the 18 with 1999
//so convert the string in Number

console.log(Number(inputYear)+18);
```

Operator == and ===

```
//simply just remember that the === is strict operator of equality and == is not the strict one

//EG
'19'== 19  IS TRUE
//but
'19'===19 IS FALSE

!== is also a Strict version of not Equal to
```

Checkout the loops of if, else if, else and swich case by your own.

falsy values

The undefine , NaN , 0, NULL , EMpty string ' ' are the values if that are converted to the boolean then they are false.

```
const Check=' ';

console.log(Boolean(Check));
--> False;

const EmptyObj={ };

console.log(Boolean(EmptyObj));
-->true

//I dont know why JS behave like that but rea the books of Kyle Simpson to know more about Javascrip
```

Switch Block

```
const day='Monday';

switch(day){
case 'Monday':
    console.log('Plan Exesucted');
    break;

case 'Thursday':
    console.log('sassy');
    break;

case 'Wednesday':
case 'Friday':
    console.log('The Party');
    break

default:
    console.log('Go to sleep');

}
```

Conditional Operator or ternary operator

```
const age=29;
 age>=19 ? console.log('Drink wine'):console.log('Do Not drink');

 or

 console.log(`Is this is the consent --> ${age>=19 ? 'Yes': 'NO'});
```

Do not break the WEB

JS Engine is backward compatible

The code written in 90's will still readable and compiled in modern JS engines. So old features are never ever removed.
But JS is not a Forward compatible,,,,

So during development use latest version of browser
During production: Use babel to trasnpile and polyfill your code ...Like converting backt to es5 to ensure

the compatibility for all the Users.

Activating the strict Mode in javascript

```
'use strict';
```

Function declaration and Expression

Funtion declaration works same but infact there are differences. Due to Hoisting the we can call the declared function before declaring it just because of JS assign the memory context first then code context

```
//Declaration
cal1(1998);
//As we have called it first then declare and this will work because of Hoisting
function cal1(BYear){
return 1999-BYear;
}


//Expression

const cal2=function(Byear){
return 1999-Byear;
}
const age1=cal2(2000);

//Now as if we called expression before declaring it it'll throw the referance error...That you have not initialised the function yet.
```

[ARTICAL]

```
Hoisting Function Expressions
Only function declarations are hoisted in JavaScript, function expressions are not hoisted. For example: this code won’t work.

helloWorld();  // TypeError: helloWorld is not a function
var helloWorld = function(){
  console.log('Hello World!');
}
```

```
As JavaScript only hoist declarations, not initializations (assignments), so the helloWorld will be treated as a variable, not as a function. Because helloWorld is a var variable, so the engine will assign is the undefined value during hoisting.

So this code will work.
```

So are let and const variables not hoisted?

The answer is a bit more complicated than that. All declarations (function, var, let, const and class) are hoisted in JavaScript, while the var declarations are initialized with undefined, but let and const declarations remain uninitialized.

They will only get initialized when their lexical binding (assignment) is evaluated during runtime by the JavaScript engine. This means you can't access the variable before the engine evaluates its value at the place it was declared in the source code. This is what we call "Temporal Dead Zone", A time span between variable creation and its initialization where they can't be accessed.

If the JavaScript engine still can't find the value of let or const variables at the line where they were declared, it will assign them the value of undefined or return an error (in case of const).

```
let a;
console.log(a); // outputs undefined
a = 5;



lexicalEnvironment = {
  a: <uninitialized>
}
```

Here during the compile phase, the JavaScript engine encounters the variable a and stores it in the lexical environment, but because it's a let variable, the engine does not initialize it with any value. So during the compile phase, the lexical environment will look like this in above;
Now if we try to access the variable before it is declared, the JavaScript engine will try to fetch the value of the variable from the lexical environment, because the variable is uninitialized, it will throw a reference error.

During the execution, when the engine reaches the line where the variable was declared, it will try to evaluate its binding (value), because the variable has no value associated with it, it will assign it undefined.

So the lexical environment will look like this after execution of the first line:

```
lexicalEnvironment = {
  a: undefined
}
```

And undefined will be logged to the console and after that 5 will be assigned to it and the lexical environment will be updated to contain the value of a to 5 from undefined.

Note — We can reference the let and const variables in the code (eg. function body) even before they are declared, as long as that code is not executed before the variable declaration.

For example, This code is perfectly valid.

```
function foo () {
  console.log(a);
}
let a = 20;
foo();  // This is perfectly valid
```

```
But this will generate a reference error.

function foo() {
 console.log(a); // ReferenceError: a is not defined
}
foo(); // This is not valid
let a = 20;
```

Remember in case of let and const it manpulate or behave not the same as it behave with var.

```
console.log(fuct);  -->Undefined

var fuct=function(){
    console.log('Saurabh Mulau');
}
```

This code or it log a undefined value on the console.

```
console.log(fuct);  -->Referance Error

let fuct=function(){
    console.log('Saurabh Mulau');
}
```

```
console.log(fuct);  -->Referance Error

const fuct=function(){
```

```
    console.log('Saurabh Mulau');
}



//so in case of let and const again it has been uninitialised lexically when we are trying to access them
```

More on JS Execution

What is an Execution Context?
Simply put, an execution context is an abstract concept of an environment where the Javascript code is evaluated and executed. Whenever any code is run in JavaScript, it's run inside an execution context.

Types of Execution Context
There are three types of execution context in JavaScript.

Global Execution Context — This is the default or base execution context. The code that is not inside any function is in the global execution context. It performs two things: it creates a global object which is a window object (in the case of browsers) and sets the value of this to equal to the global object. There can only be one global execution context in a program.

Functional Execution Context — Every time a function is invoked, a brand new execution context is created for that function. Each function has its own execution context, but it's created when the function is invoked or called. There can be any number of function execution contexts. Whenever a new execution context is created, it goes through a series of steps in a defined order which I will discuss later in this article.
Eval Function Execution Context — Code executed inside an eval function also gets its own execution context, but as eval isn't usually used by JavaScript developers, so I will not discuss it here.

Execution Stack
Execution stack, also known as "calling stack" in other programming languages, is a stack with a LIFO (Last in, First out) structure, which is used to store all the execution context created during the code execution.

When the JavaScript engine first encounters your script, it creates a global execution context and pushes it to the current execution stack. Whenever the engine finds a function invocation, it creates a new execution context for that function and pushes it to the top of the stack.

The engine executes the function whose execution context is at the top of the stack. When this function completes, its execution stack is popped off from the stack, and the control reaches to the context below it in the current stack.

How is the Execution Context created?
Up until now, we have seen how the JavaScript engine manages the execution context, Now let's understand how an execution context is created by the JavaScript engine.

The execution context is created in two phases: 1) Creation Phase and 2) Execution Phase.

The Creation Phase
The execution context is created during the creation phase. Following things happen during the creation phase:

LexicalEnvironment component is created.
VariableEnvironment component is created.

```
ExecutionContext = {
  LexicalEnvironment = <ref. to LexicalEnvironment in memory>,
  VariableEnvironment = <ref. to VariableEnvironment in  memory>,
}
```

Lexical Environment
The official ES6 docs define Lexical Environment as

A Lexical Environment is a specification type used to define the association of Identifiers to specific variables and functions based upon the lexical nesting structure of ECMAScript code. A Lexical Environment consists of an Environment Record and a possibly null reference to an outer Lexical Environment.

Simply put, A lexical environment is a structure that holds identifier-variable mapping. (here identifier refers to the name of variables/functions, and the variable is the reference to actual object [including function object and array object] or primitive value).

Each Lexical Environment has three components:

--Environment Record
--Reference to the outer environment,
--This binding.

Environment Record
The environment record is the place where the variable and function declarations are stored inside the lexical environment.

There are also two types of environment record :

Declarative environment record — As its name suggests stores variable and function declarations. The lexical environment for function code contains a declarative environment record.

Object environment record — The lexical environment for global code contains a objective environment record. Apart from variable and function declarations, the object environment record also stores a global binding object (window object in browsers). So for each of binding object's property (in case of browsers, it contains properties and methods provided by browser to the window object), a new entry is created in the record.

Read the detailed articals on execution context

Arrow Func

```
const Calage2=function(byear){
return 2017-byear;
}
//return in the arrow function happened implicitely, we do not have to write return keyword for a single line of fucntion

const calAge3=byear=>2017-byear;

const age=calAge3(1991);
console.log(age); -->Your calcuated Answer


//It'll get somewhat complicated for more parameters

const ageofret=byear=>{
    const age=2017-byear;
    const retire=65-age;
    return retire;
//here we have to explicitely mentioned return keywrd because it is not a one line function

}

//Arrow function witthh multiple parameter

const ageofret=(fName,byear)=>{
    const age=2017-byear;
    const retire=65-age;
    return `${Fname} retire in ${retire} years`;

//here we have to explicitely mentioned return keywrd because it is not a one line function
```

```
}

//Hoisting

//Yes we can say but in case of var only, It'll treat the arrow function as
a variable

console.log(fuct); -->Undefined

var fuct=()=>{
    console.log('fuck');
}
```

More on Hoisting

```
const x=1;

{
    const x=0;
    {
        //const x=8;
        console.log(x);

    }



}

O/p->> 0
```

Function calling another Function

```
//function calling other functions

function cut(fruit){
    return fruit*4;

}


function fruits(apples,oranges){
   //here simply we have called one function in another
```

```
    const applePieces= cut(apples);
   const orangePieces=cut(oranges);
   console.log(applePieces,orangePieces);

    const juice=`juice with ${apples}  and ${oranges}`;
    return juice;

}
fruits(3,4);


-->12 16
```

Arrays in Javascript

```
//Here in Javascript the array can hold multiple types of data

const friends=['Michal','steven','Jonas',1,'c',2.3,undefined, NaN, null];
friends.forEach((Ele)=>{
    console.log(typeof Ele);
})




index.js:3 string
index.js:3 number
index.js:3 string
index.js:3 number
index.js:3 undefined
index.js:3 number
index.js:3 object

//check out the types of array element


//Another way to create a new array

const years=new Array(1991,1912,9122,1991);
//length of array
years.length;

//last element
```

```
years.length-1

//array can also have or can add data type of arr

//like

const friend=['sas','sd'];
const G=['man',friends];
//if we have printed it

-->'man'
-->Array(2);
```

Basic Array Operations

Useful array method

```
const friends=['mice','cat','peter'];
//both method pop and push from rear
friends.push('hauser');

friends.pop();


//unshift add the element in front

friends.unshift('peTos');

//shift remove the element from front

friends.shift();


//position of element in the array

friends.indexOf('stevan');
//it will return -1 if element not there in aray

//method includes return bool type

friends.includes('bob');

//remember it is a strict operator so no type coersion
```

Objects:::

```
//OBJECTS
// Objects
const jonas={
    fname:'saurabh',
    Lname:'mulay',
    age:129,
        friends:['michal','Danny','Srh']


};
//Accessing value through dot notation


console.log(jonas.fname);
//The key value pairs inside of the Js object is nothing but properties
associated with that object
//order of the properties does not matter over here in Objects


//bracket Notation

console.log(jonas['Lname']);
//it is same as accesssing key in java or in apex like map.get(__key__);
```

```
//fetching values get from user
// Objects
let userinput=prompt("what do u want to know about jonas");

const jonas={
    fname:'saurabh',
    Lname:'mulay',
    age:129

};
```

```
console.log(jonas.userinput);
//now in case of let and const the values is blocked scopes and they are in temporal dead zone so window object cannot able to fetch the, same the value of jonas.userinput is "undefined" in this case now.
```

now in case of let and const the values is blocked scopes and they are in temporal dead zone so window object cannot able to fetch the, same the value of jonas.userinput is "undefined" in this case now.

```
//That is why we need a Bracket Notation  to access out the values

let userinput=prompt("what do u want to know about jonas");

const jonas={
    fname:'saurabh',
    Lname:'mulay',
    age:129

};

console.log(jonas[userinput]);

-->input- fname
o/p->>saurabh
```

adding new properties to the object

```
jonas.location='San Jos';
jonas['Insta']-'@jonas123';

console.log(jonas);
```

Object Methods

```
const Jonas={
    Fname:'Jonas',
    Lname:'sm',
    age:10,
    job:'IT',
```

```
    calAge: function(Byear){
    return 2017-Byear;
    }

};

//We can attach a function as a property in the object also, only function declaration will not wwork here so only function expression gonna help
```

Another way to access Object

```
const jonas={
    fname:'saurabh',
    Lname:'mulay',
    age:129,
    calage:function(Byear){
        return Byear-1998;
    }

};

console.log(jonas.calage(2000));

//another way to access value

console.log(jonas['calage'](1999));
```

Higher Order Functions in JS

In simple words, The Javascript treat the functions as FIrst Class Citizens!!! But why, because in any functional programming language the function as treated as a Objects.....In js Functiions are special type of Objects, why?? Because we can pass the function inside a function, accept a function as a argument inside a function and also we can set the properties in the functions as we set in objects. ** **we can try to get that thing by programming it

```
arr=['saurabh.','Masei',1,12,3,43,1,23,11];

const arrFunct=(arr)=>{
    arr.forEach(element => {
        console.log(element);

    });
```

```
    location1:'USA'


}
arrFunct.Os='Linux';
console.log(arrFunct.Os);    -->O/P Linux
arrFunct(arr);
//console.log(arrFunct['location1']);
-->Undefined
```

I dont know Why js not allow me to access the property of function funct , but i can set the property using function name. Like I have done for arrFunct.Os:
Inshort we know that now, the functions are nothing but objects in js. we have also seen that Function expression can be assign to a variable and in Hoisting it behave like a Variable.

Higher Order Function

Higher order functions are the functions that operates on other functions, either by taking them as a argument or by returning them. **So the higher order function is function that receive the function as argument and it returns the function as o/p.**
For Example

```
Array.prototype.map;
Array.prototype.filter;
Array.prototype.reduce;
//This are the higher order functions and this functions accept functional args
```

Examples:

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

//Array.prototype.map

const newArray=arr.map((index)=>index*2);
console.log(newArray);


//0/p-> [2,4,6,8]
```

We can filter on basis of date, string...But will cover the functions or different method available for those types

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

//Array.prototype.map
let target=3;
const newArray=arr.filter((index)=>index>2);
console.log(newArray);
//-->[3,4];
```

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

//Array.prototype.map
let target=3;
const newArray=arr.reduce((accumulator, currentvalue)=>{
    return accumulator+currentvalue;
});

console.log(newArray);

//Incase when the current value is not present the accumulator will auto
assign to the first value of array and the current value is second value,
//in every iteration when the function reduce called it track return sum and
it store that in a accumulator.


//What if the initial value is mentioned
const arr=[1,2,3,4];

//Array.prototype.map
let target=3;
const newArray=arr.reduce((accumulator, currentvalue)=>{
    return accumulator+currentvalue;
},100);

console.log(newArray);

//answer-->110
```

In a nutshell, a Higher-order function is a function that may receive a function as an argument and can even return a function. Higher-order functions are just like regular functions with an added ability of receiving and returning other functions are arguments and output.

Lets Build a Small Game

Game 1: Guess the Number - In perticular range given

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>

<body>

    <header>
        <h1>Guess My Number</h1>
        <p class="message">(Between 1 - 20) </p>
        <button class="again">Again</button>
        <div>??????</div>

        <h2  class="number">12</h2>


    </header>

    <main>
        <section>
            <input class="guess" type="number" />
            <button class="ClickMe">check</button>

        </section>
        <section>

            <p>start Guessig</p>
            <p >score :: <span class="score">20</span></p>
            <p> HighScore : <span class="Hscore">0</span></p>

        </section>
    </main>
    <script src="index.js"></script>
</body>
</html>
```

Guess My Number

(Between 1 - 20)

Again

??????

check

start Guessig

score :: 20

HighScore : 0

Remember that the DOM Is not a JS, they are web API's same for all the browsers. There are lot more we'll catch them and understand them fully in next Articles.

<<<--- Before implementing the game logic let's play with Javascript to manupulate the dom --->>>

```
console.log(document.querySelector('.message').textContent);

//selecting the selctor of class name message;
//we  can also able to set the textcontent
const

message=document.querySelector('.message').textContent='Yo, right Number!!! Yu win bc!';
console.log(message);

//Now checkout on the screen we actually have manupulated the DOM, The text has been changed to what we have changed.

document.querySelector('.number').textContent=19;
document.querySelector('.score').textContent=12;

const val=document.querySelector('.guess').value;
console.log(val);

//to set the value of that element

//use
```

```
document.querySelector('.guess').value=14;

const val1=document.querySelector('.guess').value;
console.log(val1);



//Handling a button, Event

//document.querySelector('.ClickMe').addEventListener('__EventName__',
(__MEtho__That goint to execute)=>{})

document.querySelector('.ClickMe').addEventListener('click',()=>{

const guess=Number(document.querySelector('.guess').value);
console.log(typeof guess);
//it;s a string so typecasting it to Number()
//we have to compare this guess with our randomly generator number
if(!guess){
    console.log('NULL GUESS!!!!');
    //and show on screen also
    document.querySelector('.message').textContent='NO Number !!!!'

}
});
```

Guess My Number

NO Number !!!!

Again
??????

19

check

start Guessig

score :: 12

HighScore : 0

Guess My Number

Yo, right Number!!! Yu win bc!

Again
??????

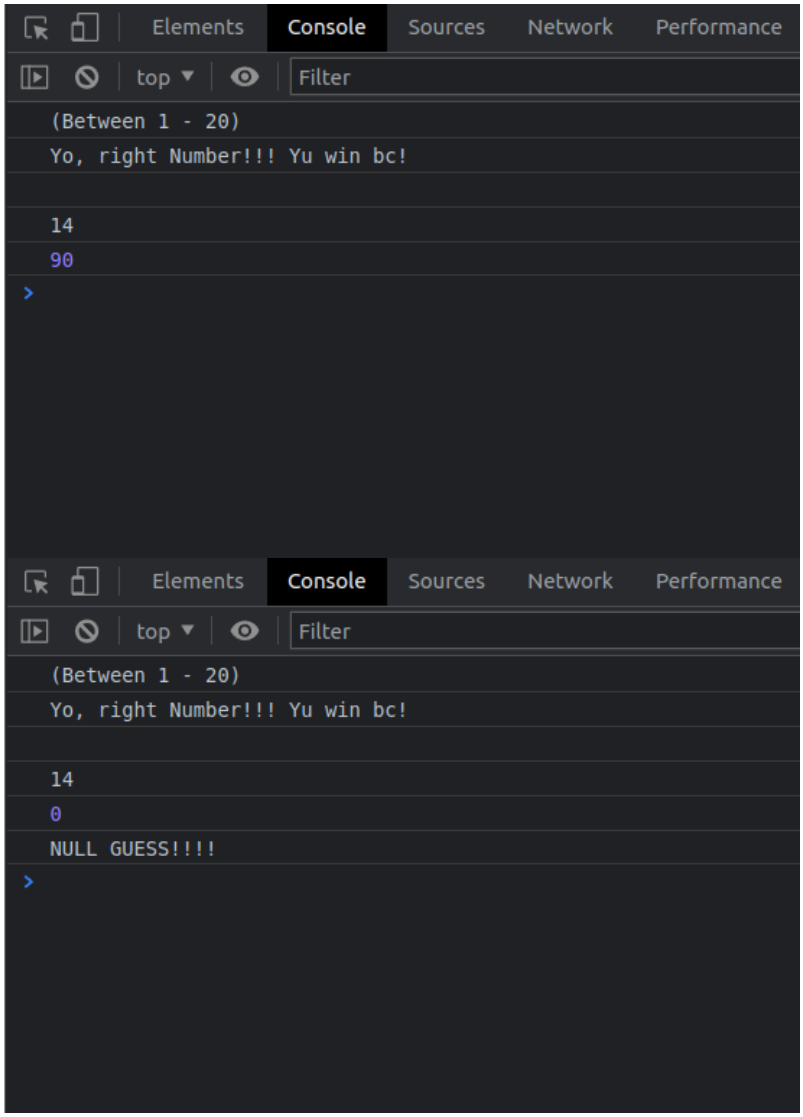
19

90 check

start Guessig

score :: 12

HighScore : 0



Guess My Number

NO Number !!!!

Again
??????

19

00 check

start Guessig

score :: 12

HighScore : 0

Implementing a Game Logic

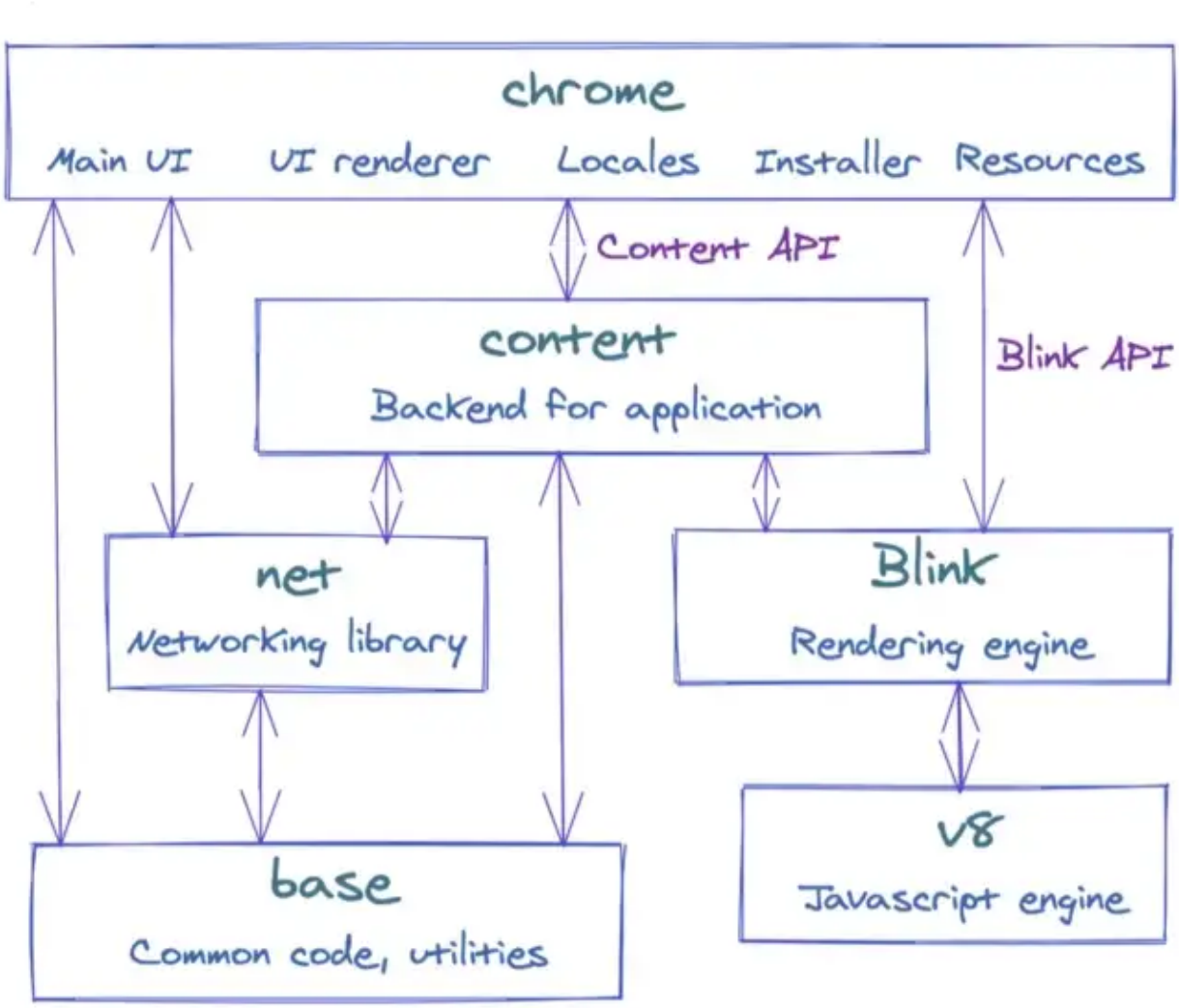
```
//Three Scenarios when guess is low high and exact same//the Math.random
```

```
gives us random numbers ranging from 0-1 so mul it with 20 gives number upto
20
//floor or ceil those numbers
//compare the number with users guess
const snumber=Math.floor(Math.random()*20);
let sscore=20;

let HigheScore=0;
document.querySelector('.ClickMe').addEventListener(
    'click', ()=>{
        const guess=Number(document.querySelector('.guess').value);
        console.log(guess, typeof guess);
        console.log(this.snumber);

        if(!guess){
            console.log('NO Number');
            document.querySelector('.message').textContent='Number Selected
Null';
        }
        else if(guess>snumber){
            if(sscore>1){
            document.querySelector('.message').textContent='You have
selected High';
            sscore--;
            document.querySelector('.score').textContent=sscore;
            }
            else{
                document.querySelector('.message').textContent='You have
Loss the game';
            }

        }
        else if(guess<snumber){
            document.querySelector('.message').textContent='You have
selected Low';
            sscore--;
            document.querySelector('.score').textContent=sscore;

        }
        else if(guess==snumber){
            document.querySelector('body').style.backgroundColor='#60b347';
            document.querySelector('.message').textContent='Perfect You won
the game';
```

```
            document.querySelector('.number').textContent=snumber;
            if(sscore>HigheScore){
                HigheScore=sscore;
                document.querySelector('.Hscore').textContent=HigheScore;

            }




        }

    }
);

document.querySelector('.again').addEventListener('click',

()=>{
    const snumber=Math.floor(Math.random()*20);
    let sscore=20;
    document.querySelector('.message').textContent='Start guessing!!(Between
1 - 20)';
    document.querySelector('.score').textContent=sscore;
    document.querySelector('.number').textContent=' ??? ';
    document.querySelector('.guess').value=' ';
    document.querySelector('body').style='#222';


}


)
```

This code isn't clean one. so there are to much inconsistencies and it has violated DRY principle, it's just basics but do not write code like that.

More On JS Engine



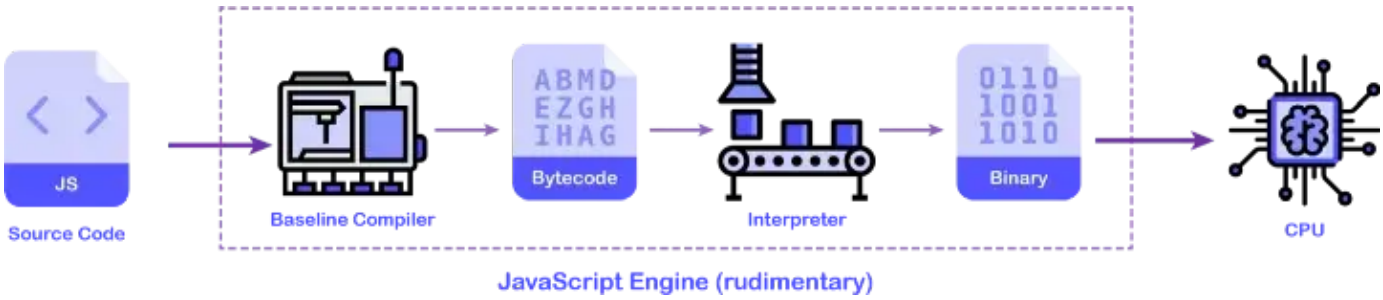
Blink is a rendering engine that is responsible for the whole rendering pipeline including DOM trees, styles, events, and V8 integration.

we are going to take a look at the JavaScript engine and its anatomy. We are also going to learn about JavaScript's call stack, event loop, task queues, and various other pieces that make JavaScript as we know it works properly

JavaScript in a nutshell
JavaScript is an interpreted language. This means we do not have to compile the JavaScript source code before sending it to the browser. An interpreter can take the raw JavaScript code and run it for you.

JavaScript is also a dynamically typed language, unlike C and C++. This means variables declared using var can store any type of data type like int, string, boolean and also complex data types like object and array.

The lack of type system is what makes JavaScript slow to run. A statically typed language can produce a much efficient machine code because of the information it has about the data like its type and size.



JavaScript Engine (rudimentary)

As you can see from the diagram above, the job of the first JavaScript engine was to take the JavaScript source code and compile it to the binary instructions (machine code) that a CPU can understand.

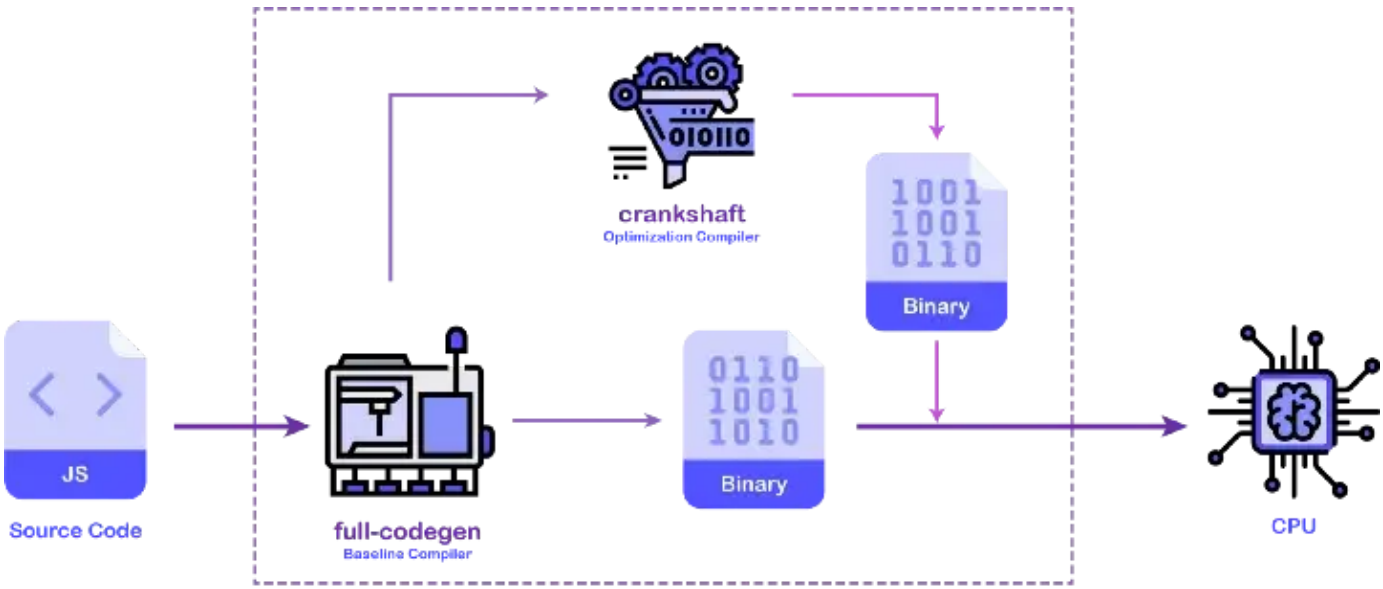
A rudimentary JavaScript engine contains a baseline compiler whose job is to compile JavaScript source code into an intermediate representation (IR) which is also called the bytecode and feeds this bytecode to the interpreter.

The interpreter takes this bytecode and converts to the machine code which is eventually run on the machine's hardware (CPU).

A baseline compiler's job is to compile code as fast as possible and generate less-optimized bytecode (or machine code in other cases). Since the interpreter has an unoptimized bytecode to work with, the application speed will be slow, however, the application bootstrap time will be very less.

When it comes to a highly dynamic and interactive web application, the user experience is very poor with this model of JavaScript execution. This problem was faced by Google's Chrome browser while displaying Google Maps on the web. To increase the JavaScript performance on the web, they had to come up with a better approach.

Google Chrome from the early days uses the V8 JavaScript engine. In the beginning, to improve the JavaScript performance, they added two pieces in their JavaScript engine pipeline as shown below.



V8 JavaScript Engine (2010)

he full-codegen was the baseline compiler whose job was to spit out unoptimized machine code as fast

as possible for faster application bootstrap.

As the application was running, the crankshaft compiler would kick in and optimize the source code and replace the parts of the machine code generated by the baseline compiler. This optimization would result in better application performance as better and better machine code is generated.The above version of the JavaScript engine does not contain an interpreter. This is a JIT (Just-In-Time) compilation model as code is compiled to the machine level on the fly and later optimized, also to the machine code.

How JavaScript is optimized?
There are various criteria for optimizing JavaScript code. Before JavaScript code is passed to the interpreter or baseline compiler, it has to first get parsed into an Abstract Syntax Tree (AST) which is a tree-like structure of the code.

When we run a JavaScript application, we do not need all the code at the application startup time. For example, if we have a function that is called on the user action, like a button click, that code can be parsed later.

Identifying things that need to be parsed immediately and generating machine code is the best strategy for faster application bootstrap.

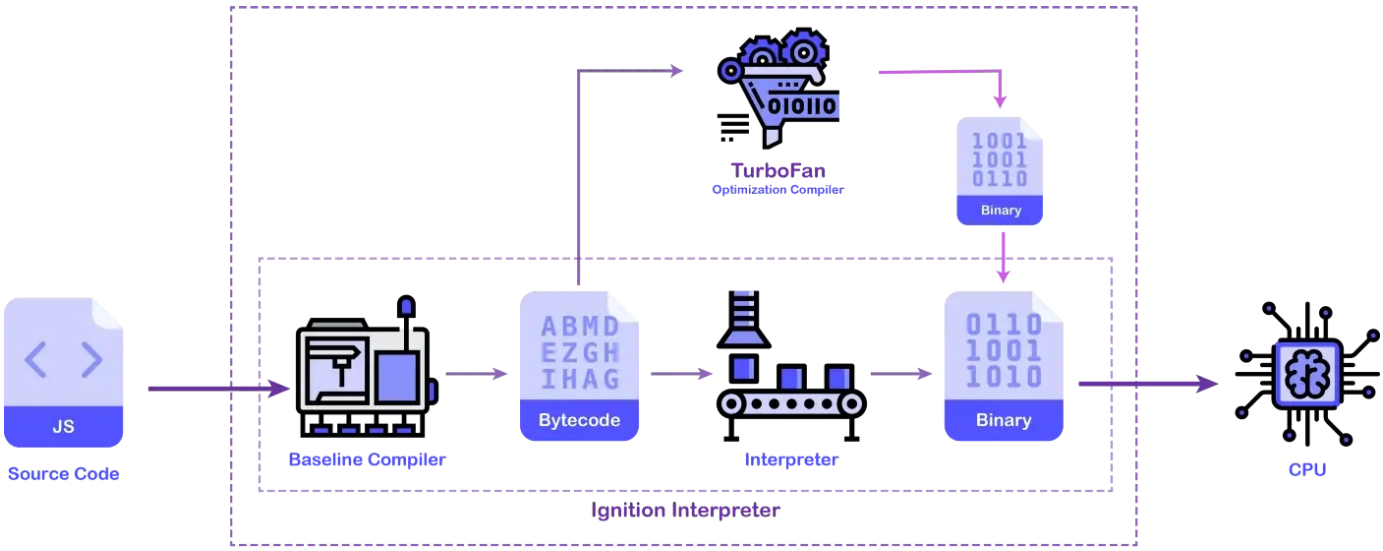
Sometimes, JavaScript code contains unnecessary complex logic that can be simplified. For example, a for loop to increment an integer can be inlined using + operations n number of times. This process is called Loop unrolling. Similar optimizations can be made using function inlining

The lack of type system in JavaScript is what makes JavaScript engine produce less optimized machine code. Hence, based on already defined values, a JavaScript engine can guess the data types of the variables and generate better machine code.

This whole process is documented and explained very well by Paul Ryan in his blog post on the V8 engine on alligator.io. You should definitely check read this article to understand these concepts in depth.

Meanwhile, what JavaScript engine can also do is gather profiling data of the code execution and look for the code that runs slower. This code is called the “Hot” code perhaps because it burns the CPU. This code can be further optimized and replaced with an optimized machine code.

Considering these things in mind and other problems caused by full-codegen and crankshaft, the V8 team created a new version of the V8 engine from the ground up. This new version of the JavaScript engine was released in 2017.



V8 JavaScript Engine (2017)

The TurboFan optimization compiler can optimize this bytecode in the background (in separate threads) as the application is running and generate a very optimized machine code that will be replaced eventually.

Turbofan receives the profiling data from the Ignition interpreter and looks for the code that is Hot. It can make the guesses on how to optimize the code better (by guessing the data types) and optimize or de-optimize the code.

Js at Runtime

But unlike other programming languages, it's single-threaded langauge at runtime. That means the code execution will be done one piece at a time. Since code execution is done sequentially, any code that takes a long time to execute will block anything that needs to be executed after that. Hence sometimes you see below the screen while using Google Chrome.

When you open a website in the browser, it uses a single JavaScript execution thread. That thread is responsible to handle everything, like scrolling the web page, printing something on the web page, listen to DOM events (like when the user clicks a button), and doing other things.

But when JavaScript execution is blocked, the browser will stop doing all those things, which means the browser will simply freeze and won't respond to anything until that task is completed.

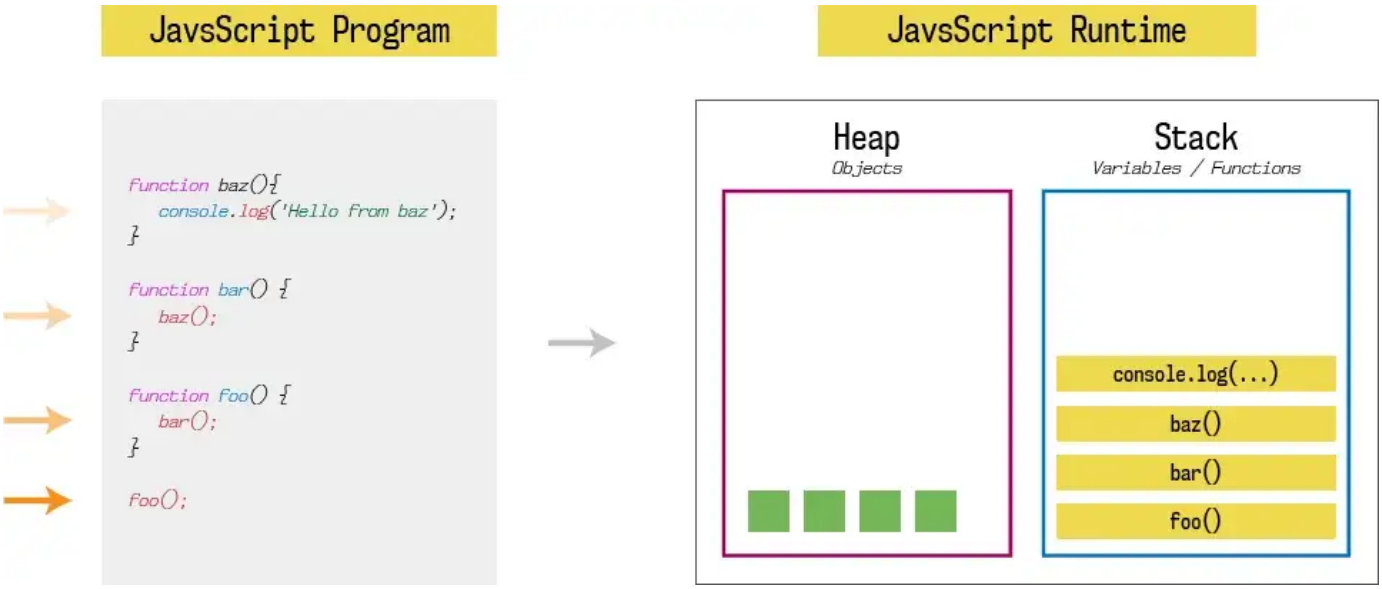
You can see that in action using below eternal while loop.

while(true){}
Any code after the above statement won't be executed as the while loop will loop infinitely until system is out of resources. This can also happen in an infinitely recursive function call.

Thanks to modern browsers, as not all open browser tabs rely on single JavaScript thread. Instead, they use separate JavaScript thread per tab or per domain. In the case of Google

Chrome, you can open multiple tabs with different websites and run above the eternal while loop.
-->That will only freeze the current tab where that code was executed but other tabs will function normally. Any tab having page opened from the same domain / same website will also freeze as Chrome implements a one-process-per-site policy and a process uses the same JavaScript execution thread.

Code Execution, Heap and Stack



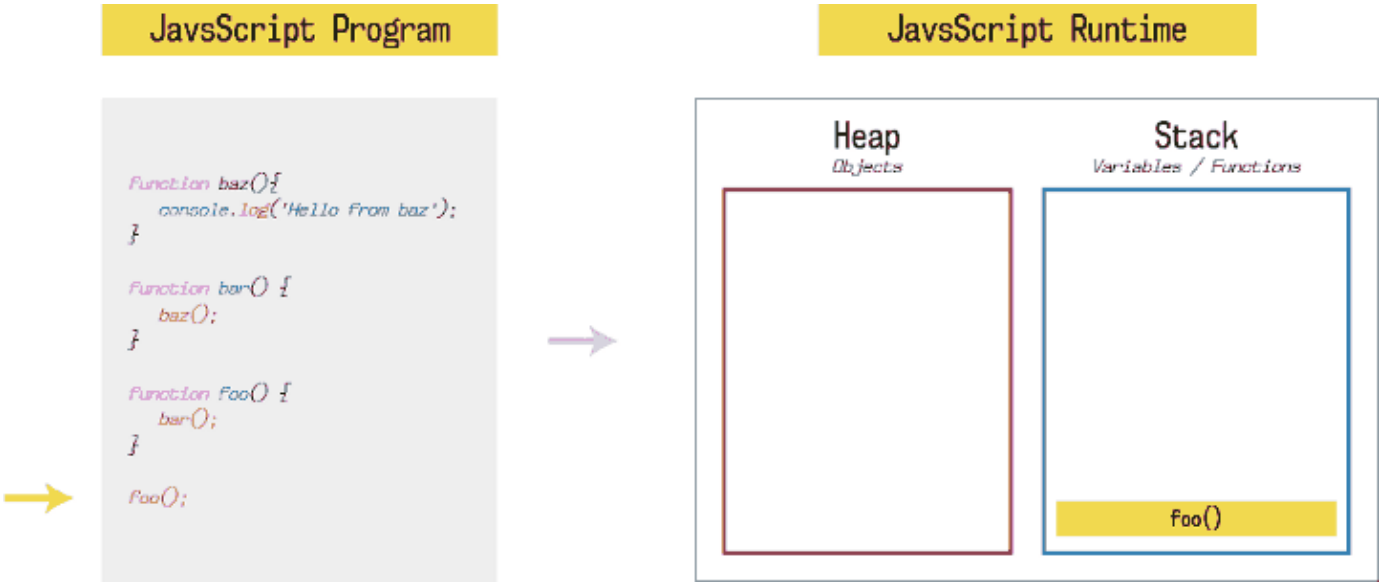

```
function baz() {
  console.log( 'Hello from baz' );
}

function bar() {
  baz();
}

function foo() {
  bar();
}

foo();
```

Here we have a simple JavaScript program that has three functions, viz. foo, bar and baz. The function foo calls the function bar and then function bar calls the function baz which logs something to the console using the console.log function provided by the runtime.
Like any other programming language, JavaScript runtime has one stack and one heap storage. A heap is a free memory storage unit where you can store memory in random order. Data that is going to persist in for a considerable amount of time go inside the heap. Heap is managed by

the JavaScript runtime and cleaned up by the garbage collector.What we are interested in is stack. A stack is LIFO (last in, first out) data storage that stores the current function execution context of a program. In the above example, when our program is loaded into the memory, it starts execution from the first function call which is foo().

Hence, the first stack entry is foo(). Since foo function calls bar function, second stack entry is bar(). Since bar function calls baz function, third stack entry is baz(). And finally, baz function calls console.log, fourth stack entry is console.log('Hello from baz').

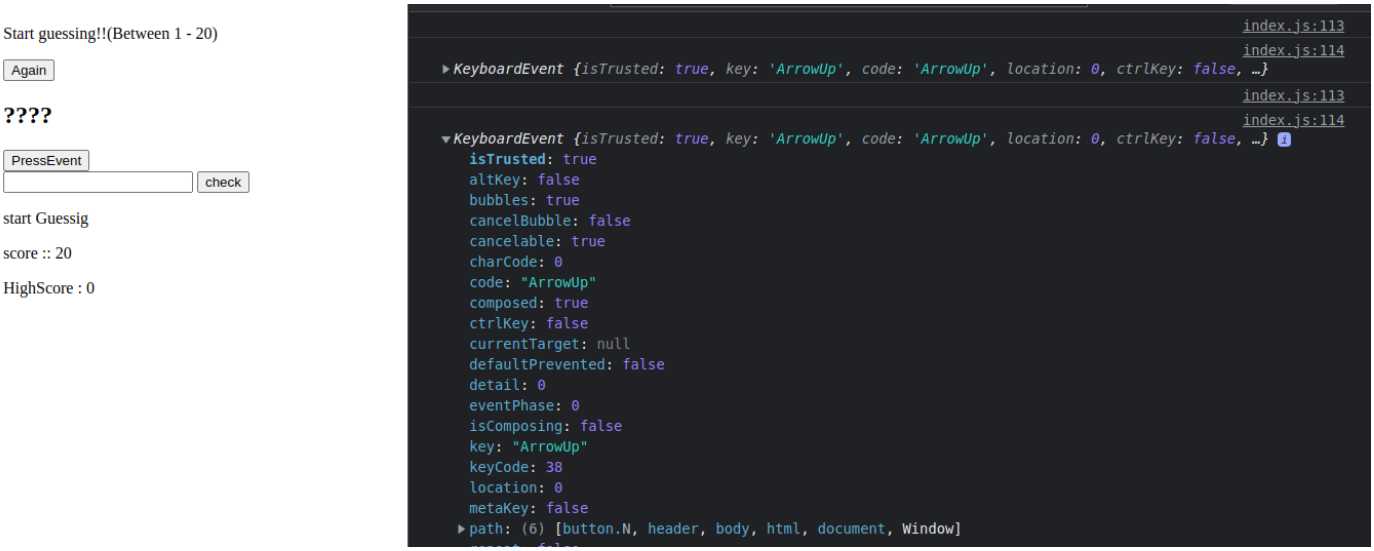
Until a function returns something (while the function is executing), it won't be popped out from the stack. The stack will pop entries one by one as soon as that entry (function) returns some value, and it will continue pending function executions.



Handling Keyboard Events

Using keydown event inside of the event listner we can actually invoke a function if user have pressed any key on keyboard.

```
<button class='n'/>


//js
document.querySelector('.n').addEventListner('keydown',(event)=>{
console.log(event);
})

//event is a obj that has been passed to the function as a arg through js ,
using that event we can figure out which key has been pressed by user.
```



Let's log that event:

Higher level overview of Javascript

Js is a high level , prototype bases object oriented , multi paradigm, interpreted or just in time compiled, dynamic , single threaded , garbage collected programming languaged with First Class Functions and Non blocking event loop concurrency model.

--> This defination combines what we are goin to learn Now...

High Level -->

We do not have to manage the resources so it makes language easy to learn but it slows down.

Garbage Collector-->
we do not have to care about what we have hold in heap or in memory.(cleaner)

Intrepreted or Just In Time compiled-->
Just-in-time compilation is a method for improving the performance of interpreted programs. During execution the program may be compiled into native code to improve its performance. It is also known as dynamic compilation.

Multi paradigm-->
procedural programming
funcional programming
Object oriented programming

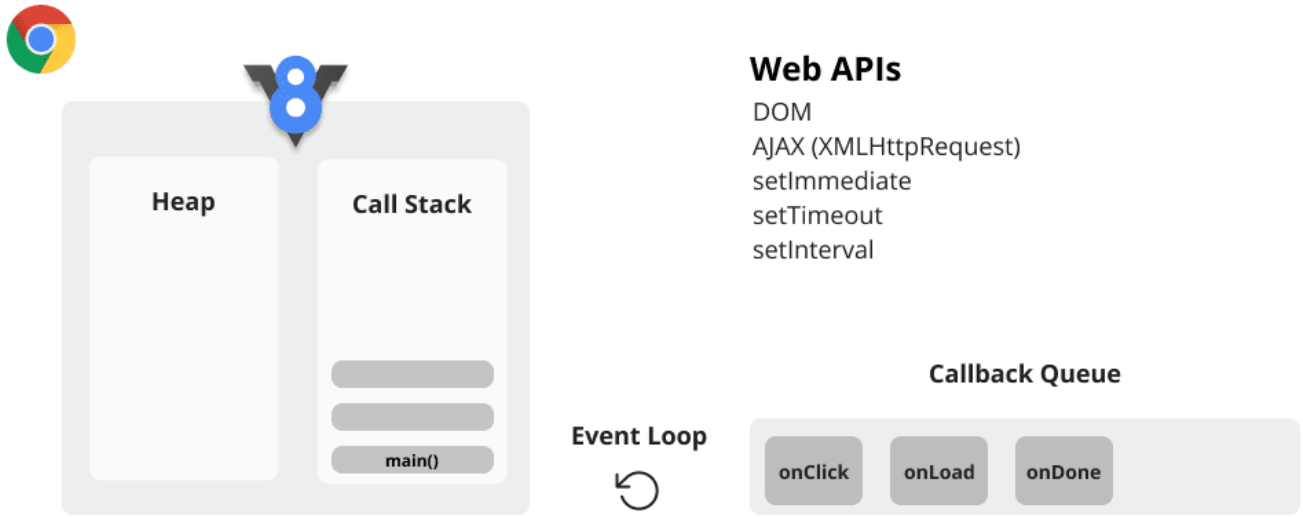
First class functions-->
we already have discuss them in above sections.

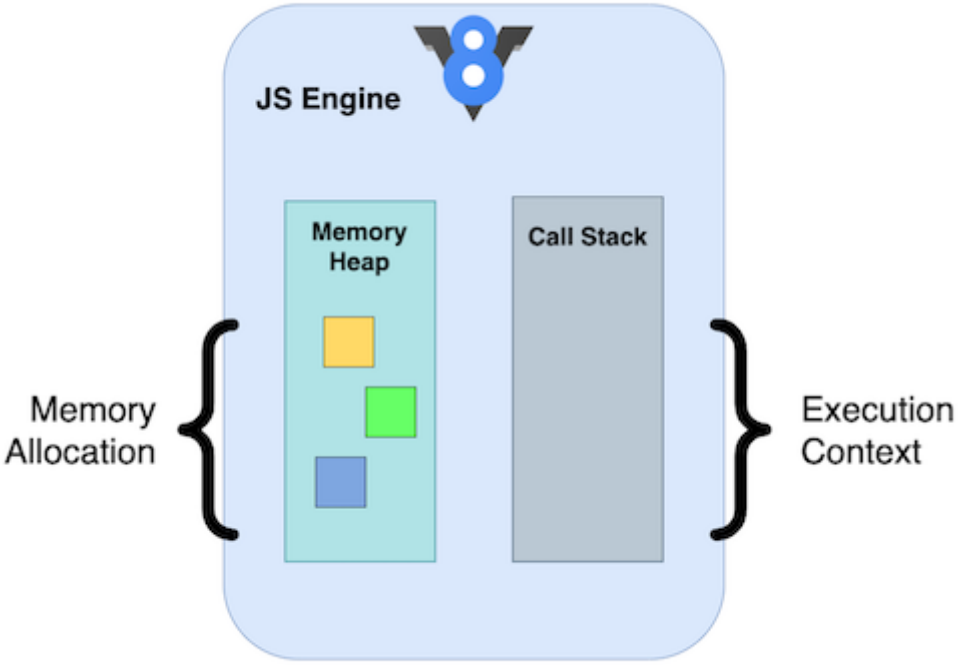
Js is Dynamic-->
it's dynamically types, we do not have to mentioned type of variable. it will assign at runtime.

Single Thread--> and<-- Non Blocking Event Loop (concurrency Model)
One process at a time.
concurrency Model: How the js engine handles multiple tasks happending at a same time.
But js is single thread so only can do one thing at a time.
But, What if the long Task???? Is it block the single or main thread?? **No, event loop takes a running task and execute them in the bacground and then puts all of them in the main thread once they finished.**

JS Engine and Runtime

Chrome or chromium based browsers used the V8.





It's easy to understand the stack and heap, Callstack is where our code get executed in execution context.Heap is unstructure memory pool where our objects get stored.

Compilation vs interpretation

Compilation-->

entire code is converted into the machine code at once and written to a binery file that can be executed by a computer.

Interpretation-->

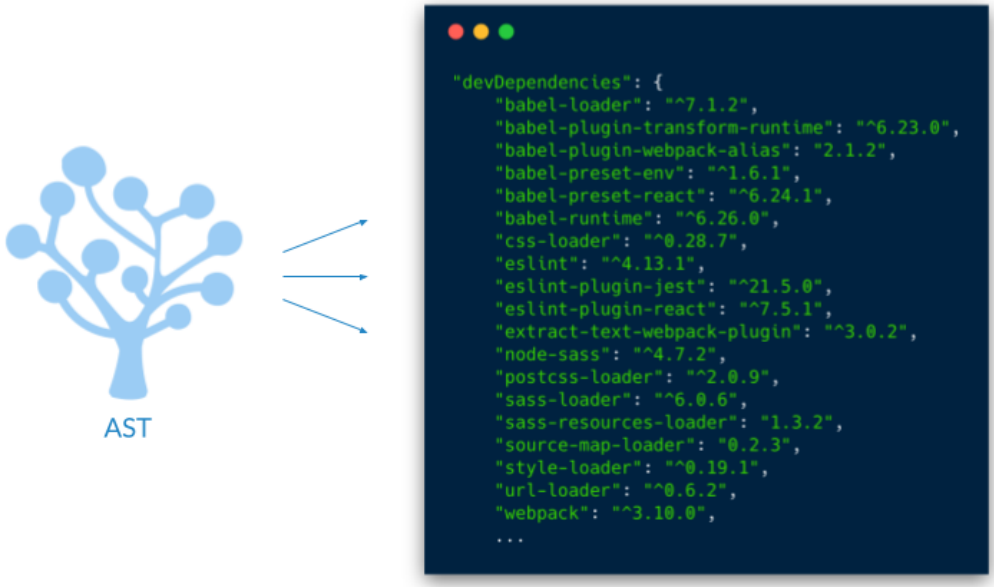
It gets executed line by line.

Js is used to be the interpreted language and the problem with interpreted languages that they are much slower in speed. So Low performance is not acceptable. **Js is not fully interpreted now. Modern Js use Just In time Compilation..**

So what is JIT compilation... Entire code has been converted into the machine code at once, then executed immediately.

Source code-->compilation-->Machine code--> execution-->programming running.

It saves source code in Abstract syntx trees.(AST), (In structured way).
Later on the AST used to convert the code in to machine code.

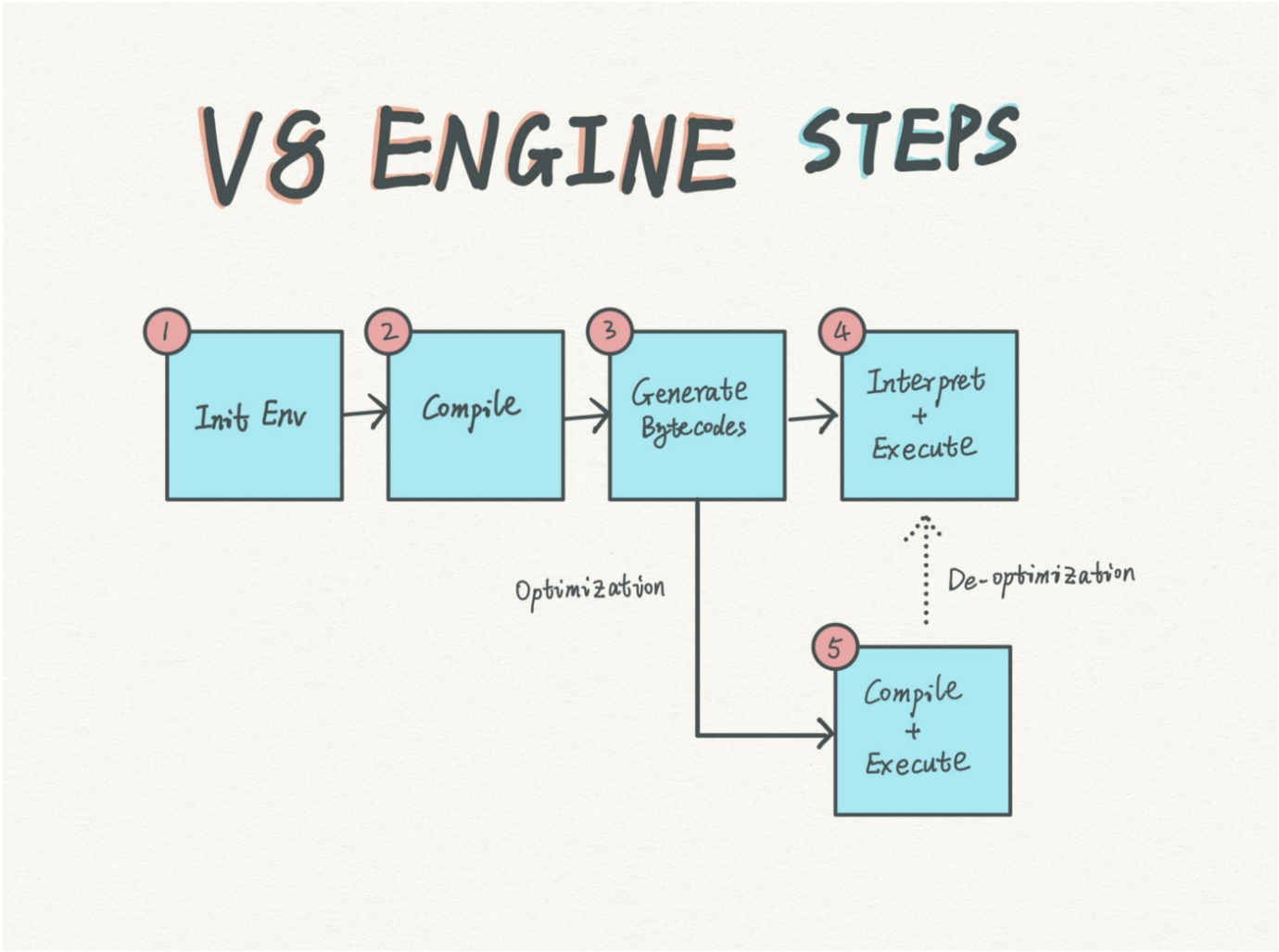


0•X

lastly execution happens in CallStack.

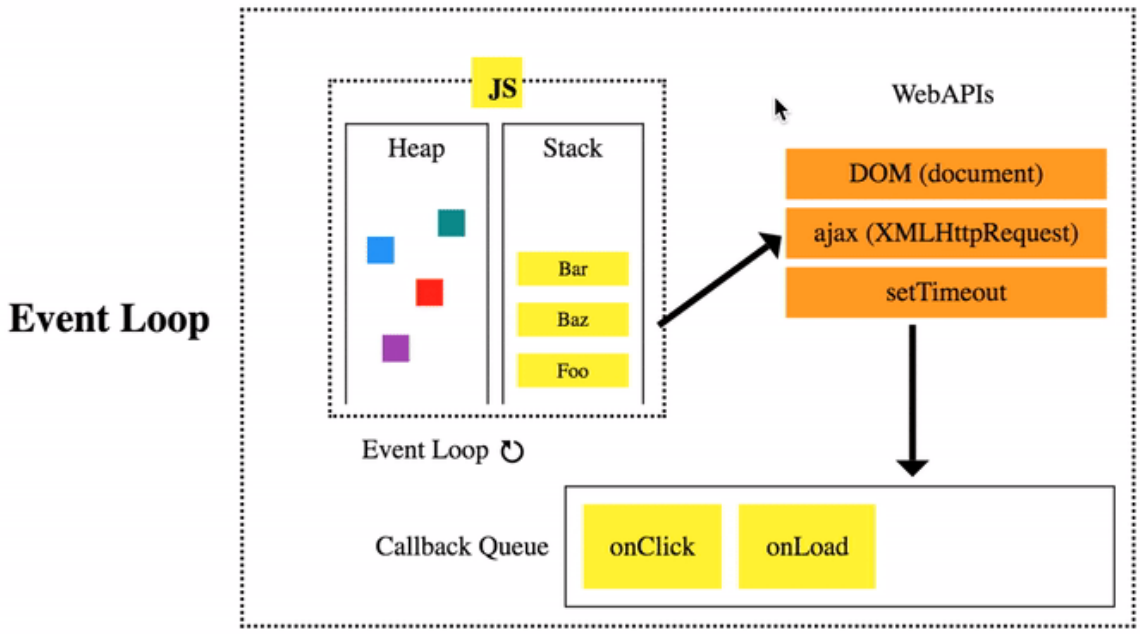
Optimization -->>
Js generate less optimised code first for the fast execution. it;s very unoptimised. So at execution context or at execution level the code / machine code gets reoptimised again and again.Unoptimised code has swapped by very optimised code every time. How ?? But js is single thread?!!! --> No! It

happens in the special threads that we cant access from our code.



JS engin is not alon enough, the web API's also provides the functioalities to the engine

CallBack QUeue



The event loop is a mechanism for scheduling the execution of JavaScript code. It ensures that the code that needs to run in response to an event, such as a user clicking on a button, is

executed at the appropriate time. The callback queue is a queue of functions that are waiting to be executed by the event loop. When an event occurs, the callback function for that event is added to the callback queue, and the event loop will execute that function when it is ready. This allows for asynchronous execution of JavaScript code, which can be useful for improving the performance of web applications.

Javascript is a single-threaded language.

The Event-loop is the secret sauce that helps give Javascript its multi-tasking abilities (almost!).

This loop constantly checks whether the call stack is empty or not and if it is, the functions waiting to be executed in the callback queue get pushed to the call stack.

Callback Queue

The Callback queue is a data structure that operates on the FIFO (first-in-first-out) principle.

Callback functions that need to be asynchronously executed, are pushed onto the callback queue.

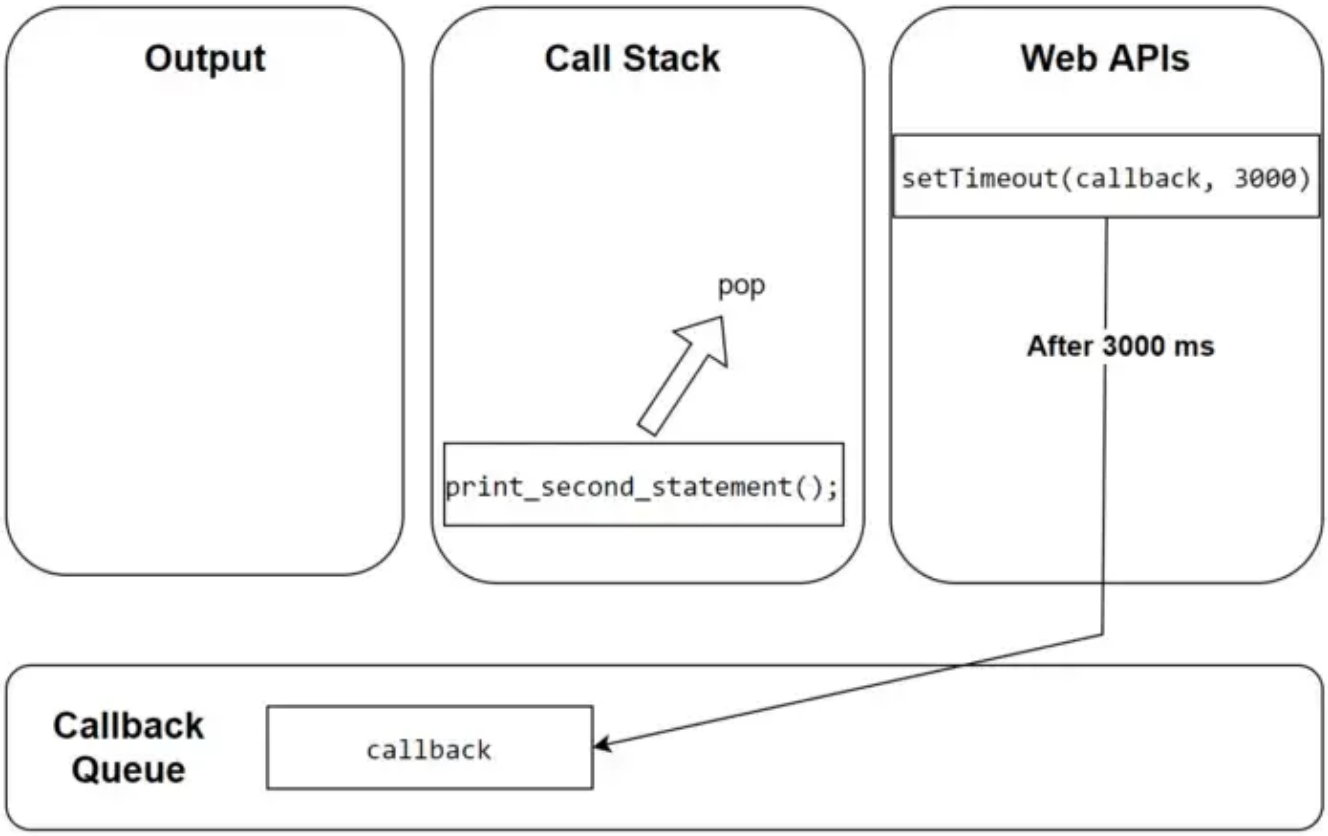
These are later pushed to the Call stack to be executed (when the event loop finds an empty call stack).

Lets understand how it;s working

```
function print_first_statement(){
    console.log("First statement");
};

function print_second_statement(){
    setTimeout(() => {
    console.log("Second statement");
}, 3000);
};

function print_third_statement(){
    console.log("Third statement");
};



//If I call the functions in sequence
//O/p will be

First Statement
Third Statement
Second Statement
```

Think about this how this happened if the Js is single threaded.
Ans is quite simple to understand, first the first context function get pushed directly into the stack and in a same time the event loop always checking if there is anything is in the queue to be pushed inside a stack if stack is empty(), but now nothing to be pushed inside of stack. Normally the js logged the ""First Statement"" on console. and the context get pop out from the stack.

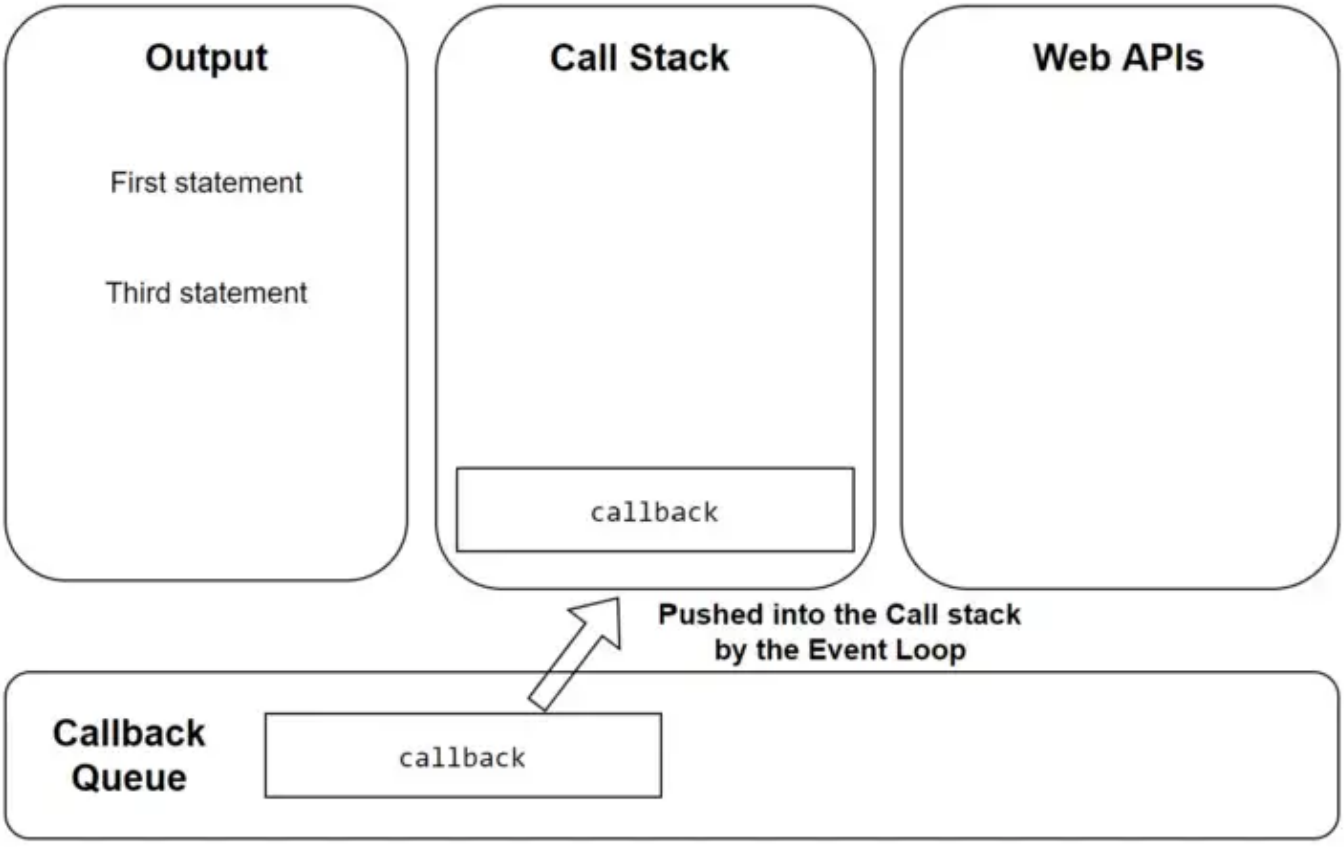
In meanTime js will check that the stack is empty and same as the language is the single threaded the js execute the second function context and push that inside of the stsck, but in second function there is a call to the web API setTimeOut( ). So Once WebAPI is done executing this (for 3000ms), it places the anonymous callback function in the Callback queue. and that context get pop out from the stack.



Now the second context get pop out from the stack and as js is move towards the next execution context so it will push the third context into the stack and logged the {"Third Statement"} before the Second Statement and same after logging on console the stack context get pop out.

Now event loop will check the stack status and it is empty so the event loop push that anonymous callback function inside the stack to get executed and after that the js will log value of Scond functional context which is "Second Statement"..
or
When the call stack becomes empty, the anonymous callback function in the callback queue will be

pushed into this empty call stack by the Event loop



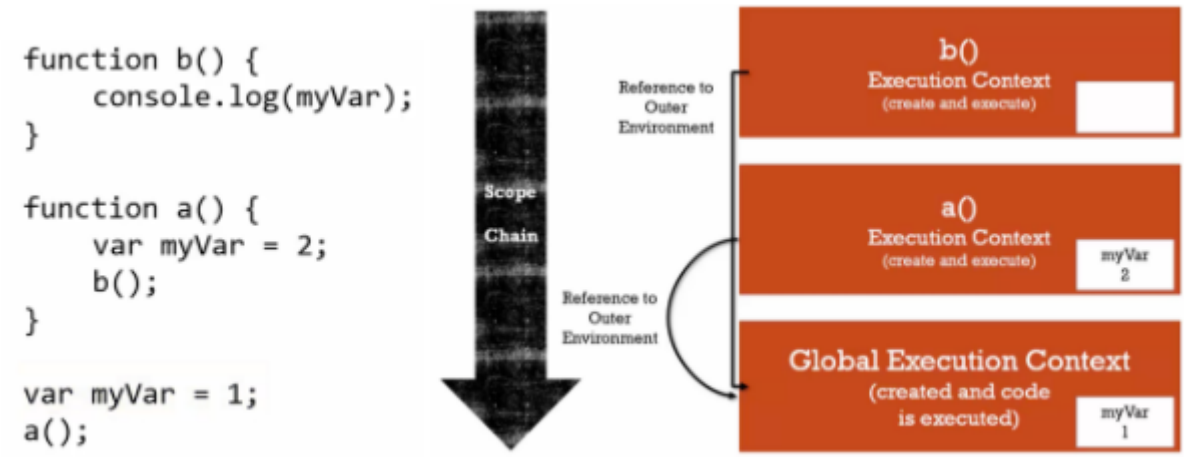
Execution Context in Details

Creation Phase
1.Variable Environment:
variable delclarations
functiona declarations
argument object stored here

2.Scope chain
3.This Keyword

This 3 things like var environment, scope chain, This keyword generated during the creation phase , right before the execution.

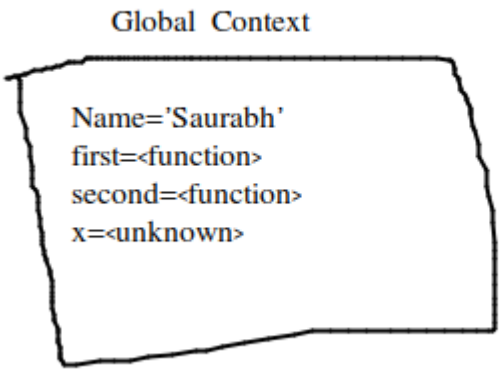
But the fact is the Arrow functions do not have this keywords as well as argument objects, they can get it from there parent functions.



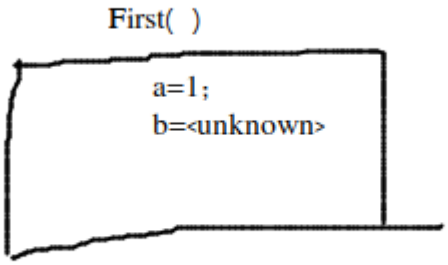
Or we can take a Example::

```
const name='Saurabh';
//arrow function
const first=()=>{
    let a=1;
    const b=second(4,4);
    a=a+b;
    return a;
}

function second(x,y){
    var c=2;
    return c;
}

const x=first();
```

Now How the global execution context will be looked like???



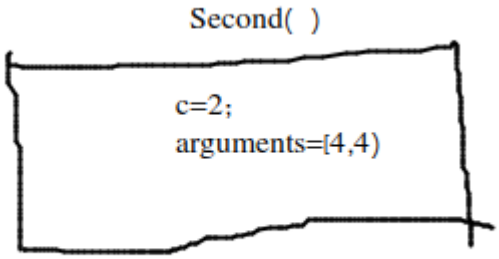
so in global context the variable name has been stored and for the function first and second only declaration get saved because function do have there own execution context. the x is unknown because first function is not executed yet, after that context will over (first function). it'll assign the value to x after that.



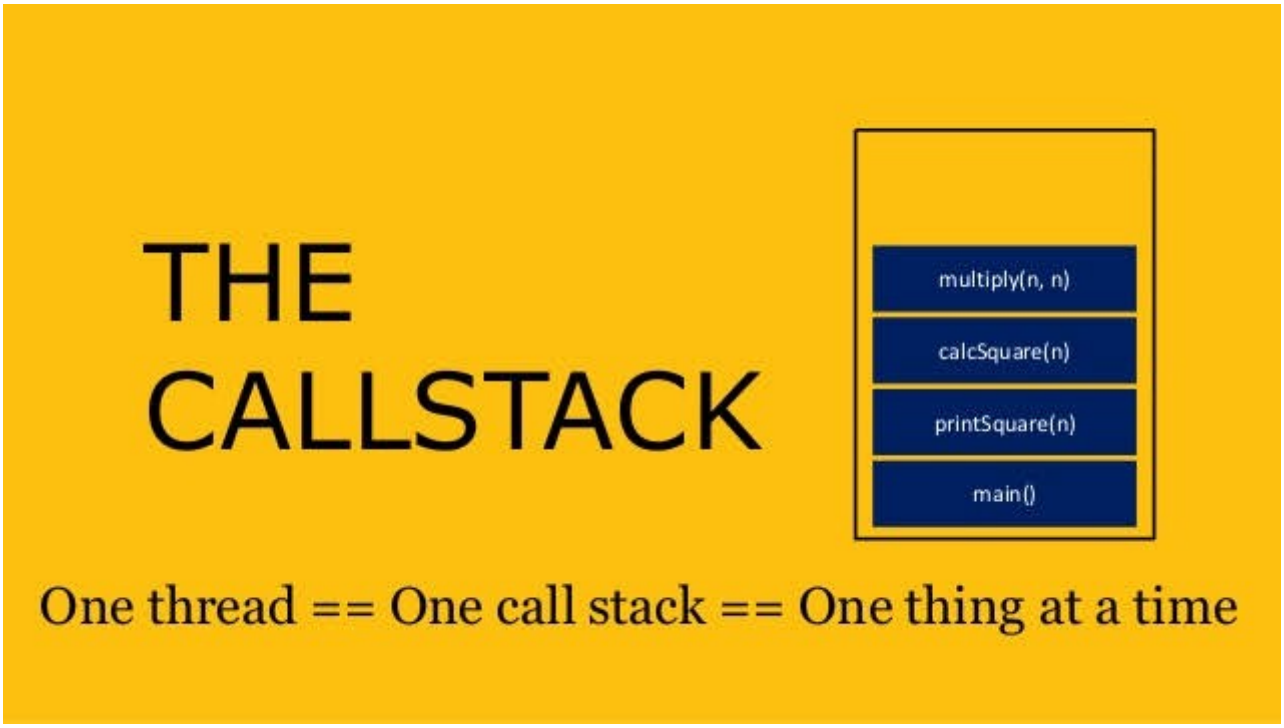
After the global execution context will pushed into the call stack, there is next function call for the first function and saparate execution context again pushed into the stack withh his own variable context and code. In the first context the value 1 has assigned to a. and the b is unknow because the value will assign to b once the second context get pop out from call stack and while returning back it will assign the value to b, till it will hold unknown.

As the First function is Arrow function so they do not have argument object and this keyword

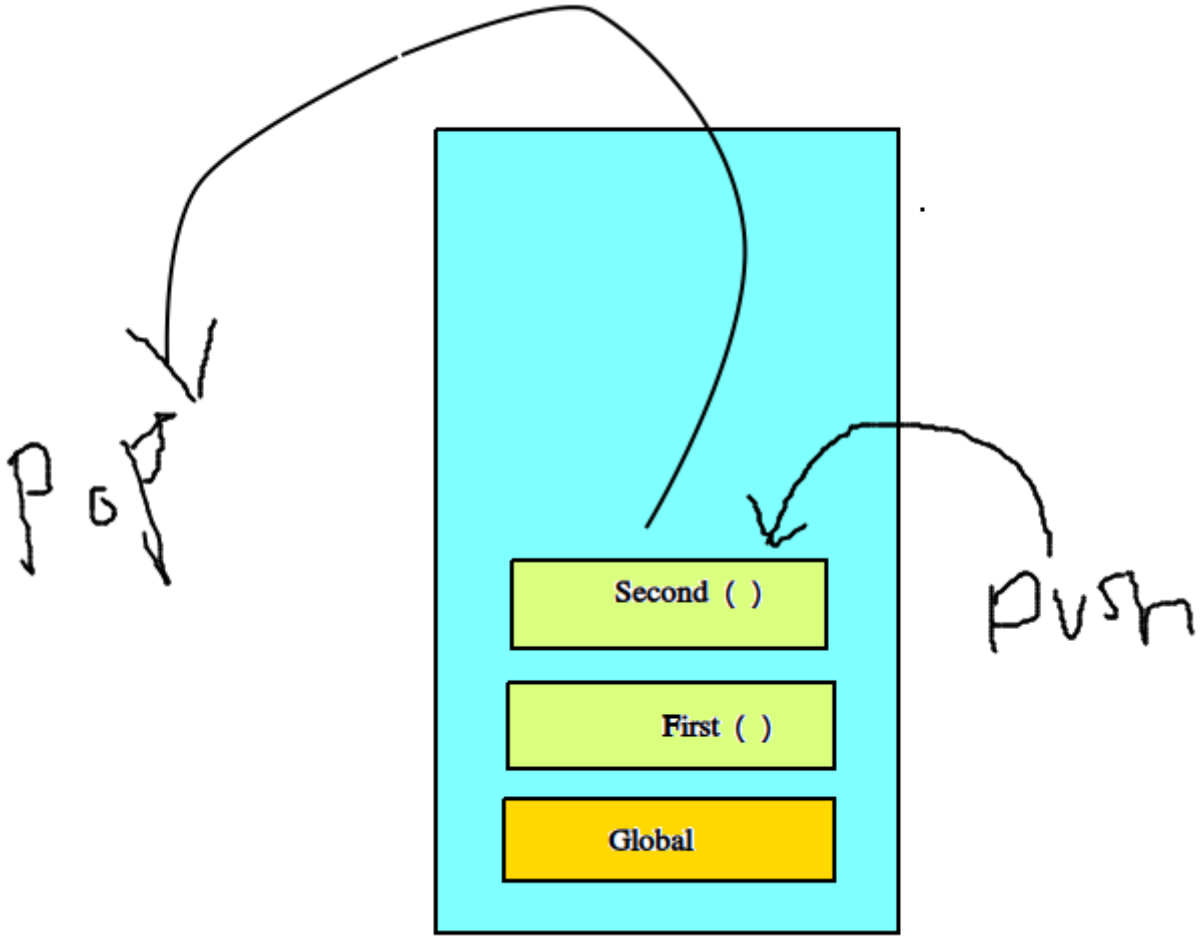
Check below that second function execution context with normal function do have argumnt object,



What actually is the callStack.



In context with the above example,
Call stack is a place where the execution context get stacked on top of each other, to keep the track of where we are in the execution.



Scope and Scope Chain --- -- -

In the context of programming, the scope chain is the chain of nested scopes (i.e., environments) that are searched for variable bindings. The scope chain is created at runtime, and it determines which variables are accessible to a given piece of code. For example, in a JavaScript program, the scope chain is used to resolve variable references. When a variable is accessed, the JavaScript interpreter searches through the scope chain to find the innermost variable binding for that variable, starting with the current scope and then moving outward to enclosing scopes until it finds the binding or reaches the end of the scope chain.

Why is it so important.??
in js we have lexical scoping, lexical scoping-> scoping is controlled by placement of functions and blocks in the code,
Lexical scoping, also known as static scoping, is a way of determining the accessibility of variables in a program. In lexical scoping, the accessibility of a variable is determined by its location in the source code. This means that a variable is only accessible within the code block in which it is defined, as well as any inner code blocks.

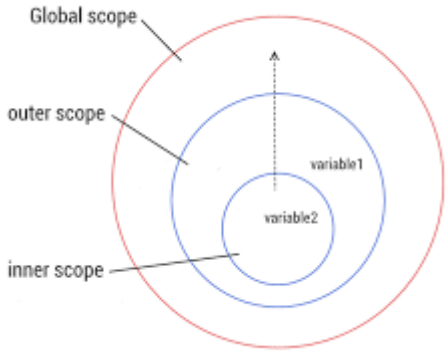
Indirectly inner function have access to the variables that are defined inside the outer or parent function.

Scopes:

1. Global
2. function
3. block

Functions are block scoped but only in the strict Mode. If strict mode has been revoked then they are in there functional scope;

Scope of variable is the region of our code where the certain variable can be accessed.

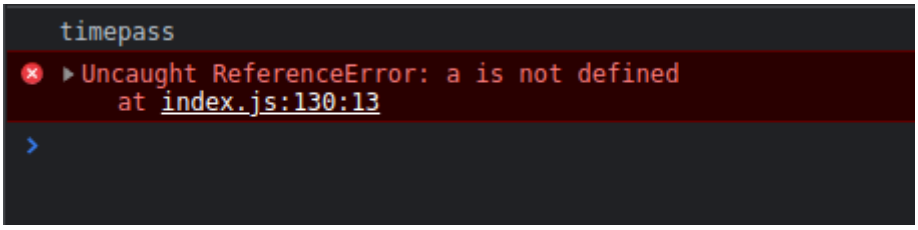



```
const arrowf=()=>{
    let a='timepass';
    let b='fun';
    const v=()=>{
        console.log(a);
    }

    v();
}

arrowf();
```

as the variable a is available inside the arrowf and it can be accessable to its inner function v.
but ouside of the arrowf the varible a and b is not available.
let's check the o/p;

```
arrowf();
```

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



further execution has been stoped because of referance error has occur. (Mighr be one can say that thread is blocked--but not sure).
This method of checking the variable in the scopes called variable Lookup. It cannot be look down only look up in scopes.

only let and const are block scoped. var is not

```
const arrow=()=>{

if(age>=19){
    var m=true;
    let decade=5;

}
    const arr=()={
        //variable defination

    }

}
```

here in the above code the var m is not the block scope but it;s more on a functional scope because var do not support the block scope so in arrow function var can be accessable anywhere.

Hoisting in JS

Hoisting: makes some type of variables accessible in code befire they actually declared.
before execution code is scanned for the variable declarations and for each variable a new property is created in the variable environment object.

In JavaScript, hoisting refers to the behavior of moving declarations to the top of a code block. This means that variables and functions can be used before they are declared.

For example, the following code will still work even though the hello function is called before it is declared:

```
//check the function declaration is hoisted

hello();

function hello() {
  console.log("Hello, World!");
}
```

This is because in JavaScript, declarations are processed before code execution. This means that declarations are "hoisted" to the top of the code block, so the hello() function can be called before it is declared.

However, it's important to note that only declarations are hoisted, not assignments. This means that the following code will not work, because the x variable is assigned a value before it is declared:
->

```
console.log(x);  // This will throw an error
//use let and const not var , var is actually hoisted. as let and const also hoisted but they are in Temporal dead zone

var x = 5;
```

In this case, the x variable is not hoisted to the top of the code block, so trying to access its value before it is declared will result in an error. (Not actually in case of var)

Overall, hoisting can be a useful behavior in JavaScript, but it can also lead to unexpected results if not used carefully. It's generally a good idea to declare variables and functions at the top of your code to avoid any confusion.

Why TDZ???
(temporal dead zone) as the accessing the variable before declaration is a bad practice to cope up with that js introduces TDZ.

Why Hoisting exist if it is problamatic. ???

in some cases or in some programming techniques it is important to access the function before declaration. (Mutual-- Recursion).

Examples of TDZ and Hoisting::::

```
console.log(me);
console.log(me1);
```

```
console.log(me2);

var me='sdasd';
let me1='poe';
const ='Loper';

Undefined

Referance Error.----here execution terminated
```

```
console.log(me(2,4));

console.log(me1(2,4));
console.log(me2(2,4));

function me(a,b){
return a+b;
}

const me1 = function(a,b){
return a+b;
}

const me2=(a,b)=>{
return a+b;
}

//only function declaration get hosited here
```

this Keywords

When the global execution context pushed into the stack the this keyword will be assign to the window object first,
This is a special varible that is created for every execution context (Every Functtion). Takes the value of (Points to) the "Owner" of the function in which the this keyword is used.

It' points to the owner of that function!!! They value of this keyword is not static... it depends on how the function is called and its value is only assigned when the function is actually called.

function attached to an Object.
Lets see a Example

```
const jo={
    name:'Jo',
```

```
    year:1299,
    calag:function(){
            return 1999-this.year;
    }
};

jo.calag();

//in this,this is pointing to the object as calag is method and the value of this is jo
in this case we can access prop of jo using this.
```

Now here one point to be understood that the this keyword in strict mode will point to undefiined first. Other case it'll point to global window object,

In JavaScript, the keyword this refers to the object that is currently being referred to or used. It is commonly used in object-oriented programming to refer to the current object, or to the object that an event is currently being called on.
That's not enought i think!!!

This's values is wholly depends on how that function has called..

LEts analyze the ways!!!!

Method-->> This== It points to the object that is calling the method.
Ex:

```
const jo={
    name:'Jo',
    year:1299,
    calag:function(){
            return 1999-this.year;
    }
};

jo.calag();
```

In simple funtion call :: this== is undefined first (In strict mode);

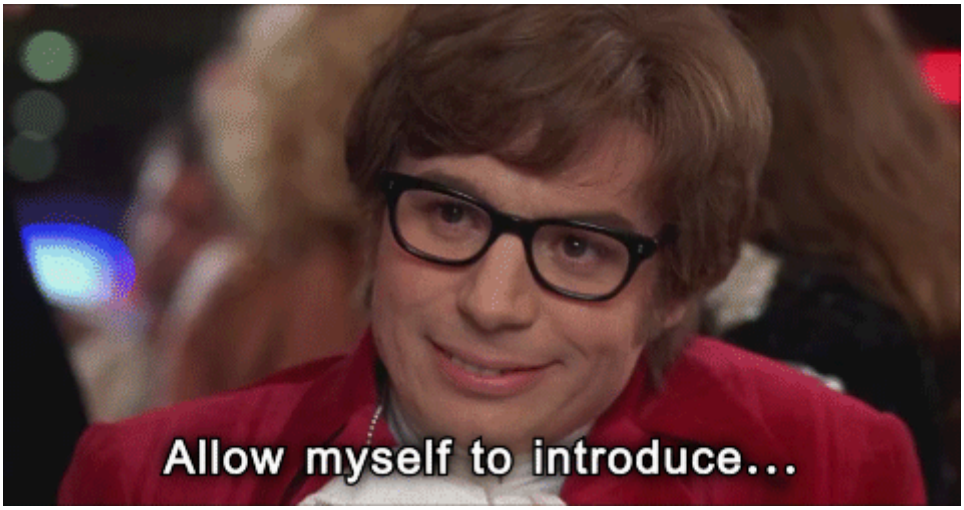
In Arrow function :: this== here this pointing to (as arrow do not have there own this keyword) the parent functions in simple it called as Lexical This. <this of surrounding functions (lexical this)>

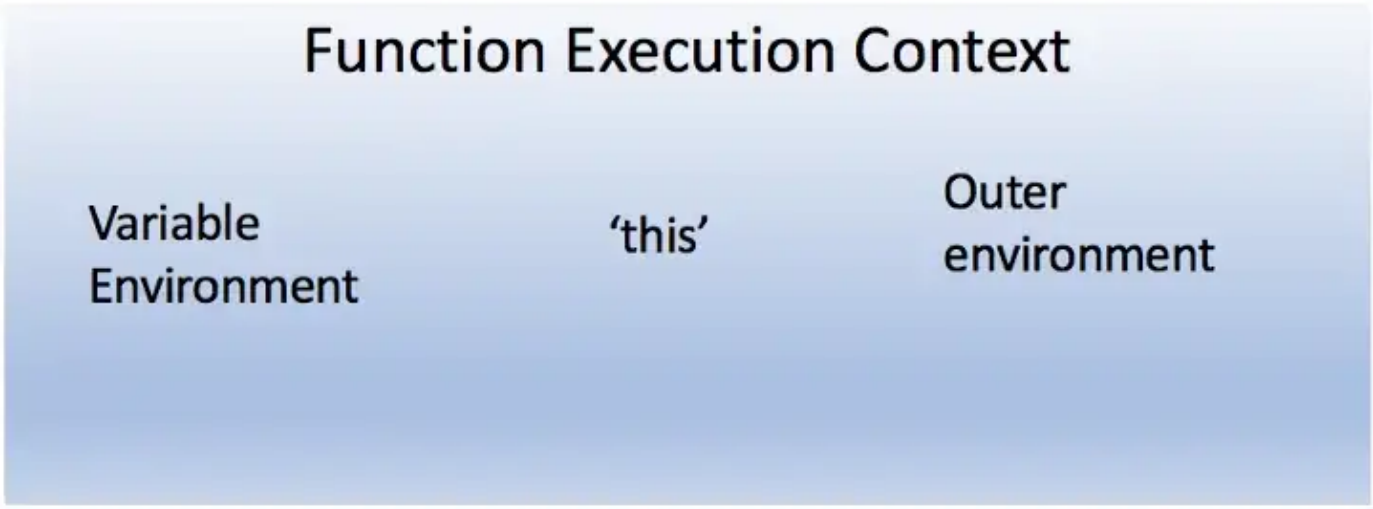
In Event Listner:: this== here this will points t o the DOM element that the Handler is attached to.

[IMP]
this will not points to the function itself or in function which we are using it. this will not point to variable environent also.

we'll goint to discuss the call, apply, bind , new methods in later sections.

This is like actual this:::
Allow my self to introduce myself/.....





[Article On this KeyWord]
-->Covered most of the aspect of this keywrd in js:

The this keyword in JavaScript confuses new and seasoned JavaScript developers alike. This article aims to elucidate this in its entirety. By the time we make it through this article, this will be one part of JavaScript we never have to worry about again. We will understand how to use this correctly in every scenario, including the ticklish situations where it usually proves most elusive.

[sc:mongodb-book]
We use this similar to the way we use pronouns in natural languages like English and French. We write,

“John is running fast because he is trying to catch the train.”

Note the use of the pronoun “he.” We could have written this: “John is running fast because John is trying to catch the train.” We don’t reuse “John” in this manner, for if we do, our family, friends, and colleagues would abandon us. Yes, they would. Well, maybe not your family, but those of us with fair-weather friends and colleagues. In a similar graceful manner, in JavaScript, we use the this keyword as a shortcut, a referent; it refers to an object; that is, the subject in context, or the subject of the executing code. Consider this example:

```
  var person = {
    firstName: "Penelope",
    lastName: "Barrymore",
    fullName: function () {
        // Notice we use "this" just as we used "he" in the example sentence
earlier?:
        console.log(this.firstName + " " + this.lastName);
    // We could have also written this:
        console.log(person.firstName + " " + person.lastName);
    }
}
```

If we use person.firstName and person.lastName, as in the last example, our code becomes ambiguous. Consider that there could be another global variable (that we might or might not be aware of) with the name “person.” Then, references to person.firstName could attempt to access the firstName property from the person global variable, and this could lead to difficult-to-debug errors. So we use the “this” keyword not only for aesthetics (i.e., as a referent), but also for precision; its use actually makes our code more unambiguous, just as the pronoun “he” made our sentence more clear. It tells us that we are referring to the specific John at the beginning of the sentence.
Just like the pronoun “he” is used to refer to the antecedent (antecedent is the noun that a pronoun refers to), the this keyword is similarly used to refer to an object that the function (where this is used) is bound to. The this keyword not only refers to the object but it also contains the value of the object. Just like the pronoun, this can be thought of as a shortcut (or a reasonably unambiguous substitute) to refer back to the object in context (the “antecedent object”). We will learn more about context later.

Basics of this in js
First, know that all functions in JavaScript have properties, just as objects have properties. And when a function executes, it gets the this property—a variable with the value of the object that invokes the function where this is used.

The this reference ALWAYS refers to (and holds the value of) an object—a singular object—and it is usually used inside a function or a method, although it can be used outside a function in the global scope. Note that when we use strict mode, this holds the value of undefined in global functions and in anonymous functions that are not bound to any object.

The Biggest Gotcha with JavaScript “this” keyword

If you understand this one principle of JavaScript’s this, you will understand the “this” keyword with clarity: this is not assigned a value until an object invokes the function where this is defined. Let’s call the function where this is defined the “this Function.”

Even though it appears this refers to the object where it is defined, it is not until an object invokes the this Function that this is actually assigned a value. And the value it is assigned is based exclusively on the object that invokes the this Function. this has the value of the invoking object in most circumstances. However, there are a few scenarios where this does not have the value of the invoking object. I touch on those scenarios later.

The use of this in the global scope

In the global scope, when the code is executing in the browser, all global variables and functions are defined on the window object. Therefore, when we use this in a global function, it refers to (and has the value of) the global window object (not in strict mode though, as noted earlier) that is the main container of the entire JavaScript application or web page.

Thus:

```
var firstName = "Peter",
    lastName = "Ally";

    function showFullName () {
    // "this" inside this function will have the value of the window object
    // because the showFullName () function is defined in the global scope,
just like the firstName and lastName
    console.log (this.firstName + " " + this.lastName);
    }

    var person = {
    firstName   :"Penelope",
    lastName    :"Barrymore",
    showFullName:function () {
    // "this" on the line below refers to the person object, because the
showFullName function will be invoked by person object.
    console.log (this.firstName + " " + this.lastName);
    }
    }

    showFullName (); // Peter Ally

    // window is the object that all global variables and functions are
defined on, hence:
    window.showFullName (); // Peter Ally
```

```
    // "this" inside the showFullName () method that is defined inside the
person object still refers to the person object, hence:
    person.showFullName (); // Penelope Barrymore
```

When this is most misunderstood and becomes tricky

The this keyword is most misunderstood when we borrow a method that uses this, when we assign a method that uses this to a variable, when a function that uses this is passed as a callback function, and when this is used inside a closure—an inner function. We will look at each scenario and the solutions for maintaining the proper value of this in each example.

Fix this when used in a method passed as a callback

```
   // We have a simple object with a clickHandler method that we want to use
when a button on the page is clicked
   var user = {
   data:[
   {name:"T. Woods", age:37},
   {name:"P. Mickelson", age:43}
   ],
   clickHandler:function (event) {
   var randomNum = ((Math.random () * 2 | 0) + 1) - 1; // random number
between 0 and 1

   // This line is printing a random person's name and age from the data
array
   console.log (this.data[randomNum].name + " " + this.data[randomNum].age);
   }
   }

   // The button is wrapped inside a jQuery $ wrapper, so it is now a jQuery
object
   // And the output will be undefined because there is no data property on
the button object
   $ ("button").click (user.clickHandler); // Cannot read property '0' of
undefined
```

In the code above, since the button ($(“button”)) is an object on its own, and we are passing the user.clickHandler method to its click() method as a callback, we know that this inside our user.clickHandler method will no longer refer to the user object. this will now refer to the object where the user.clickHandler method is executed because this is defined inside the user.clickHandler method.

And the object that is invoking user.clickHandler is the button object—user.clickHandler will be executed inside the button object's click method.

Note that even though we are calling the clickHandler () method with user.clickHandler (which we have to do, since clickHandler is a method defined on user), the clickHandler () method itself will be executed with the button object as the context to which "this" now refers. So this now refers to is the button ($("button")) object.

At this point, it should be apparent that when the context changes—when we execute a method on some other object than where the object was originally defined, the this keyword no longer refers to the original object where "this" was originally defined, but it now refers to the object that invokes the method where this was defined.

Solution to fix this when a method is passed as a callback function:
Since we really want this.data to refer to the data property on the user object, we can use the Bind (), Apply (), or Call () method to specifically set the value of this.

I have written an exhaustive article, JavaScript's Apply, Call, and Bind Methods are Essential for JavaScript Professionals, on these methods, including how to use them to set the this value in various misunderstood scenarios. Rather than re-post all the details here, I recommend you read that entire article, which I consider a must read for JavaScript Professionals.

To fix this problem in the preceding example, we can use the bind method thus:

Instead of this line:

$ ("button").click (user.clickHandler);
We have to bind the clickHandler method to the user object like this:

```
$("button").click (user.clickHandler.bind (user)); // P. Mickelson 43
```

Fix this inside closure
Another instance when this is misunderstood is when we use an inner method (a closure). It is important to take note that closures cannot access the outer function's this variable by using the this keyword because the this variable is accessible only by the function itself, not by inner functions. For example:

```
var user = {
  tournament:"The Masters",
  data      :[
  {name:"T. Woods", age:37},
  {name:"P. Mickelson", age:43}
  ],

  clickHandler:function () {
```

```
    // the use of this.data here is fine, because "this" refers to the user
object, and data is a property on the user object.

    this.data.forEach (function (person) {
    // But here inside the anonymous function (that we pass to the forEach
method), "this" no longer refers to the user object.
    // This inner function cannot access the outer function's "this"

    console.log ("What is This referring to? " + this); //[object Window]

    console.log (person.name + " is playing at " + this.tournament);
    // T. Woods is playing at undefined
    // P. Mickelson is playing at undefined
    })
    }

    }

    user.clickHandler(); // What is "this" referring to? [object Window]
```

this inside the anonymous function cannot access the outer function's this, so it is bound to the global window object, when strict mode is not being used.

Solution to maintain this inside anonymous functions:

To fix the problem with using this inside the anonymous function passed to the forEach method, we use a common practice in JavaScript and set the this value to another variable before we enter the forEach method:

```
  var user = {
    tournament:"The Masters",
    data      :[
    {name:"T. Woods", age:37},
    {name:"P. Mickelson", age:43}
    ],

    clickHandler:function (event) {
    // To capture the value of "this" when it refers to the user object, we
have to set it to another variable here:
    // We set the value of "this" to theUserObj variable, so we can use it
later
    var theUserObj = this;
    this.data.forEach (function (person) {
```

```
    // Instead of using this.tournament, we now use theUserObj.tournament
    console.log (person.name + " is playing at " + theUserObj.tournament);
    })
    }

    }

    user.clickHandler();
    // T. Woods is playing at The Masters
    //  P. Mickelson is playing at The Masters
```

It is worth noting that many JavaScript developers like to name a variable “that,” as seen below, to set the value of this. The use of the word “that” is very awkward for me, so I try to name the variable a noun that describes which object “this” is referring to, hence my use of var theUserObj = this in the preceding code.

Fix this when method is assigned to a variable

The this value escapes our imagination and is bound to another object, if we assign a method that uses this to a variable. Let’s see how:

```
// This data variable is a global variable
    var data = [
    {name:"Samantha", age:12},
    {name:"Alexis", age:14}
    ];

    var user = {
    // this data variable is a property on the user object
    data    :[
    {name:"T. Woods", age:37},
    {name:"P. Mickelson", age:43}
    ],
    showData:function (event) {
    var randomNum = ((Math.random () * 2 | 0) + 1) - 1; // random number
between 0 and 1

    // This line is adding a random person from the data array to the text
field
    console.log (this.data[randomNum].name + " " +
this.data[randomNum].age);
    }
```

```
    }

    // Assign the user.showData to a variable
    var showUserData = user.showData;

    // When we execute the showUserData function, the values printed to the
console are from the global data array, not from the data array in the user
object
    //
    showUserData (); // Samantha 12 (from the global data array)
```

Solution for maintaining this when method is assigned to a variable:
We can fix this problem by specifically setting the this value with the bind method:

```
    // Bind the showData method to the user object
    var showUserData = user.showData.bind (user);

    // Now we get the value from the user object, because the this keyword
is bound to the user object
    showUserData (); // P. Mickelson 43
```

[END]

Check out more on this Keyword refer the book, this and Object written by kyle Simpson for deep understanding.

practical Examples with this

```


console.log(this);
//this will points to global window obj

const calc=function(byear){
    console.log(12);
    console.log(this);
    //here the this value assigned to undefined
};
calc(19);
```

```
//In case of arrow function

const calage = ()=>{
    console.log(this);
    //here this is the window obj that is parent of this calage function
```

```
    //here it uses lexical this where is in it's parent scope

};
calage();
```

```
const Jonas={
    year:1991,
    calage:function(){
        console.log(this);
        //here this is referance to jonas obj because jonas called this
calage funcion
    }
};
Jonas.calage();


//try with global values
const data=[
    {

    p1:'name',
    p2:'age'
    }
];


const user={
    data:[{
        p1:'name1',
        p2:'age2'
    }],
    Poke:data.forEach( function(element){
        //here it;s problematic to get to know this keyword check it by own
        // console.log(this.data.p1);
        // console.log(this.data.p2);


    })
}
```

```
//Borrowing

const mat={
    year:2019,
```

```
};

mat.calage=Jonas.calage;

//here we have simply copy the method of jonas to mat that is called Method
Borrowing

mat.calage();

//now for this , this has been attached to the  mat object
//{year: 2019, calage: f}
//Now here the method is still inside the Jonas object but still it has been
pointing to the mat obj because mat has called the method

const f=jonas.calage;
//copying the function in variable f , just copy do not call using ()

f();

//here this is undefined. calage: f}
//index.js:190 Uncaught ReferenceError: jonas is not defined

//Here now jonas is no more owner of the this keyword, but that is not
correct way to say this as the context has changed only
```

[Note]

Normal function do have argument object where u can pass the number of parameter and as args they are saved into the array of arguments.

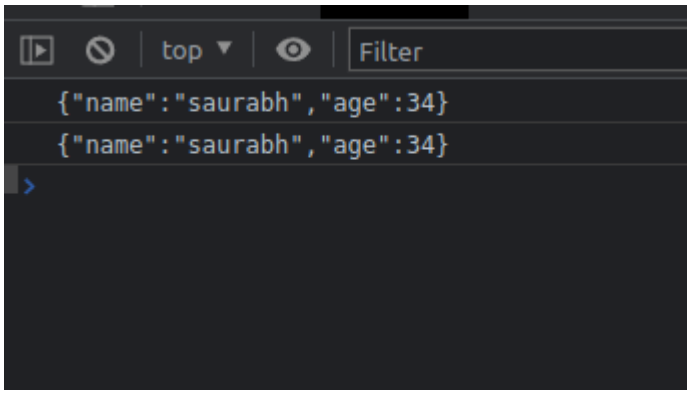
But , they arrow function do not behave the same they do not have args object or array.

In modern Js we aint use argument array that much.

Primitive Vs Objects (Premitive vs Object Types)

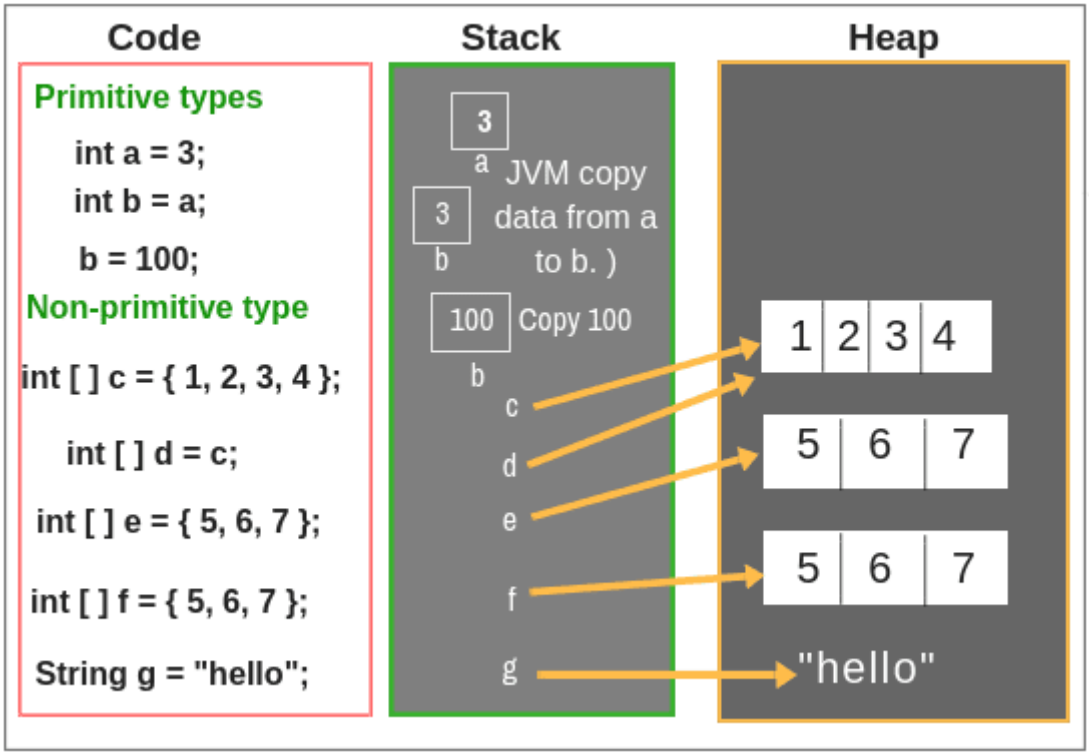
```
const myobj={
    name:'saurabh',
    age:21
};

const someonesObj=myobj;
//Might be there referance has copied not the values now both have same
referance
```

```
//lets check
someonesObj.age=34;

console.log(JSON.stringify(myobj));
console.log(JSON.stringify(someonesObj));
```

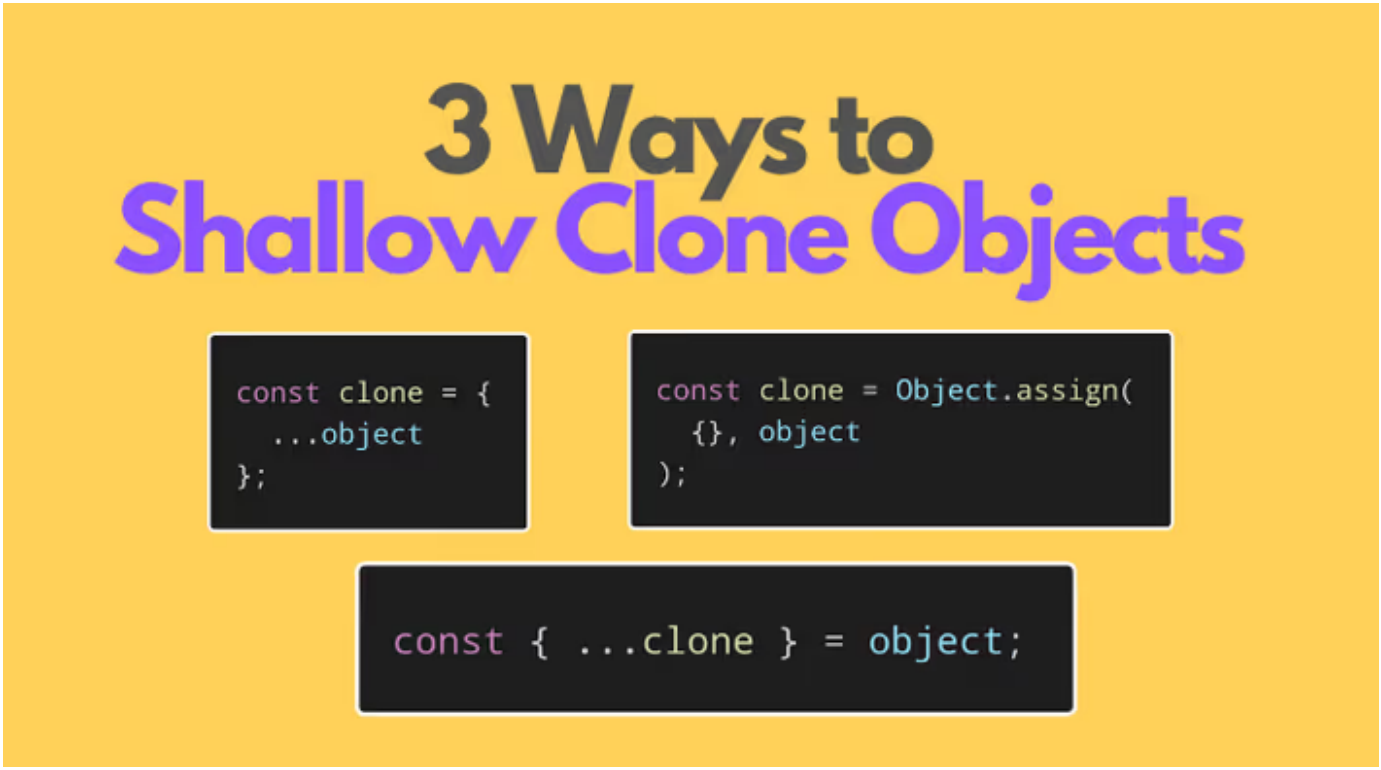


here the value of both obj have changed.
Remember that the Primitives are primitive type while Objects are referance type

lets say that the objects are store in heap and there referance is stored inside of stack while the primitives are directly stored inside of stack. understand with the help of below image.



In C and c++ we are fucked up if we aint release the memory but in these languages the garbage collector do his own work perfectly.



we are goin to see the sprade operator in upcoming sections.

Simple data structures -> Array, set Maps,strings

Array destructuring:

```
const arr=[1,2,3];
const [a,b,v]=arr;
console.log(a,b,v);

//Try with objects but they aint iterative so it'll throw the error but
array in object will work fine

//example

const res={
    name:'saurabh',
    category:['a','b'];
}
const[name,category]=res;
//this aint gonna work, maybe!!!

const[main,prod]=res.category;
//this destructure the categories in new varriables like main and prod

[main,prod]=[prod,main];
//this is swapping the values and it looks pretty  simple
```

```
const nested=[1,2,[5,6]];
//here we can skip 2 while destrrcturing
const[i, ,j]=nested;

//or destructure more

const[i, ,[j,k]]=nested;

const[i=1,k=1,j=1]=nested;
//this setting default valuues will help us.might this help us when getting values from api.
```

Destrcturing Objects

```
const res={
name:'saurabh',
cat:'poker',
cash:'0901'
};

const{name,cat,cash}=res;

console.log(name,cat,cash);
//as the name, cat and cash are the properties so naming must be matched.
//now we can change the naming vars

const{name:ResName, cat:category,cash:money}=res;

//this is how one can modify the names, Object destructuring also we used in many application.

//giving  default values
//-->
const {menu=[] ,starterMenu: starters=[]}=res;
//assume that we have startermenu and menu as a our props of res object;
```

```
const resturent={

//we can destructure the passed object as a parameter inside a function argumen
```

```
orderdelivery:function({time,name,address}){

        console.log(time);

    }
};

resturent.orderdelivery({
    name:'daur',
    time:'22:10',
    address:'LA'
});
```

The Sprade Operator

```
const arr=[7,8,9];
//suppose we have to add some of the elements in front of this array;
//In es6 we can do it using the sprade operator

const newarr=[1,2,3,...arr];
//O/p-> [1,2,3,7,8,9];

//what if we havedone like

const newarr=[1,2,3,arr];
//this will print -> [1,2,3.[7,8,9]];
```

Short Circuiting || and && operators

```
//short circuiting -- Use any data type and return any data tyope

console.log(3 || 'Jonas'); -->3

console.log('' || 'Jonas'); -->Jonas

console.log(true || 0); -->true

console.log(undefined || null); -->null
```

```
//as always it goin to print the truthy value, in case of 3 and jonas the immediate check is 3 and then
//it pass that check and print the 3 on console. same for empty strinng ans Jonas the empty string is Falsy value
//so it goin to print the Jonas, In case of undefined and null it will print null as undefined  is falsy values. although the null also a falsy value.
```

**In case of && the flow is , it will check every truthy value if all vaues are truthy then it'll print last one for example
3 && 'Jonas ' here 3 is truthy value and jonas also a truthy value so when we are using && the it"ll logg Jonas on console. **

Nllish Coalescing Operator (??)

```
const number=0;
const g=number || 10; -->10

//but if we used Nullish coalescing operator
//it'll print 0

const g1=number ?? 10; --> 0
```

For Of Loop

```
const menu=['asda','poker','poe,'];

for(const item of menu)console.log(item);

//we can use break and continue in this loops also

//For of Loop

const menu=['asda','poker','poe,'];

for(const item of menu)console.log(item);

console.log(menu.entries());
//this is array of iterators or it's a array iterator

for(const i of menu.entries()){
    console.log(i);
```

```
}
```

Optional Chaining

```
const hotel={
    openhours:
{
    thus:{
        open:'2',
        close:'5'
    },
    fri:{
        open:'1',
        close:'12'
    }

}

};

console.log(hotel.openhours.fri.close);


//suppose here we have access for monday but monday dont exist in obj
//think about a real time scenario where u dont know the hotel timing or
data is requested throught api

//we can add if lopp but what if there are alot properties

//In optional chaining if any properties not defined in obj and user is
accessing that prop
//it will throow the undefined insted of error

console.log(hotel.openhours.mon?.open);
//if it exist then print open hr and if not it log undefine on console.
```

Traversing through objects

```
for(const day of Object.keys(hotel.openhours)){
    console.log(day);

}
```

```
for(const Hrs of Object.values(hotel.openhours)){
    console.log(Hrs);

}
//destructuring

for(const [day,{open,close}] of Object.entries(hotel.openhours)){
    console.log(day,open,close);
}
```

Sets and Maps and strings

```
const orderset=new Set(['pasta','Pizza','burshetta']);
//set can hold mixed data type in javascript and duplicated values not
allowed
//set are also iterable and order is relavent

console.log(new Set('jonas'));
//as o/p is {'j','o'...'s'}
//so remember that the strings also iterable

console.log(new Set());
//this is empty set
//to check size

console.log(orderset.size);

//to check the value has in the set

console.log(orderset.has('Pizza'));
//it'll return true if the value is in set

orderset.add('gbread');
orderset.add('gbread');

//the value get added once only because it is set only
///to delete
orderset.delete('Pizza');

console.log(orderset);

//there are no index in set so iterator may help
```

now remember set is not implemented to replace arrays , many of the important needed operations we can able to do on arays we cant do that on sets.

Maps

```
const rest=new Map();
//creating a empty map

rest.set('name','Classico Italiano');
//set is pretty similar to add and push

rest.set(1,'LA');
rest.set(2,'Portugal');

//set method returns a map lets print it

console.log(rest.set('espano','Porg'));
//this will return Map(4) {'name' => 'Classico Italiano', 1 => 'LA', 2 =>
'Portugal', 'espano' => 'Porg'}

//so if set method returning a map so we can have a chain of set method

rest.set('categories',
['pixxa','mango','joiekk']).set('open',11).set('close',23).set(true,'We are
Open').set(false,'we are close');

console.log(rest);

rest.get(1);
//will print LA

//Boolean keys

const time=21;
const val=rest.get(time > rest.get('open')&& time< rest.get('close'));
console.log(val); //it will print we are open

//delete the keys
rest.delete(2);

//size
rest.size;
//size is s property not a function
```

Iterate over a map

```
//maps are iterable

for(const [key,value] of quest){
    console.log(key,'---->',value);

}
```

There are essentially 3 source of data::

1.from a program itself
2.from the UI.
3.From the external sources : data fetched from web API (Like weather api).

-->collection of this data need data strutctures so need to store them somewhere.
if simple data--> then array or sets

if key value pair(JSON format)--> Maps and Objects

Other Build in ds:
WeakMap
WeakSet

Not Build in ds:
LL,
Queue,
stack

I suggest to refer the documentation for different functions assaociated with string maps and sets and array, dot waste time to remember those.

More on Functions

Default Parameters:

```
bookings=[];
const createBooking=function(Fname,numberpass=1,price=199){

    const booking={
        Fname,
        numberpass,
        price
    }
    console.log(booking);
```

```
    bookings.push(booking);
}
createBooking('LHterminal');

//if we have not set or pass the values of all parameters they defaultyly
set to undefined
//so we have used the shortcrcuiting method to set those default,but es6
syntax is much simple
//we can define the default values/ expression in function args itself as we
have done in above code
```

Function accepting callback function

```
const oneword=function(str){
    return str.replace(/ /g,'').toLowerCase();
    // / /g is regex checkout the regex in javascript

}
const upperFirstword=function(str){
    const[first,...others]=str.split(' ');
    return [first.toUpperCase(), ...others].join(' ');


};

const trasformer=function(str,fn){
    console.log(`Orignal string:: ${str}`);
    console.log(`Transformed string:: ${fn(str)}`);

    console.log(`Transformed by:: ${fn.name}`);
    //here fn.name is actually printing the name of that function


}
trasformer('javascript is the best',oneword);
trasformer('javascript is the best',upperFirstword);
```

check the trasformer function accet the function as a argument that is why we called them a higher order function.

callback function allow us to create abstraction...
we have hide the detail of some functional defination from the user.

functions returning functions

```
const greet=function(greeting){
    return function(name){
        console.log(`${greeting} ${name}`);
    }
}

const greeterHey=greet('Hey');
console.log(greeterHey('Jonas'));

//or

greet('helo')('saurabh');
//this will also goin to work


//using arrow functions

const greetaarr=gretting=>name=>console.log(`${gretting} ${name}`);

greetaarr('Hi')('Jonas');

//This is confusing but it is a simple way to write this all stuff
```

call and apply

```
const luftansa={
    airline:'Luft',
    icode:'LH',
    bookings:[],
    book(flightnum,name){
        //this syntax introduced in es6
        console.log(`${name} booked on seat on ${this.airline} flight
${this.icode} ${flightnum}`);
        this.bookings.push({flight: `${this.icode} ${flightnum}`,name});
        console.log(this.bookings);
```

```
    },

}

luftansa.book(234,'Jonas');

const eurowings={
    name:'Er',
    icode:'Er',
    bookings:[],



};
```

```
//const book=luftansa.book;
//now it is a simple regular function call so this is now goin to point to
undefined check out the behaviour of this keyeword

//book(23,'siels');
//to fix this problem...we need to tell js what this keyword explicitily
pointingg to
//call apply bind will gonna help



const book=luftansa.book;
book.call(eurowings,23,'sara willams');
console.log(eurowings);  //-->{name: 'Er', icode: 'Er', bookings: Array(1)}

//call method 1st argument is explicitly setting the this keyword pointer to
eurowing obj

book.call(luftansa,212,'cooper');
console.log(luftansa.bookings);  //1{flight: 'LH 212', name: 'cooper'}

//apply method does the same but the apply method takes array of argument as
second arg

const flightdata=[432,'Merie coper'];

book.apply(eurowings,flightdata);
console.log(eurowings);
```

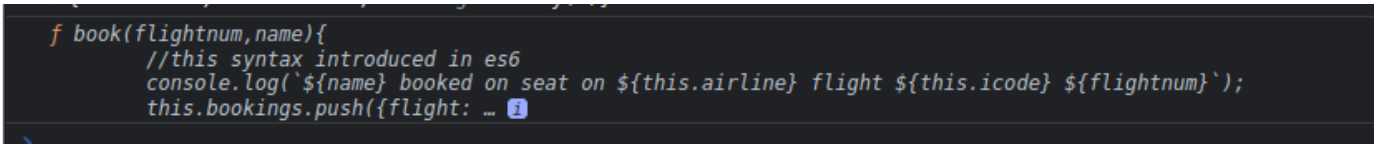
```
//thats how both keywords are working


//in modern js we no longer use apply method so same we can do using call

book.call(eurowings,...flightdata);
```

***Bind method**

Bind is same as call but it returna a brand new function copy of what we are gonna refer to--> for examle

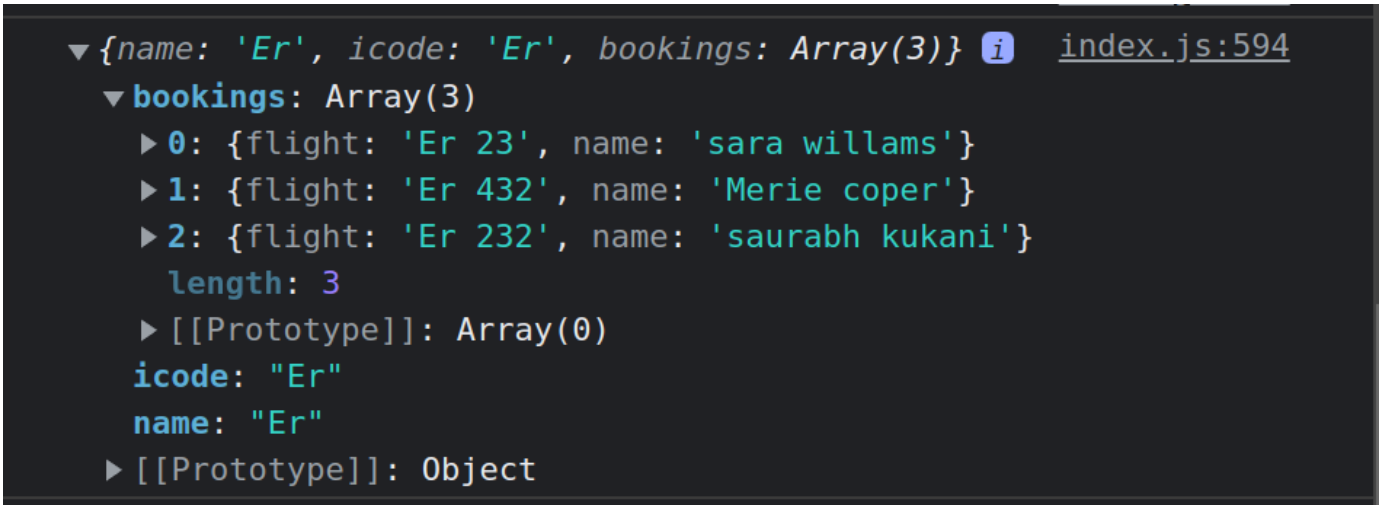
(refer the above code):
console.log(book.bind(eurowings));

//This statement gonna return a function!!!



we can use this function now to pass the values!!

```
console.log(book.bind(eurowings)(232,'saurabh kukani'));
console.log(eurowings);
```



the argument has been passed to the new function that is refering o the eurowing object.

```
const addTax=(rate,value)=>value+value*rate;

console.log(addTax(0.1,200));

const addVAT=addTax.bind(null,0.23);
```

```
//the first argumenet of bind is this, second argument we have mentioned as a preset value for rate that is goint to be 0.23 now onwards

console.log(addVAT(400));

//using the callbacks we can do the same without using bind

const addTax1=(rate)=>{
    return addVAT1=(value)=>{
        return value+value*rate;

    }
};


const add=addTax1(0.23);

console.log(add(1200));

//->this will print the value

//try this easy syntax of arrow functions

const addTAXX=rate=>value=>value+value*rate;
```

Immediately Invoked functions exprssions (IIFE)

```
//NON-iife function
const runonce=function(){

    console.log('This only run once');
};

runonce();
//here we can call i many time as we want




//IIFe syntax ,,,,() covering function inside a --> () makes it as a expression so when we directly call it it calll only once in a execution cycle
```

```
(function(){
    console.log('This only run once!!!!!!!!');
})();

//with arraow funct
(()=>console.log('This will also execute the same'))();
```

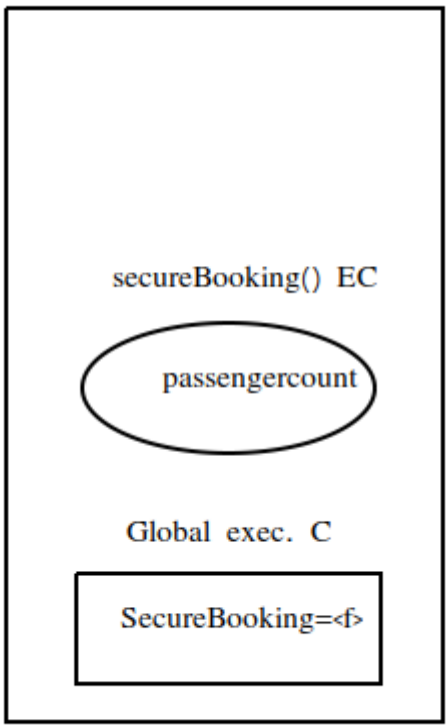
but why we need iife's??? we know function create a scope and what ever we defined inside of that functions those properties are private and we have encapsulated method and that variable together. data encapsulation and data privacy is important in any programming language so iife invented.

```
{
    const pokker='ere';

}
console.log(pokker);
//it will throw the error as it is in block scope
```

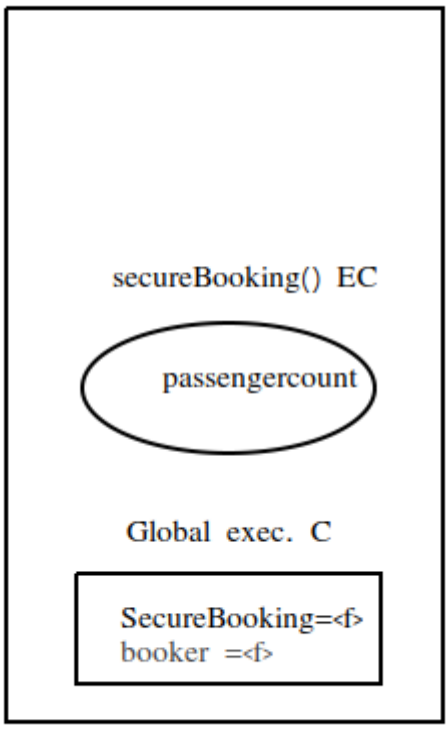
Closures

it's a confusing concept not a hard concept in java script. If u have understood the execution context and how actually js is working, u get to know how closure works.

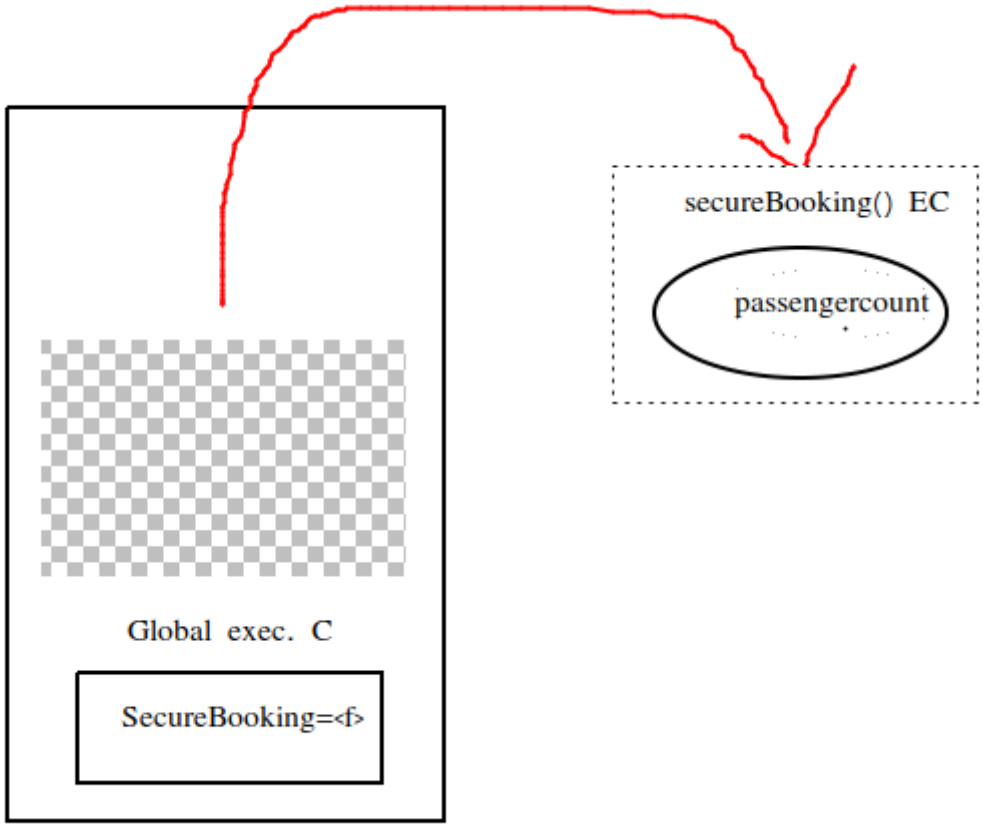
```
//lets understand the clousures with the given example

const securebooking =function(){
    let passengercount=0;

    return function(){
        passengercount++;
        console.log(`${passengercount} passengers`);


    }
}

const booker=securebooking();
console.log(booker);
booker();
```



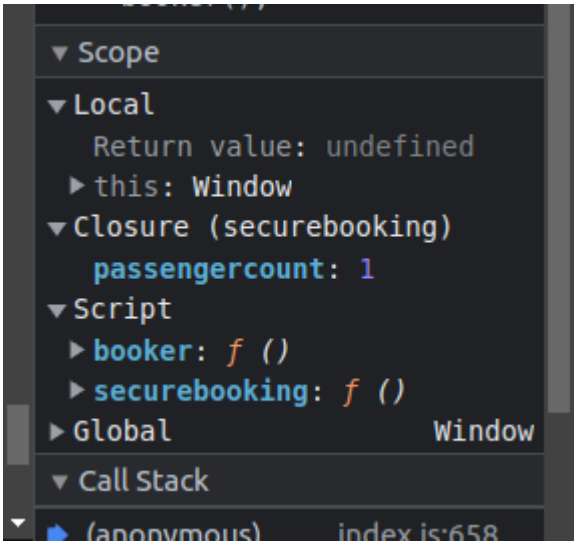
Now first the global exexution context get pushed into the callstack with the Global var or function --> secureBooking() =. then on the top of that, the functional scope or the one more execution context get pushed into the callstack with his local scoped variable passengerCount. Now rember that we can lift the state up or function have access to all the global variable btut the global context do not have access to function's owned local variables.
Here we have saved a new function tht is return by securebooking function inside a variable booker...so global context also have booker variable.boker is nothing but a function.



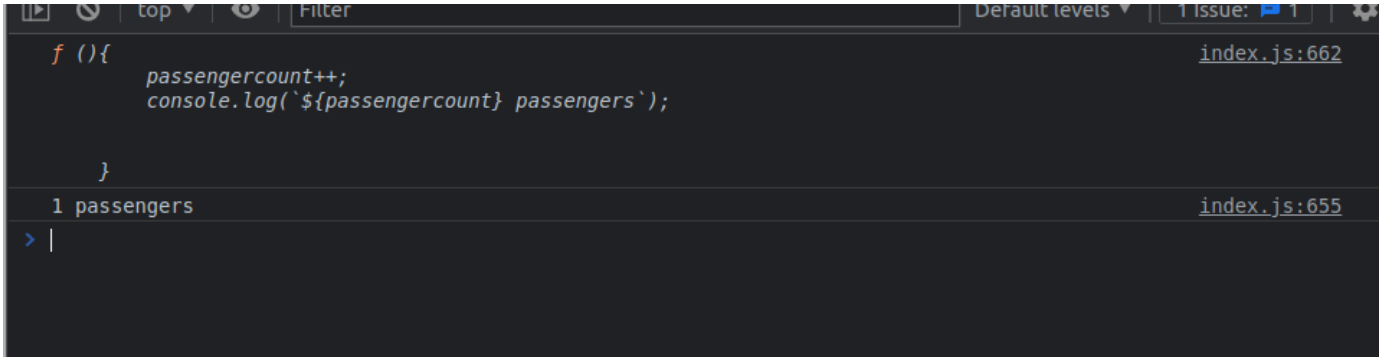
The secureBooking functional context has been poped out from the stack once it has been fully executed.



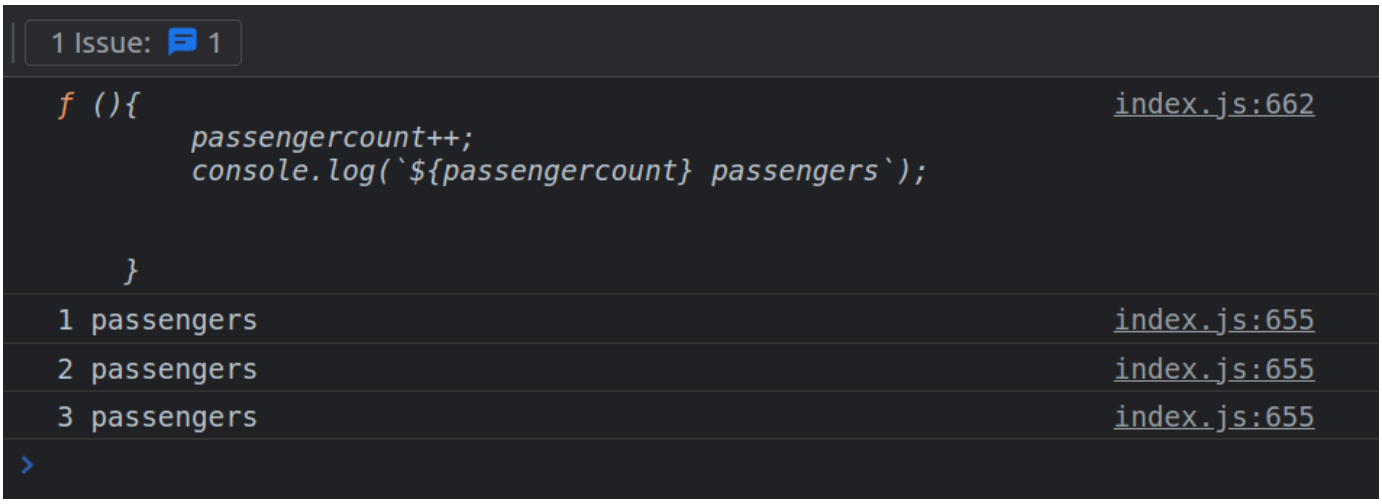
This goin to help us to understand closue in more realistic way;
Lets call out the function multiple times and will goint to chek if that really incrementing the values or not
booker();
booker();
booker();



here we can track that after 1st execution the closure is formed already!!!



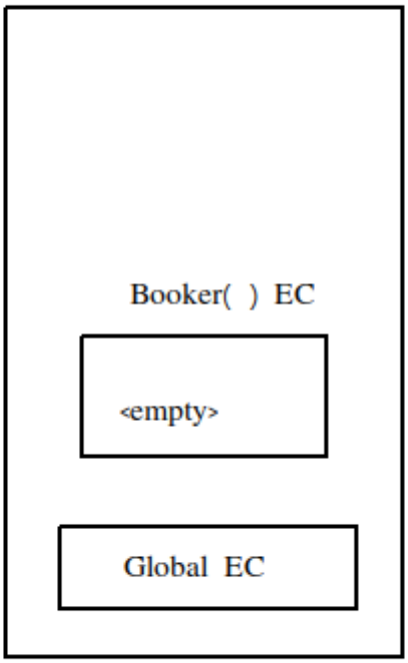
debugging it further and checking if iti incrementing or not!



yaah we can check that the value has been incremented.
How that has done or possible !

How the booker function is Incrementing or updating passenger count after it has been executed and that is inside the booker functions securebooking that no longer exist now.

Appretiate this strage behaviour of js
...
Now check and run the booker function and check what goin to happen -->



while executing the booker function the execution context has been pushed into the stak first with empty var envirnment because no variable has been declared into that context.
Now how the booker gonna find the passengercount variable???
[IMP]
--> Remember that any function always has access to the variabe environment of the execution context in which the function was created. in case of booker the fuction has been born in context of secure booking which was poped of previously. so booker get access to the variable environment in wich passengercount resides.this is how it can able to read and manupualate the passengerrcount variable. that is the closure.

so function has access to the variable environment of the execution context in which it has been created.

Clousure--> Variable envrionment attched to the function, exactly as it was at the tme and place the function was created.

so in defination closure is closed-over variable environment of the execution context in which a function was created, even after that execution context is gone.
or
the closure gives fuction access to all the variables of its parent function, even after that parent function has returned. the function keeps a reference of it's outer scope which preserves the scope chain throught time.

lets try to find what is booker-->
console.dir(booker);

check out there is closure that coming from securebooking
closure(securebooking)

examples of closure:

```
//lets create a situation for closures

let f;
const g=function(){
    const a=23;
    f=function(){
        console.log(a*2);

    }
}

const h=function(){
    const b=777;
    f=function(){
        console.log(b*2);

    }
}
```

```
g();

f(); //at this point the f is different funct
//console.dir(f);

h();

f();//at this time it is different
//console.dir(f);
```

lets print f function..
console.dir(f);
//when with context of function --> g

with cotext of function h -->

[Article]

What is a Closure?

A closure is the combination of a function bundled together (enclosed) with references to its surrounding state (the lexical environment). In other words, a closure gives you access to an outer function's scope from an inner function. In JavaScript, closures are created every time a function is created, at function creation time.

To use a closure, define a function inside another function and expose it. To expose a function, return it or pass it to another function.

The inner function will have access to the variables in the outer function scope, even after the outer function has returned.

Using Closures (Examples)

Among other things, closures are commonly used to give objects data privacy. Data privacy is an essential property that helps us program to an interface, not an implementation. This is an important concept that helps us build more robust software because implementation details are more likely to change in breaking ways than interface contracts.

In JavaScript, closures are the primary mechanism used to enable data privacy. When you use closures for data privacy, the enclosed variables are only in scope within the containing (outer) function. You can't get at the data from an outside scope except through the object's privileged methods. In JavaScript, any exposed method defined within the closure scope is privileged.

Closure example 2

```
//example 2
```

```
//timer is another greate example to understand closure

const boardPassenger=function(n,wait){
    const perGrp=n/3;
    setTimeout(()=>{
        console.log(n);
        console.log(`${perGrp} groups boarded`);},3000);
    console.log(`will resume in ${wait}`);
}

boardPassenger(12,23);


// will resume in 23
// 12
// 4 groups boarded
```

closures have priority over the scope chain

Examlple of iife function

```
(function(){
    const header1=document.querySelector('h1');

    header1.style.color='red';

    document.querySelector('body').addEventListener('click', function() {
        header1.style.color='blue';

    })

})();
//this  goin to change the h1 tag color after clicking on body to blue for one time only.
```

Arrays and Project Basis on arrays.

checkout the array methods...
array itselfs are objects so this fuctions are predefined for them and user can use them easily,,

1.Slice
2.Splice //this one also mutate the array

3.reverse //this method mutate the array
4.concat
5.Join
6.at

for each method

```
const movement=[200,450,-23,130,123213,140,-23123];
//without for each loop
for (const movements of movement){
    if(movements>0){
        console.log(`you have deposited ${movements}`);
    }
    else{
        console.log(`you have withdraw ${movements}`);

    }
}

//simple using for each loop

movement.forEach((element)=>{
    if(element>0){
        console.log(`you have deposited ${element}`);

    }
    else{
        console.log(`you have withdraw ${element}`);


    }
})
```

for each is actually a higher order function and it need a callback function inside. foreach call that callback function and pass its current element inside of it.

argument inside of the for each callback

```
movement.foreach((currentelement,index,array){
    console.log(currentelement,index,array);
    });
```

but out oof this three we use what we needed to fullfill the scenario

```
//in a a for of loop we use entries function which return us a
array, while destructuring it the first parameter is index and second is
array

for(const [i,movement] of movements.entries()){

    }
```

for each loop over a map
it also take first value as currentvalue or respective key and second argument is key while third argument is entire map.

```
const currenncies=new Map([['USD','United state Doller'],['EUR','EUro'],
['Rupee','IndianRupee']]);

currenncies.forEach((currValue,key,map)=>{
    console.log(currValue,key,map);

})
```

same over a set also goin to work...
same the currentvalue, key and entire set has been added as args of callbeck function inside the foreach for sets also but set do not have indexes and keys ,... !!! it was decided by js developert to keep it as it is to avoid confusion.

```
const currentCurrencies=new Set(['USD','GBP','UPS','EURO']);

currentCurrencies.forEach((currval,key,set)=>{
    console.log(currval);
    console.log(key);
    console.log(set);
})
```

-->Promises-->>

In meantime try to understand below Code which is base on Promises, we are goin to catch those concept in next sessions

```
const Mpp=new Map([['biskut',124],['chees',1244],['pizza',1234]]);

let walletBalance=2000;

const numberOforders=prompt('Give the number o things u have to order');
const numberOforders1=Number(numberOforders);
```

```
const cart=new Set();

for (let index = 0; index < numberOforders1; index++) {
    const input=prompt('give ur order, we have chees,pizza and biskut');
    cart.add(input);

}
```

```
//promise chaining

const promise=createorder(cart);

promise.then(function(orderID){
    return orderID;
}).then(function(orderID){
    let payAmount=proceedToPay(orderID,cart,walletBalance);
    //console.log('print inside a chain'+ payAmount)
    return payAmount;
}).then(function(payAmount){

     showOrderSummery(cart,payAmount,walletBalance);
     return payAmount;
}).then(function(payAmount){
     updateWallet(payAmount,walletBalance);
}).catch(function(reject){
    console.log(reject);

});
```

```
function createorder(cart){

    const pr=new Promise((resolve,reject)=>{
        if(cart.size){
            console.log(cart.size +'cart is not empty');

            const orderID="1234";
            console.log(orderID);

            setTimeout(()=>{
                if(orderID){
                resolve(orderID);
                }
                console.log(pr);
```

```
                },1000);

        }
        else{
            reject('The cart is empty');
        }
    });
    return pr;
```

```
function proceedToPay(orderID,cart,walletBalance){

        let paybleAmout=0;
        console.log(cart);
        console.log(orderID);

        cart.forEach(function(item){

            console.log(item);
            const price=Mpp.get(item);
            console.log(price);
            if(price){
                //walletBalance-=price;
                paybleAmout+=price;
                console.log('cakculating payble amount');

            }
            if(walletBalance==0){
                console.log('U do not have enough balance to pay, Do u need
any other option??');

            }
        })

    console.log('the payble amount-->'+ paybleAmout);
        return paybleAmout;

}
```

```
function showOrderSummery(cart,payAmount,walletBalance){
    for(const item of cart){
        console.log(item, '---> '+ Mpp.get(item));
```

```
    }
    console.log(payAmount);
    console.log(walletBalance);
}
```

```
function updateWallet(payAmount,walletBalance){
    console.log(payAmount);
    walletBalance= Number(walletBalance)-payAmount;
    console.log(walletBalance);

    if(walletBalance<0){
        alert(`Can't proceed to place your order as You have to add ${Math.abs(walletBalance)} inside a wallet`);

    }
    else{
        alert('Thanks for ur order');
    }

}
```

Try to understand the code. if u aint, do not worry we gonna catch those topics after the Object oriented programming concpts

Advance DOM and Events

topics covered::

How DOM works
Selecting creating ,deleting Elements
Style ,attribute and classes
Event and Event Handler
DOM Traversing
...

Remember that the DOM is very complex API and that contains a lot of methods and properties with DOM tree

DOM is abbreviation for Document Object Model. It’s a tree like structured representation of HTML elements. For sake of simplicity, let’s consider a simple scenario. When you enter an URL in the browser, browser asks web server (where that URL points to), to return simple HTML document. That document contains nothing but plain text but written in HTML language (text/html). When browser get’s the response, that text is used to print different segments and content on the screen with different styles and behaviour. How browser does that from simple HTML text? This is where DOM comes into picture.

Every DOM tree starts from document node. All these nodes are mainly divided into 5 categories.

- DOM tree starts from document node.
- All HTML elements have their separate nodes known as element nodes.
- Any text or tab character on a single line forms one text node. new line character is also a text node.
- HTML attribute is also a node knows as attribute node.
- Comment is also node in DOM known as comment node.

Once browser creates DOM tree, these nodes are accessible from JavaScript using API provided by document.

[read out a full article on DOM]

Click here

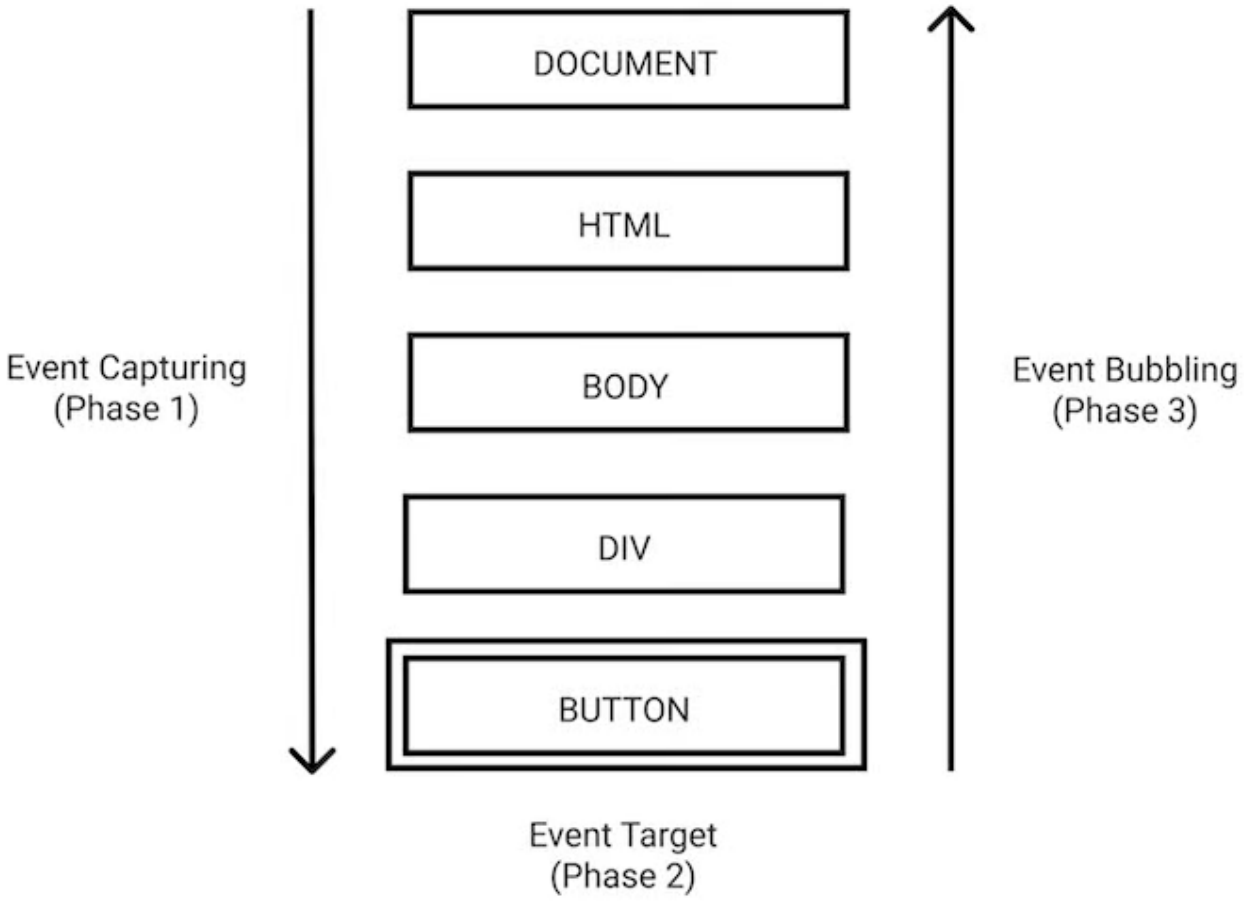
ways to Listen a events

```
h1.addEventListner('Mouseenter',()=>{
alerrt('');
})
```

Checkout mdn for events, checkout the events available.

Eveent Bubbling and capturing In js -->

Event has bubbling phase as well capturing phase.



Event Bubbling and Event Capturing is the most used terminology in JavaScript at the time of event flow. In the JavaScript, Event Flow process is completed by three concepts :

Event Capturing.
Event Target.
Event Bubbling.

Events :
Events are responsible for interaction of JavaScript with HTML web pages. The general definition of event is any act can occur by someone. In the web development, the definition of events is also same. Events can be subscribed by listeners that occurs only when the particular event can be fired.
Event Flow :
Event flow is the order in which event is received on the web page. If you click on an element like on div or on the button , which is nested to other elements, before the click is performed on the target element. It must trigger the click event each of its parent elements first, starting at the top with the global window object. By default, every element of HTML is child of the window object.

History of Event Flow :
In the Fourth Generation of Web browser War, main browser community, Internet Explorer 4 and Netscape Communicator 4 met with each other for searching the solution to the way of Event flow.

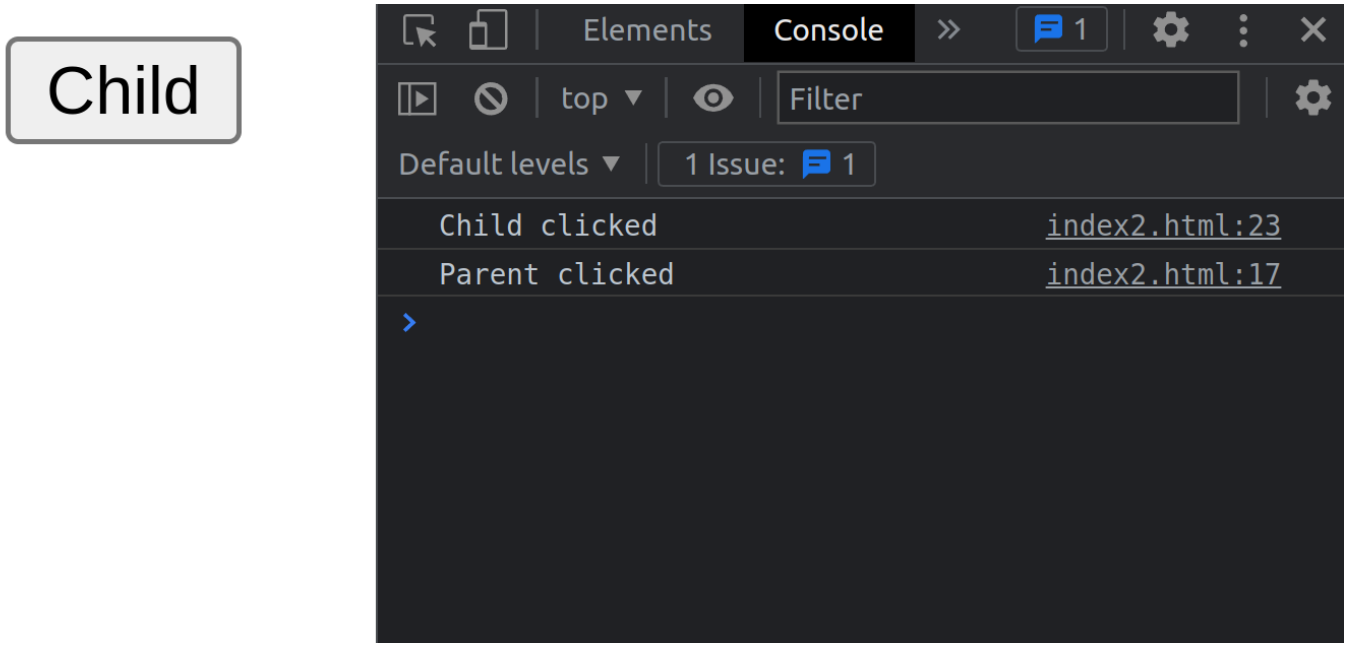
Basically, both developer teams met with each other to discuss which way is more suitable for Event Flow. There are two ways Top to Bottom(Event Capturing) and other one is Bottom to Top (Event Bubbling). But unfortunately, both of them apply opposite approach. Internet Explorer 4 adopts the Event Bubbling approach and Netscape communicator 4 adopts Event Capturing approach.

Event Bubbling :
Event Bubbling is the event starts from the deepest element or target element to its parents, then all its ancestors which are on the way to bottom to top. At present, all the modern browsers have event bubbling as the default way of event flow.

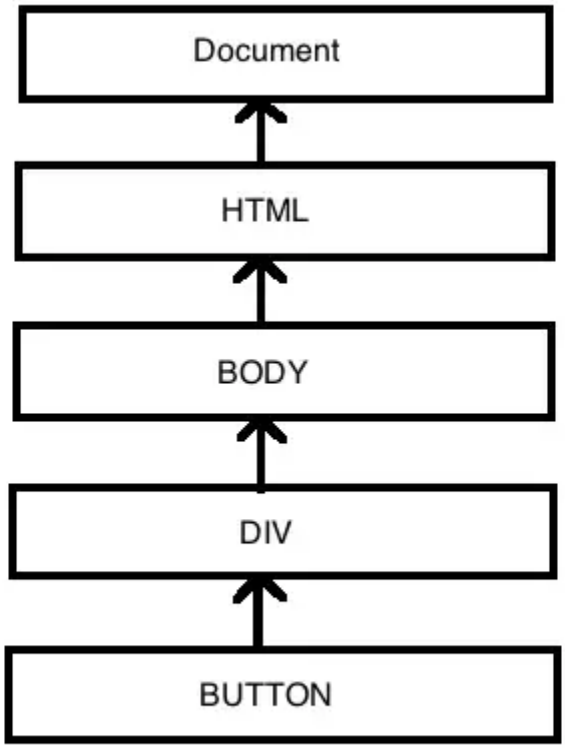
Consider an example on Event Bubbling :

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>Event Bubbling</title>
</head>
<body>
  <div id="parent">
    <button id="child">Child</button>
  </div>

  <script>
    var parent = document.querySelector('#parent');

      parent.addEventListener('click', function(){
        console.log("Parent clicked");
      });

    var child = document.querySelector('#child');

      child.addEventListener('click', function(){
        console.log("Child clicked");
      });
  </script>
</body>
</html>
```

O/P->



after clicking on child element the event related to that of the child has invoked and event is passed it to the above of his hierarchy and catched by parent also and after that the function or event related to that of the parent wlll also gets invoked. thats what we can called the event has gets buubbled up from the target element of an event.



Event Bubbling

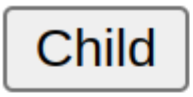
[IMPORTANT]

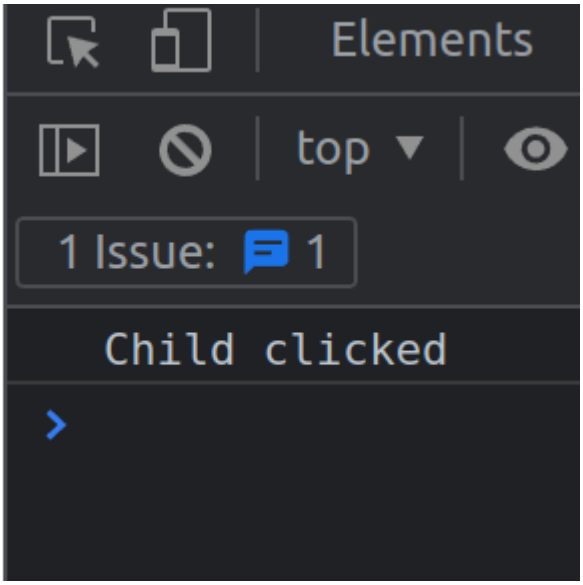
Stop Event Bubbling :

If you want to stop the event bubbling, this can be achieved by the use of the

event.stopPropagation() method. If you want to stop the event flow from event target to top element in DOM, event.stopPropagation() method stops the event to travel to the bottom to top.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>Event Bubbling</title>
</head>
<body>
  <div id="parent">
    <button id="child" onclick="event.stopPropagation()">Child</button>
  </div>

  <script>
    var parent = document.querySelector('#parent');

      parent.addEventListener('click', function(){
        console.log("Parent clicked");
      });

    var child = document.querySelector('#child');

      child.addEventListener('click', function(){
        console.log("Child clicked");
      });
  </script>
</body>
</html>
```





here we have stoped the event to get bubbled up or we just have stop the event gets propogated from the target event element to that of the parents in the hierarchy.

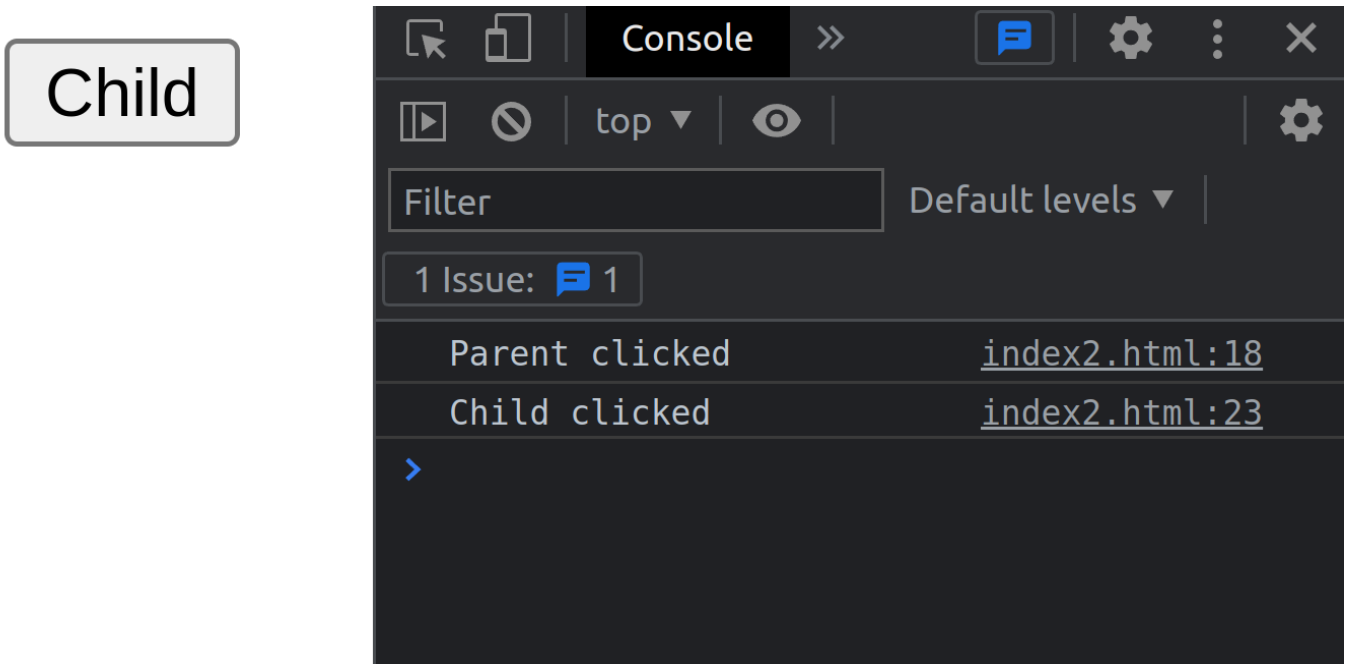
<<----------Event Capturing---------->>

Event Capturing :
Event Capturing is the event starts from top element to target element. Modern browser doesn't support event capturing by default but we can achieve that by code in JavaScript.

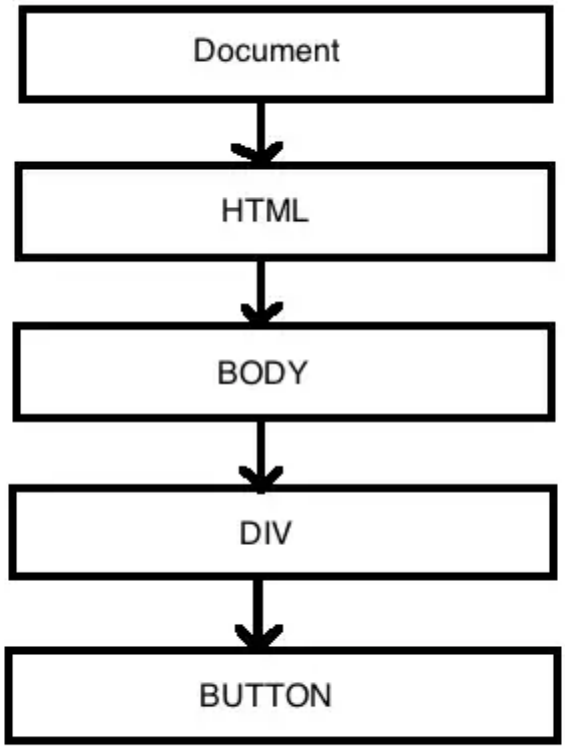
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>Event Capturing</title>
</head>
<body>
  <div id="parent">
    <button id="child">Child</button>
  </div>

  <script>
    var parent = document.querySelector('#parent');
    var child = document.querySelector('#child');

    parent.addEventListener('click', function(){
      console.log("Parent clicked");
    },true);
```

```
    child.addEventListener('click', function(){
      console.log("Child clicked");
    });
  </script>
</body>
</html>
```

when adding a event listner, inside of addEvenntListner method we have 3rd argument to enble event capturing, we have to set it true inorder to activate the event capturing.

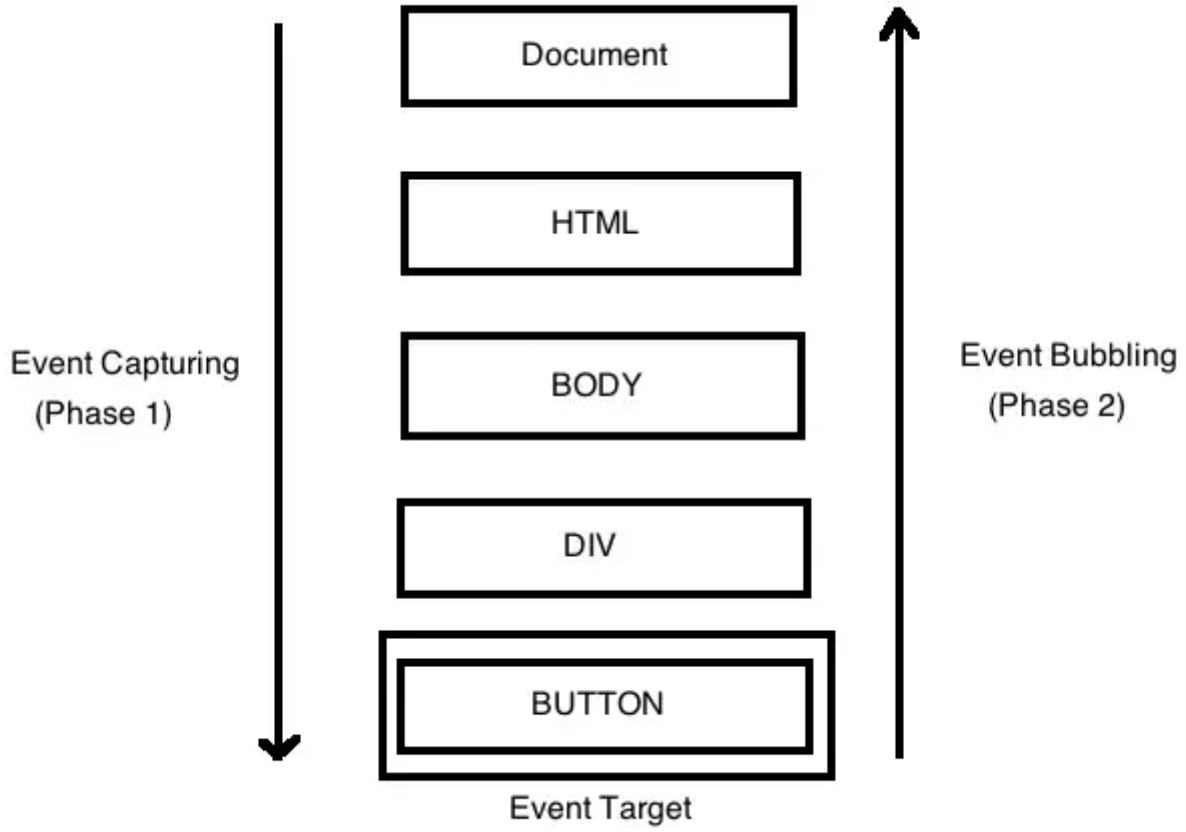


- In the Javascript code, Firstly we can assign the html element to the variable with the help of document.querySelector()function
- After that we can attach, a click event to parent div and child button also. and both of function just print the strings value on the console by the use of console.log().
- We can use third optional argument of addEventListner to set true to enable event capturing in the parent div.
- When we click on the button first run the function which is attached on div , after that onclick function of button runs. This is due to Event Capturing. First run the event which is attached with parent element then event target.



Event Capturing

<---------Complete flow--------->



End Of Topic

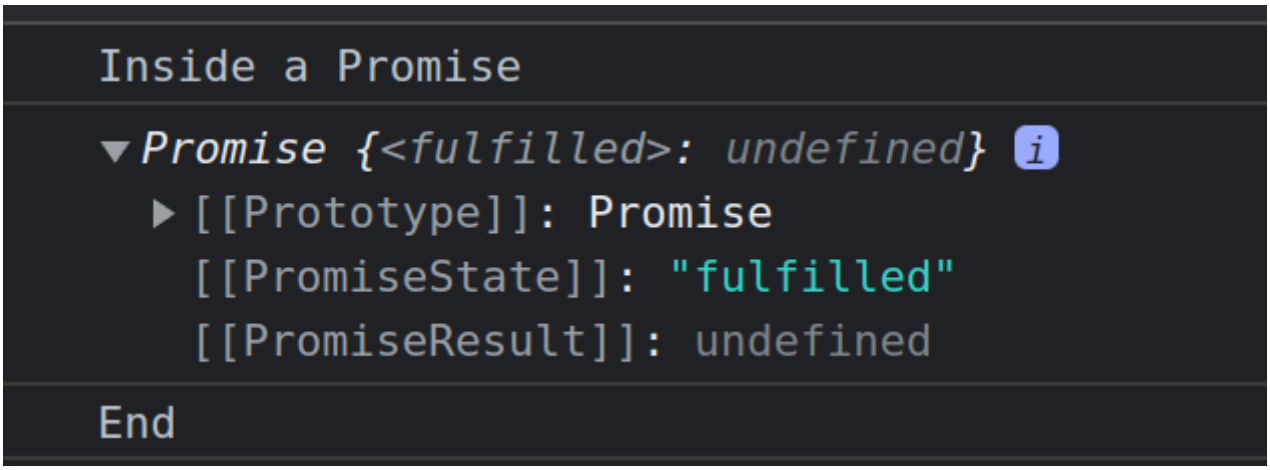
Promises

here we are goin to learn promises Through examples

Promise constructor-->

```
var promise1=new Promise((resolve,reject)=>{
    console.log('Inside a Promise');
    resolve();

})

console.log(promise1);
console.log('End');
```

Synchronized code blocks are always executed sequentially from top to bottom.
When we call new Promise(callback), the callback function will be executed immediately.
so output is-->



then() ---->

```
var promise1=new Promise((resolve,reject)=>{
    console.log('Inside a Promise');


    resolve("the value has been passed here");

})

promise1.then(()=>{
    console.log('I am inside then-s callback')
```

```
})

console.log(promise1);
console.log('End');
```

In this code snippet, a piece of asynchronous code appears. That is, the callback function in .then().

Remember, the JavaScript engine always executes synchronous code first, then asynchronous code.

When encountering this problem, we only need to distinguish between synchronous code and asynchronous code.
o/p-->

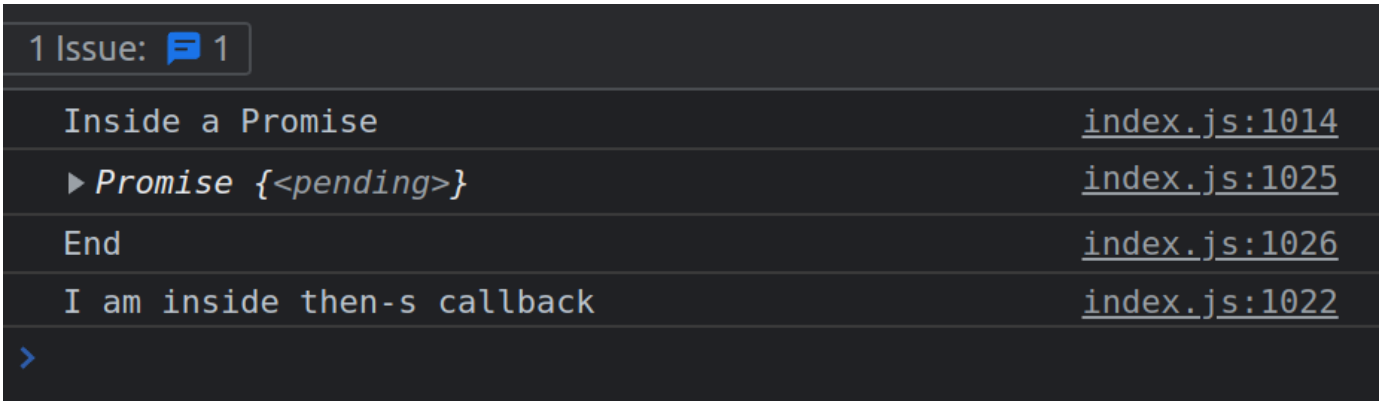
```
Inside a Promise                                    index.js:1014
                                                    index.js:1025
▶ Promise {<fulfilled>: 'the value has been passed here'}
End                                                 index.js:1026
I am inside then-s callback                         index.js:1022
```

We ow that once the promise gets fullfilled only then the callback inside that of the the method will goin to invoke, Lets understnd it clearlywith an example of setTimeOut( );

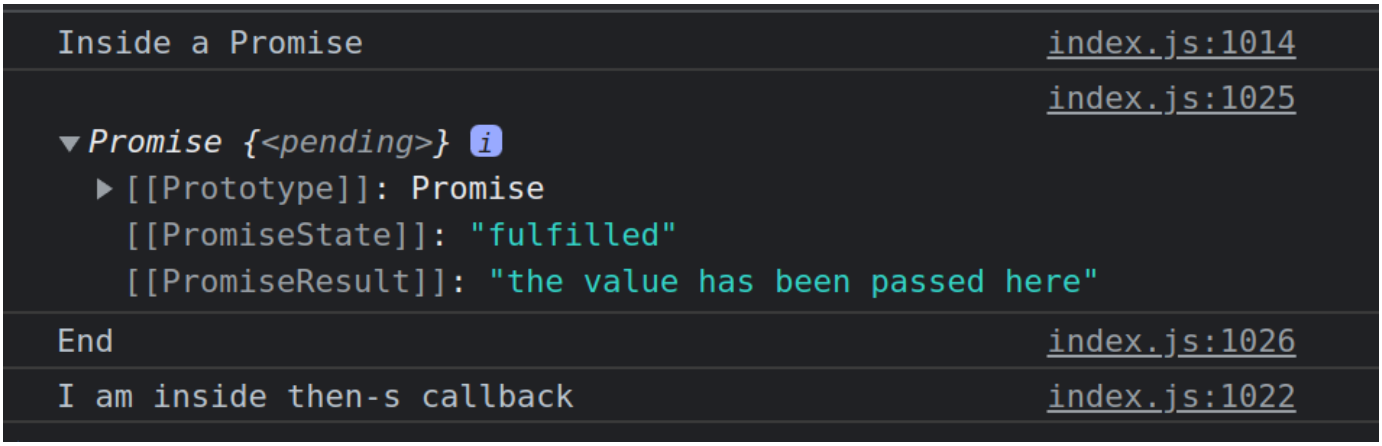
```
var promise1=new Promise((resolve,reject)=>{
    console.log('Inside a Promise');

    setTimeout(()=>{ resolve("the value has been passed here");
},4000);

});

promise1.then(()=>{
    console.log('I am inside then-s callback')
})

console.log(promise1);
console.log('End');
```

time has not been fully elapsed.

```
Inside a Promise                                    index.js:1014
▶ Promise {<pending>}                               index.js:1025
End                                                 index.js:1026
>
```



The promise has been resolved but js logging it as pending, checkd with the issue but it's JAVAscripts behaviour.
check it ot it's weird but okay!



!!!!

What is the proof that resolve method does not hamper with our code and interrupt the current thread

lets check it out!!!

```
var promise1=new Promise((resolve,reject)=>{
    console.log('Inside a Promise');

    resolve("the value has been passed here");
    console.log("Resolve function not interruption sync flow")



});

promise1.then(()=>{
    console.log('I am inside then-s callback')
})
```

```
console.log(promise1);
console.log('End');
```

```
Inside a Promise                                        index.js:1014
Resolve function not interruption sync flow             index.js:1017
                                                        index.js:1027
▼Promise {<fulfilled>: 'the value has been passed here'} i
  ▶[[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: "the value has been passed here"
End                                                     index.js:1028
I am inside then-s callback                             index.js:1024
>
```

Remember, the resolve method does not interrupt the execution of the function. The code behind it will still continue to execute.

Function Returning Promise*

```
var funcc=()=>{
    return new Promise((resolve,reject)=>{
        console.log("Im Inside Promise's callback");
        resolve("Resolving promises");
        console.log('The Resolve over');

    })


}
console.log(funcc());

//onsole.log(funcc());
funcc().then(()=>{
    console.log('Inside a then callback');
}
)

console.log('END');
```

o/p--->

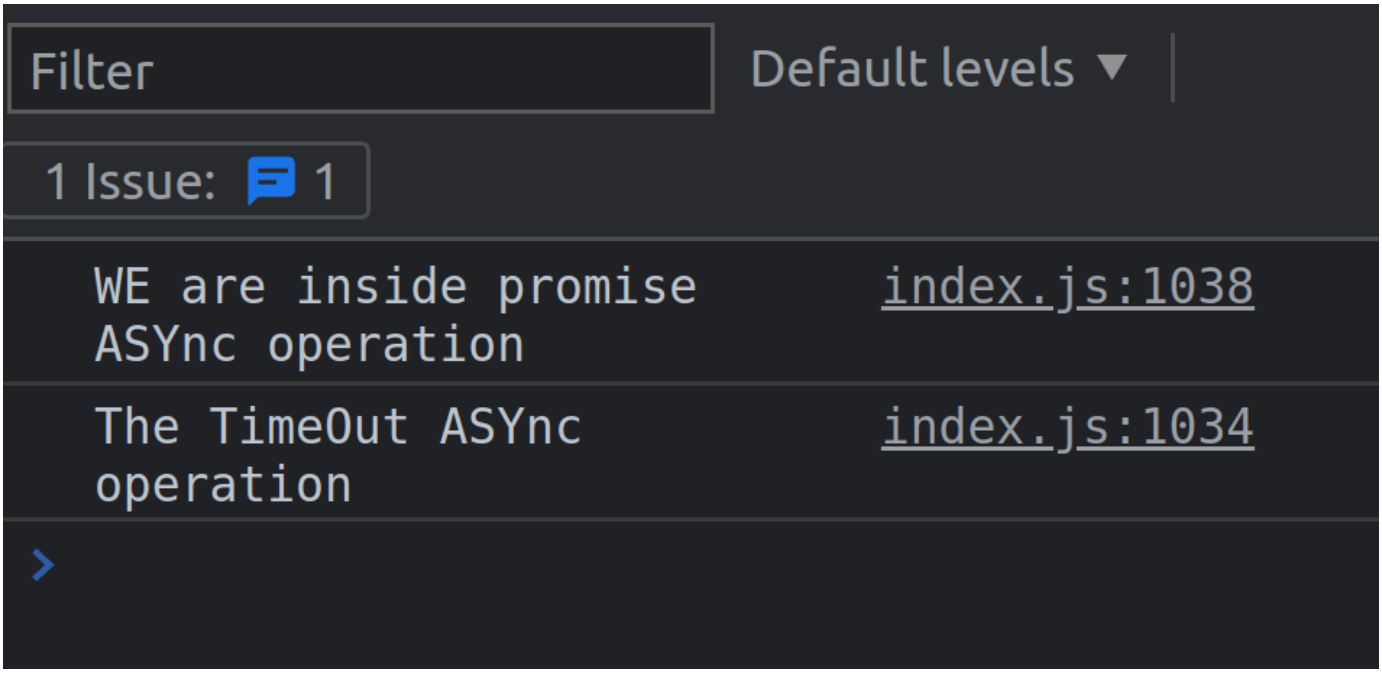
```
Im Inside Promise's callback                    index.js:1015
The Resolve over                                index.js:1017
▼Promise {<fulfilled>: 'Resolving promises'} i  index.js:1023
  ▶[[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
    [[PromiseResult]]: "Resolving promises"
Im Inside Promise's callback                    index.js:1015
The Resolve over                                index.js:1017
END                                             index.js:1031
Inside a then callback                          index.js:1027
```

Two async Operations

```
setTimeout(()=>{console.log('The TimeOut ASYnc operation')});


Promise.resolve().then(()=>{
    console.log('WE are inside promise ASYnc operation')
})
```

Now setTimeout and promises are both work asynchronous especially the method then will be act as a async and run after the promise get fullflled but remember that here the settimeout also have 0 second timer!!!! Whs's goin to execute first???
check with ur own eyes!!!

That's amazing!!!!



We know that many things are NOT performed in a first-in, first-out order, such as traffic.

Priority
We generally divide vehicles into two categories:

General vehicles
Vehicles for emergency missions. Such as fire trucks and ambulances.
When passing through crowded intersections, we will allow fire trucks and ambulances to pass first.
Emergency vehicles have more priorities than other vehicles. Keywords: priorities.

In JavaScript EventLoop, there is also the concept of priority.

Tasks with higher priority are called microtasks. Includes: Promise, ObjectObserver, MutationObserver, process.nextTick, async/await .
Tasks with lower priority are called macrotasks. Includes: setTimeout , setInterval and XHR .

Tasks with lower priority are called macrotasks. Includes: setTimeout , setInterval and XHR!

Although setTimeout and Promise.resolve() are completed at the same time, and even the code of setTimeout is still ahead, but because of its low priority, the callback function belonging to it is executed later.
[00026d63e2790635f738b95905d38fa6.png]

Thats the reason why i have trust issues with setTimeout!!!!!!

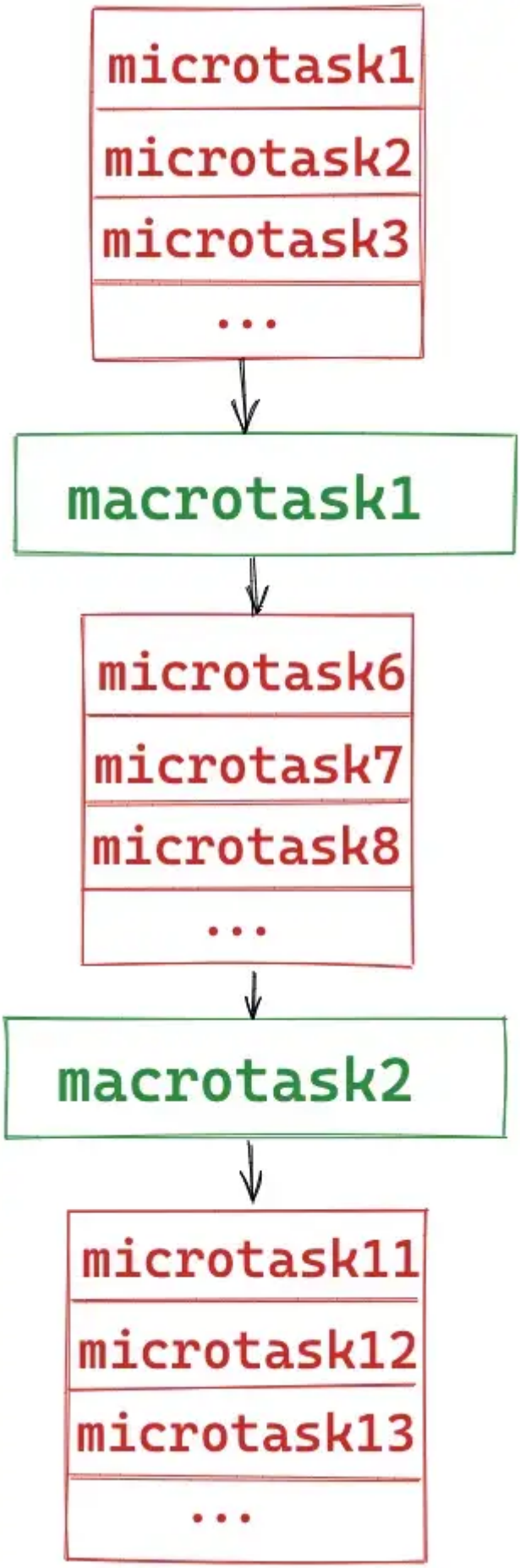
But aks a question that what if we are resolving the promises inside a setTimeout, does that unresolved promise which have high priority will never ever give chance to the settimeout to run???? But, thats how not js ENgine Works!!!

```
const promise = new Promise((resolve, reject) => {
    console.log(1);
    setTimeout(() => {
      console.log("timerStart");
      resolve("success");
      console.log("timerEnd");
    }, 0);
    console.log(2);
  });
```

```
promise.then((res) => {
  console.log(res);
});

console.log(4);
```

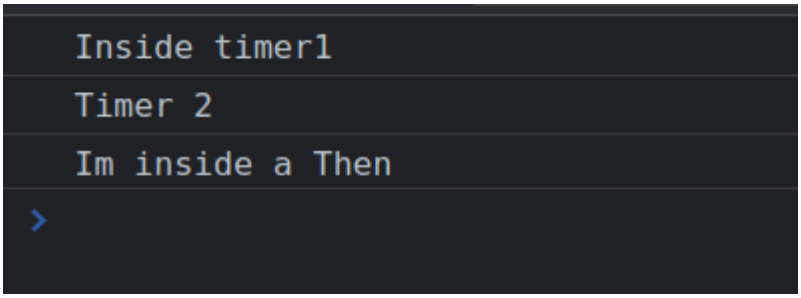
in case the current promise is still in the pending state, the code in this will not be executed at present.Then execute macrotask!!!
And the state of the promise becoming to fulfilled .Then, with Event Loop, execute the microtask again:

priority between Microtask Queue and MAcrotask Queue!!!




```
const Timer1=setTimeout(()=>{
        console.log('Inside timer1');
    },0)
```

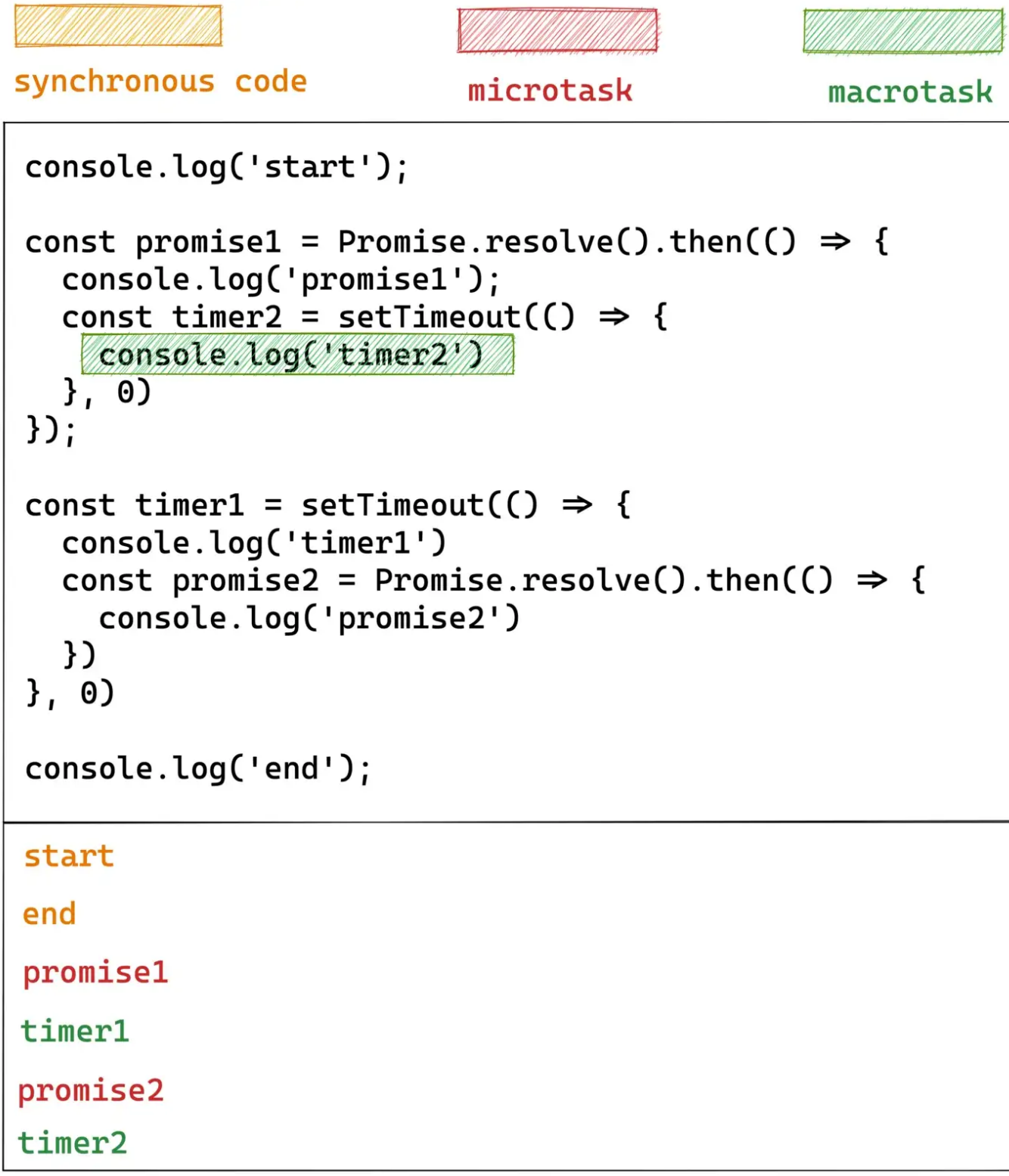
```
const Timer2=
    //console.log('Inside Timer 2');
    setTimeout(()=>{
        console.log('Timer 2')
        Promise.resolve().then(()=>{

            console.log('Im inside a Then');

        })

    },0);
```



Priorities are hard to understand for begginer but chekout the codes to understand it more, read out the articles

example on microtask and macrotask queue!!

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

const promise1 = Promise.resolve().then(() => {
  console.log('promise1');
  const timer2 = setTimeout(() => {
    console.log('timer2')
  }, 0)
});

const timer1 = setTimeout(() => {
  console.log('timer1')
  const promise2 = Promise.resolve().then(() => {
    console.log('promise2')
  })
}, 0)

console.log('end');
```



Create/ Thought a stack structure in your mind and predict how it's printing like this.

Object Oriented Programming

classses and instanaces
(traditional OOP)

Js does not support real clasees like represented,

```
User{
        user
       passwprd
       email


       method login();
        }
```

Four fundamental principle of oop

1.Abstrasction
2.encapsulation
3.Inheritence
4.Polymorphismm

1. Abstraction is igonoring or hiding the details that dont matter, allowing us to get the overview perspective of things we are implementing, Insted of messing with details that dont matter to our implementation .

class Phone{
charge
volume
voltage
temperature

Ram(){};
Rom(){};
VolumeButton(){;
voltage(){};
charging(){};
}

so when I am using this device or phone I do not have to care about the internal implementtation. That is what we can say Abstraction!!!

Encapsulation-->
Keeping athe properties and methods private inside a class so they are not accesssible outside of the classs. some of the methods can be exposed as a Public interface (API)
(Private keyWord not exist in JS)

how to create prototypes -->there are 3 methods

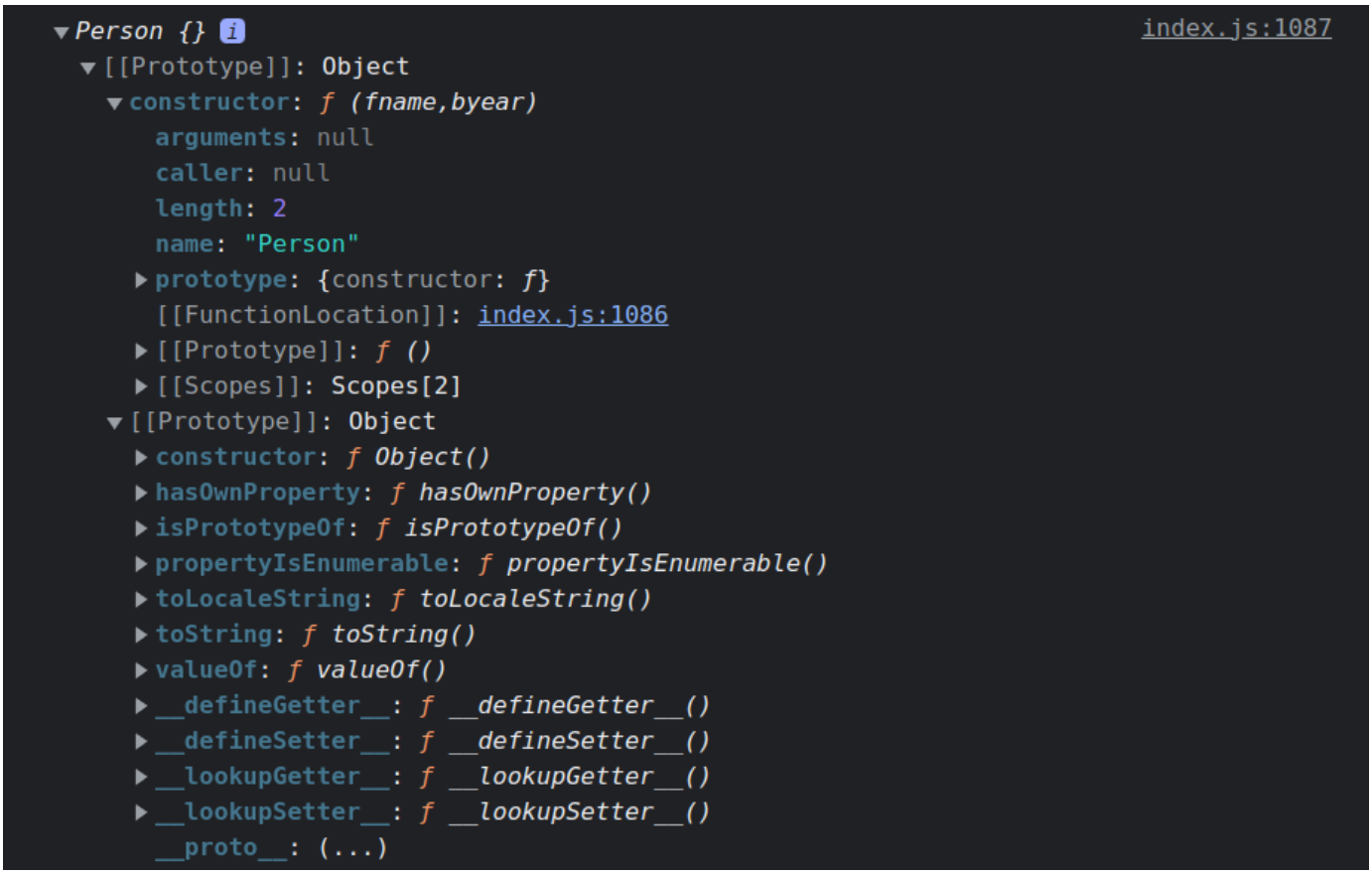
- constructor function ->technique to create a object from a function!
- ES6classes --

- object.create -->easy method and straight forward way!!!

constrctor function and this keyword

```
//constructor function and new word
//constrctor function start with capital letter
//only function declaration and expression works here because arrow function
do not have there own this keyword
const Person=function(fname,byear){
console.log(this);
}

new Person('Jonas',1999);
```

there is procedure that
1.new object {} empty object is created firstly
2.this keyword is pointing to that of empty object, in function call.
3.newly created object {} now linked to prototype
4.now the object created automatically return through function. or function automatically return { } / object.



look at the picture above we have printed the this keyword inside the object and this is pointing to theempty object or constructor function of Person.

checkout how we goin to use construcetor function

```
const Person=function(fname,byear){
// console.log(this);

this.fname=fname;
this.byear=byear;

}

const jonas=new Person('Jonas',1999);
console.log(jonas);
```

object created from a class is Instance of that class but as we know js do not have classes like traditional OOP.
console.log(jonas instanceOf Person) will return true.

when we are creating propertied inside of the constructor functions we can also create a function as shown in below code, but it is a bad practice if there are 1000'os of object of the constrctor func then all objects are carrying that method without any sense, to cope with that we goin to see prototype and inheritence in next session

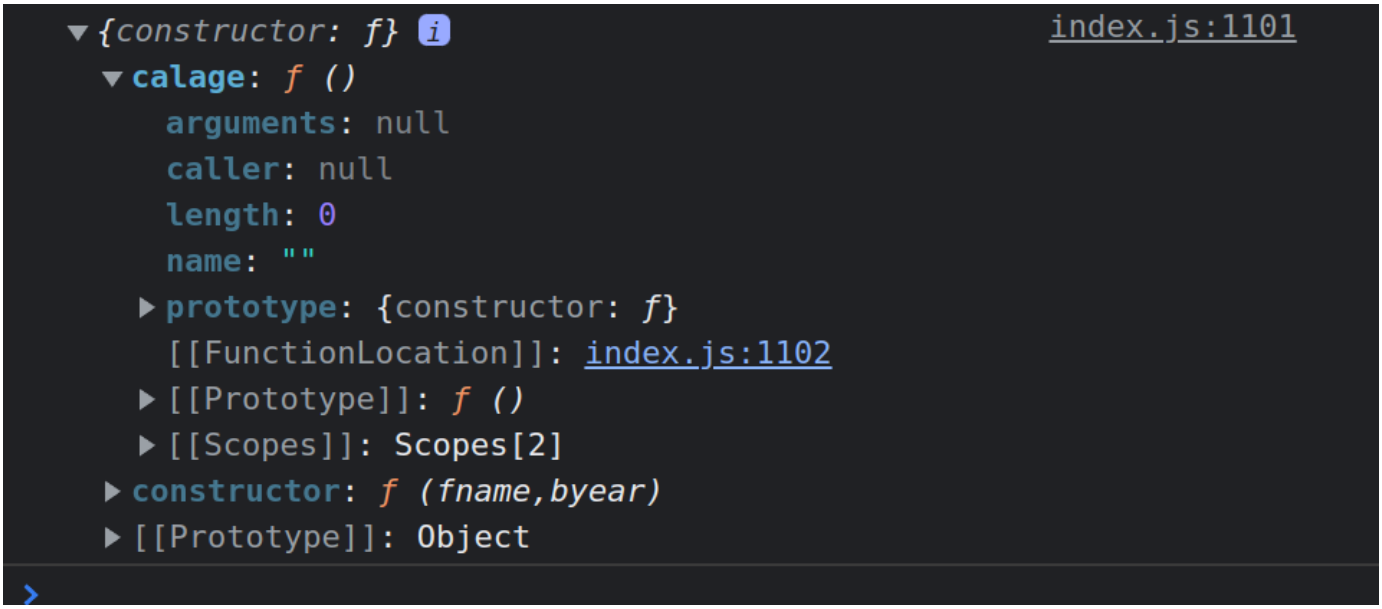
```
const Person=function(fname,byear){
// console.log(this);

this.fname=fname;
this.byear=byear;
//this is bad practice , do not create a function inside of constructor function
this.calage=function (){
    return 1999-this.byear;
}
```

```
}
```

Prototypes

```
const Person=function(fname,byear){
// console.log(this);

this.fname=fname;
this.byear=byear;
//this is bad practice , do not create a function inside of constructor
function




}

// const jonas=new Person('Jonas',1999);
// console.log(jonas);
//this new opearator or we are calling function with new operator

console.log(Person.prototype);
Person.prototype.calage=function (){
    return 1999-this.byear;
}
```

lets print what is actually Prototype is-->



Look calage is defined inside that of the prototype and our object gets liked with that prototype and we know that the object can access the method of prototypes as array can access Array.prototype samme here!!!

Now we can access the calage function !

```
const Saurabh=new Person('Saurbh',2000);

console.log(Person.prototype);
Person.prototype.calage=function (){
    console.log( 1999-this.byear);
}

Saurabh.calage();
```

```
▶ {constructor: ƒ}                                    index.js:1102
-1                                                    index.js:1104
>
```

we have access to that method just because of prototype inheritence. we also call this Delegation.

if we print prototype of obeject saurabh-->
console.log(saurabh.__ proto __);

```
▼ {calage: ƒ, constructor: ƒ} i                       index.js:1109
  ▼ calage: ƒ ()
      arguments: null
      caller: null
      length: 0
      name: ""
    ▶ prototype: {constructor: ƒ}
      [[FunctionLocation]]: index.js:1103
    ▶ [[Prototype]]: ƒ ()
    ▶ [[Scopes]]: Scopes[2]
  ▶ constructor: ƒ (fname,byear)
  ▶ [[Prototype]]: Object
>
```

Person.prototype is not the prototype of Person , ,remember this is most confusing part as the ....It's the prototype of the Objects that are created with Person constrtor.

lets check it out

```
console.log(Person.prototype.isPrototypeOf(Person));
console.log(Person.prototype.isPrototypeOf(Person.prototype));
console.log(Person.prototype.isPrototypeOf(Saurabh));
```

console.log(Person.prototype.isPrototypeOf(Person));
console.log(Person.prototype.isPrototypeOf(Person.prototype));
console.log(Person.prototype.isPrototypeOf(Saurabh));

```
false
false
true
```

Yaah that sounds weird!!! But, It's JS!!!!!!

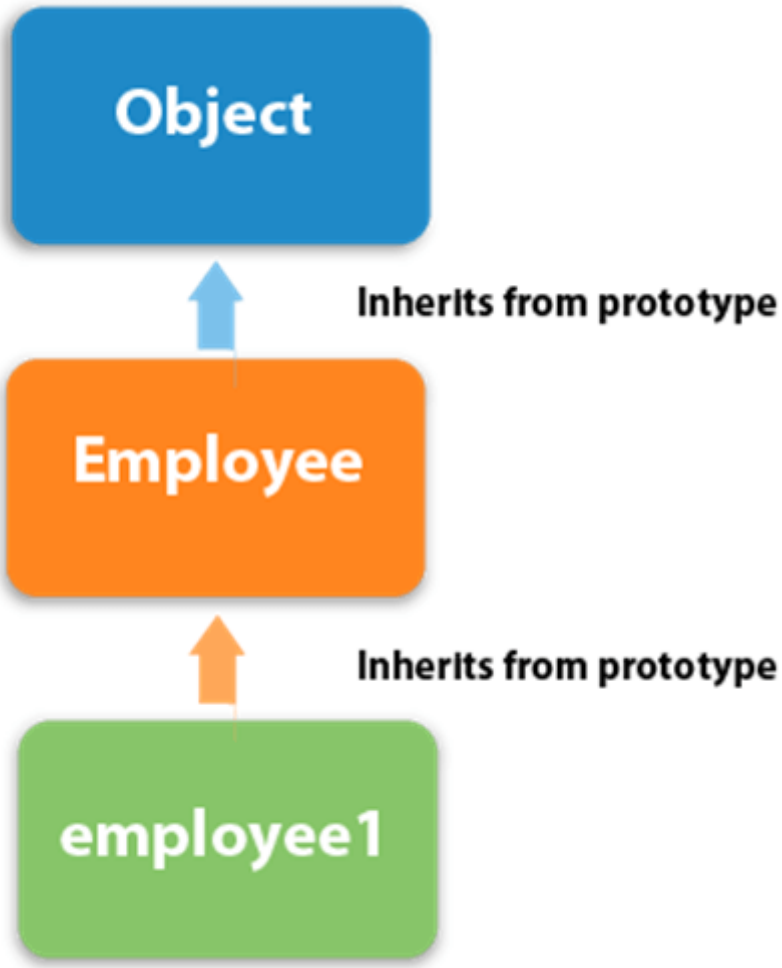
and one more question!! that where the **proto** property cme from on the object of person -Saurabh;
--> Remember the 4 steps of how the new keyword acually works in Js
1.First new object/ empty object gets created.
2.function gets calles and this keyword gets assigned to the empty object.
3.Now 3rd step is the ans , as the empty object gets linked with the prototype there only the property came into the picture. so that we can access saurabh.__ proto __;
4.In 4th step the function will return the empty object as we have try to log this value inside function and it is a empty object.

We can set properties also on prototypes !!!

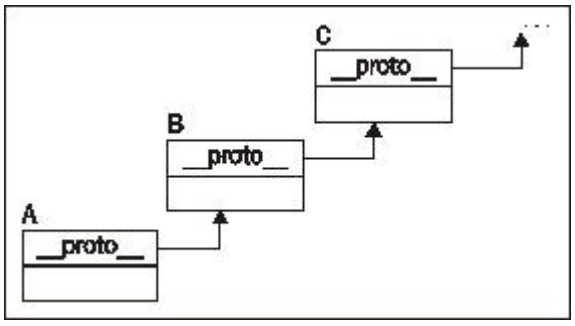
```
Person.prototype.species-"hOMO, Erectus";
console.log(saurabh,species);


//now remember the property species is inherited not actuallyown by saurabh object so if we try to log saurabh.hasOwnPropery('Species') --> it will return false;


saurabh.hasOwnProperty('species');
//-->false;
```

Prototypal Inheritence and Prototype Chain

we can call the calage method on all the objects of Person and it really improves the performance. So object of person and Prototype forms a chain ...That chain aint restrct out here..
Person.prototype is prototype of the persons;s objects so think about that the perosn.prototype is itself is a object and it maybe have it's own prototype ans yes it does have, because every obj in js has pototype.



it's nothing but a Object.prototype or a constructor function Object( )...This is prottype of our prottype Person.Prototype...



Object's prototype is pointing to Null so it indicate the end of the chain!!! it's asame concept as scope chain....it; look up in chain.

```
console.log(Saurabh.__proto__.__proto__);
//-->{constructor: ƒ, __defineGetter__: ƒ, __defineSetter__: ƒ,
hasOwnProperty: ƒ, __lookupGetter__: ƒ, …}

console.log(Saurabh.__proto__.__proto__.__proto__); //-->this is Null
console.log(Person.prototype.constructor); //It is nothing but a Function
Person!!!
// ƒ (fname,byear){
//     // console.log(this);

//     this.fname=fname;
//     this.byear=byear;
//     //this is bad practice , do not create a function inside of
constructor function




//     }




//console.log(Saurabh.__proto__.__proto__.__proto__.__proto__); -->This will
throe error\

arr=[1,2,3,4,45,4,4,4];

console.log(arr.__proto__===Array.prototype); //o/p->True

//we can add any method to Array.Prototype and customly define it and all
array can use it

Array.prototype.Uniqueee=function(){
    return [...new Set(this)];
}
console.log(arr.Uniqueee());

//this is not the way to do this stuff if js goin to introduce this method
in future then our code might get broked
```

ES6 Classses

```
//class expression and declaration

//expression
const Personcl=class{

}

//declarations
class PersonCL{
    //first add a constrctor it' works same as constructor functions

    constructor(fname,Byear){
        this.fname=fname;
        this.Byear=Byear;
    }

    //methods

    caculateage(){
        return Math.abs(1993-this.Byear);
    }
}

const jonas=new PersonCL('JOnas',1999);

console.log(jonas.caculateage());

console.log(jonas.__proto__===PersonCL.prototype);
//this will return true
```

Creatting a saprate prototypal mehod also goin to work in same fashion.
define a method outside of a class!!

```
PersonCL.prototype.greet=function(){
    return `Welcome, ${this.fname}`;
}
```

and jonas object can access this method because it has inherited from PersonCL.prototype

```
class PersonCL{
    //first add a constrctor it' works same as constructor functions

    constructor(fname,Byear){
        this.fname=fname;
```

```
        this.Byear=Byear;
    }

    //methods

    caculateage(){
        return Math.abs(1993-this.Byear);
    }

}


PersonCL.prototype.greet=function(){
    return `Welcome, ${this.fname}`;
}

const jonas=new PersonCL('JOnas',1999);

console.log(jonas.caculateage());
console.log(jonas.greet());
console.log(jonas.__proto__===PersonCL.prototype);
```

the things u should know about class

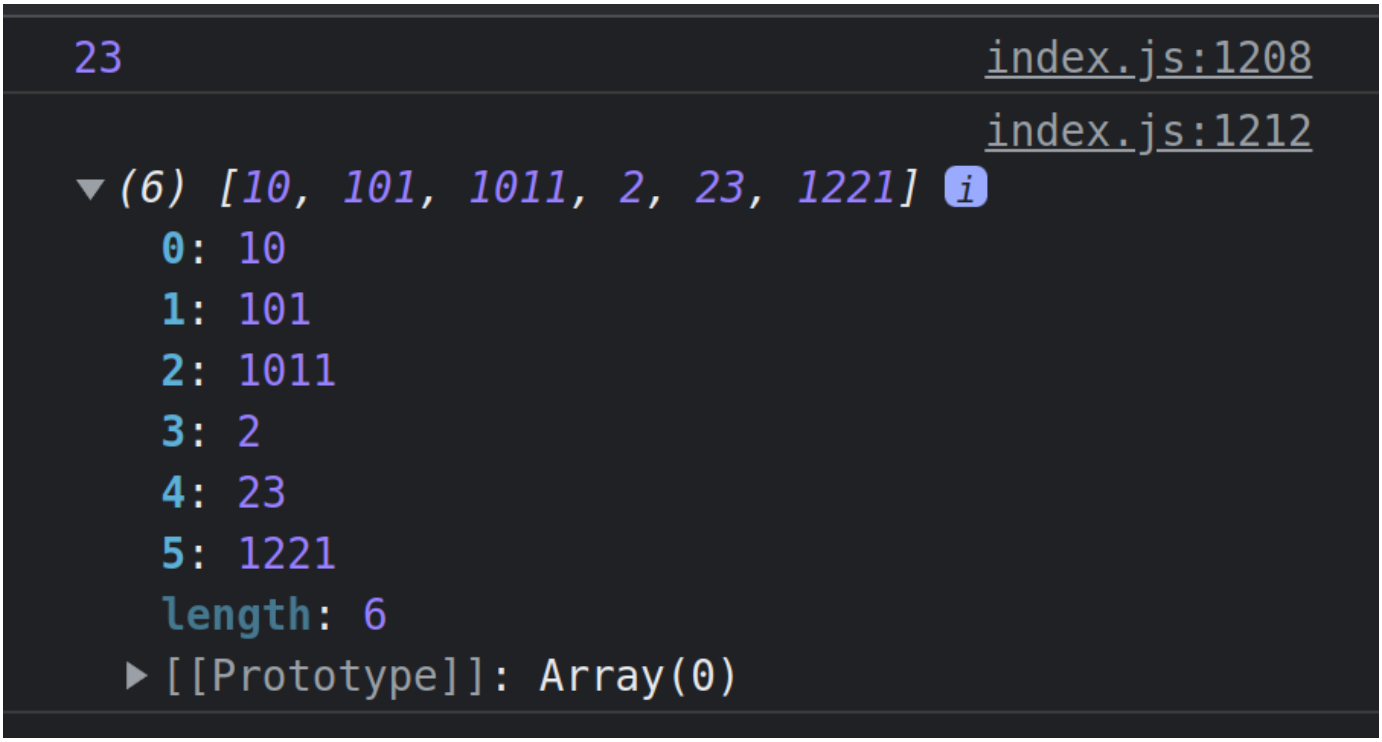
1. classes are not hoisted. (including class declaration and expression)
2.classes are special kind of function so they are first class Citizens
3.classses are executed in strict MODE.

<----------Getters and Setters---------->

//Getters and Setters-->

```
//getter setter here are properties,of an object not a functions to get
execute so dont call them expliicitely

const account={
    owner:'Jonas',
    movements:[10,101,1011,2,23],

    get latest(){
        return this.movements.slice(-1).pop();
    },
```

```
    set latest(mov){
        this.movements.push(mov);

    }

}

console.log(account.latest);
account.latest=1221;

//this is property not a function so u cant invoke it to set\

console.log(account.movements);
```



getter setter are actually useful for validation purpose...
naming confict arises actually when u r trying to change the properties inside the set method and also inside the constructors...This is an infinite loop to set the properties so always try to have different naming convention inside the setter and Contructors.

<----Static Methods----->
As MDN describes it, “Static methods are called without instantiating their class and are also not callable when the class is instantiated. Static methods are often used to create utility functions for an application.” In other words, static methods have no access to data stored in specific objects.\

Lets check it out what are those-->

```
class PersonCL{
    //first add a constrctor it' works same as constructor functions
```

```
    constructor(fname,Byear){
        this.fname=fname;
        this.Byear=Byear;
    }

    //methods

    caculateage(){
        return Math.abs(1993-this.Byear);
    }

}
```

this is our class and to create a static method on the class..

```
PersonCL.hey=function(){
    console.log("Hey");
}
```

This is our static method and we can say that it is a way to defined a statcic method on class directly not on it's prototype so, PersonCl has hey static method and the object aint able to access the static method because the defination is not on prototype but actuallly on class itself..

That's why we cant able to perform-->

```
[1,2,3,4].from( )

//this is not a valid way but the below one is

Array.from( )
```

anotther way to define a static methos is -->

```
class PersonCL{
    //first add a constrctor it' works same as constructor functions

    constructor(fname,Byear){
        this.fname=fname;
        this.Byear=Byear;
    }

    //methods
```

```
    caculateage(){
        return Math.abs(1993-this.Byear);
    }

    static get LetestOs(){
        return 'Ubuntu22.04';

    }

    static gain(){
        this.profit="60";

    }

}
```

Third way to IMPLEMENT Delegation or prototypal inheritence!

```
const personProto={
    calage(){
        console.log(2018-this.byear);
    },
};

const steven=Object.create(personProto);
console.log(steven);
steven.name='Steven';
steven.byear=1999;
steven.calage();

//here we ha\ve implemented prototype with totally different way
```

we can set the prototype manually to any object we want. here, we have manually set the property of steven object to personProto....

before doing the reassignment keep the steven type let rather than const...

```
console.log(steven.__proto__===personProto); //this will return true

//here we have changed manualy the prototype of stven so
steven=Object.create(ppr);
```

```
console.log(steven.__proto__===personProto); //this will goin to return false
```

Inheritence between the classes --constructor function

```
const person=function(fname,byear){
    this.fname=fname;
    this.byear=byear;
}

person.prototype.calage=function(){
    console.log(2018-this.byear);

}
const student=function(fname,byear,course){
    // this.fname=fname;
    // this.byear=byear;
   //person(fname,byear); this is a regular funtion call and in regular function call this keyword is set to undefied so it'll print name as undefinee
   //so fiirst Manually set the keyword
   const P=person.bind(this,fname,byear);
   P();
   //or simple user call
   //person.call(this,fname,byear); //this will goint to work, Apply is outdated

   console.log(P);

    this.course=course;
}


student.prototype.introduce=function(){
    console.log(`my name is ${this.fname} and course is ${this.course}`);

}

const mike=new student('Mike',2020,'computer science');
mike.introduce();
```

here the person and student are two constructor functions and u can see that we are setting the fname and year in person as well as in student s to avoid this we can directly call Person constructor inside the student but that is not the way because the Person( ) is a normal functional call and inside the

normal functtion call the this keyword is set tto undefined....Here we have to manually change the context of this to the Object og Person, so we have use Bind or call , bind return a functionna and call aint , prefer call here ...we have set the context of this here and the e have pass the argumens like fname and year inside the call function as a arguments to Person constructor.
here we want Person to be parent and student is child and student can access methods and props of Person class.

why we cann't able to do-->

```
student.prototype=Person.prototype
```

this statement indirectly saying that the student property and the persons propery is thesame but that is not correct and that is not what we want.
thats why we need object.create to link the prototypes.

```
const person=function(fname,byear){
    this.fname=fname;
    this.byear=byear;

}

person.prototype.calage=function(){
    console.log(2018-this.byear);

}
const student=function(fname,byear,course){
    // this.fname=fname;
    // this.byear=byear;
   //person(fname,byear); this is a regular funtion call and in regular function call this keyword is set to undefied so it'll print name as undefinee
   //so fiirst Manually set the keyword
   const P=person.bind(this,fname,byear);
   P();
   //or simple user call
   //person.call(this,fname,byear); //this will goint to work, Apply is outdated

   console.log(P);

    this.course=course;
}

student.prototype=Object.create(person.prototype);
```

```
student.prototype.introduce=function(){
    console.log(`my name is ${this.fname} and course is ${this.course}`);

}

const mike=new student('Mike',2020,'computer science');
mike.introduce();
mike.calage();
console.log(mike.__proto__);
console.log(mike.__proto__.__proto__);
```

but the problem is now student.prototype.constructor is also pointing to the Person constructor...Change it's context by doint

student.prototype.constructor=student;

bby doint this nnow mike is no longer a prototype of Person .

for Encapsulation Just check out Iife Functions(Immediately invoked fnction expression.)

ES6 classes concept use extends keyword for inheritenece...
check out inheritence using object.create also...

using --> object.create( )

```
//using object.create implement a inheritence or complex prototypical chanin


const personProto={
    calage(){
        console.log(2033-this.byear);
    },

    init(fname,byear){
        this.fname=fname;
        this.byear=byear;
    }
}
```

```
const steven=Object.create(personProto);

const StudentProto=Object.create(personProto);

StudentProto.init=function(fname,byear,course){

    personProto.init.call(this,fname,byear);
    this.course=course;

}
StudentProto.introduce=function(){
    return 'I am objexyt'
}
const jay=Object.create(StudentProto);


jay.init('jay',2029,'cmpssce');
jay.introduce();
jay.calage();
```

<----------Encapsulation---------->

protected properties and methods

public interfaces or method that are accessible outside of a class we called them api.... some of the method we do not wanted to expose to the outside worls or outside of a class that we can called

encapsulation.

in the below code u we are doing nothing but faking the encapsulation!

```
class Account{
   constructor(owner,curr,pin){
    this.owner=owner;
    this.curr=curr;
    this.pin=pin;
    this.movements=[];
   }

   get getMovements(){
    return this.movements;
   }
   deposit(val){
    this.movements.push(val);
   }
   withdraw(val){
    this.deposit(-val);
   }

   approveLoan(val){
    return true;
   }

}

const a=new Account('saurabh','rupee',41002);
a.deposit(12);
a.deposit(90);
const arr=a.getMovements;
console.log(arr);
```

Truely private class field and metthods

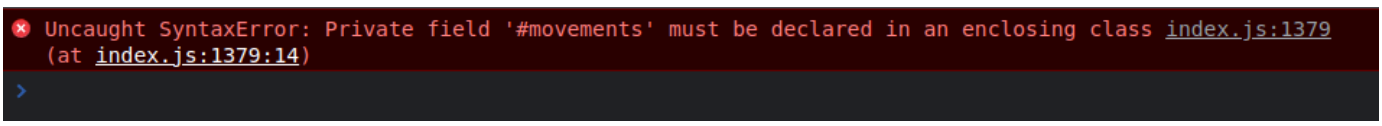
(It's not yet the part of javascript)'

why it called classfields ....because many languages the properties usually calles as fields...

```
class Account{

    //public fields (instances)
    locale=navigator.language;
```

```
    //_movements=[];

    //private fields
    #movements=[];
    //now this field is not accessible outside of the class try to access it through object.


   constructor(owner,curr,pin){
    this.owner=owner;
    this.curr=curr;
    this.pin=pin;
    //this.movements=[];
   }

   get getMovements(){
    return this.#movements;
   }
   deposit(val){
    this.#movements.push(val);
   }
   withdraw(val){
    this.deposit(-val);
   }

   approveLoan(val){
    return true;
   }

}

const a=new Account('saurabh','rupee',41002);
a.deposit(12);
a.deposit(90);
const arr=a.getMovements;
console.log(arr);
console.log(a.#movements);
```

When the user is trying to access the private fileld-->



private method aint implemented yet but with# method inside a class the class or js take it as a field.

```
#approveLoan(val){
        return true
        }

console.log(instance.#approveLoan);
//this will return the error
```

we can use stactic methods they aint available on instances but using class we can use them easily.