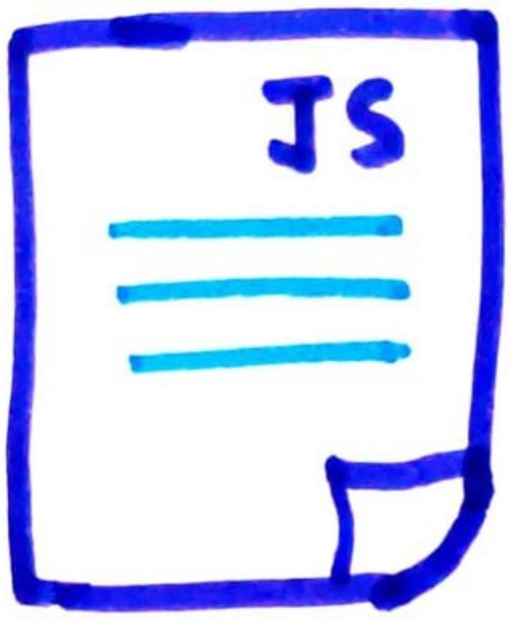
JavaScript Engine



Hey, I'm JavaScript Can you help me run

Did someone say anything? I don't understand!

Okay... So the browser doesn't understand JavaScript.

What it understands is bits (1's and 0's)

Who can help us here?

Yes!! The Javascript Engine

codeWithSimran

codeWithSimran_

There are a lot of JavaScript Engines out there written by really smart people!

For example :- V8 engine is written in C++ (yes they're programmed too and can be in a different language)

Okay, so what's inside this JavaScript Engine?

JE

Memory heap

This is where all the memory gets allocated
e.g. var a=5;
memory allocated to variable a

Call Stack

This is where your program executes. It keeps track of where we are in the code

So ever heard of a memory leak?

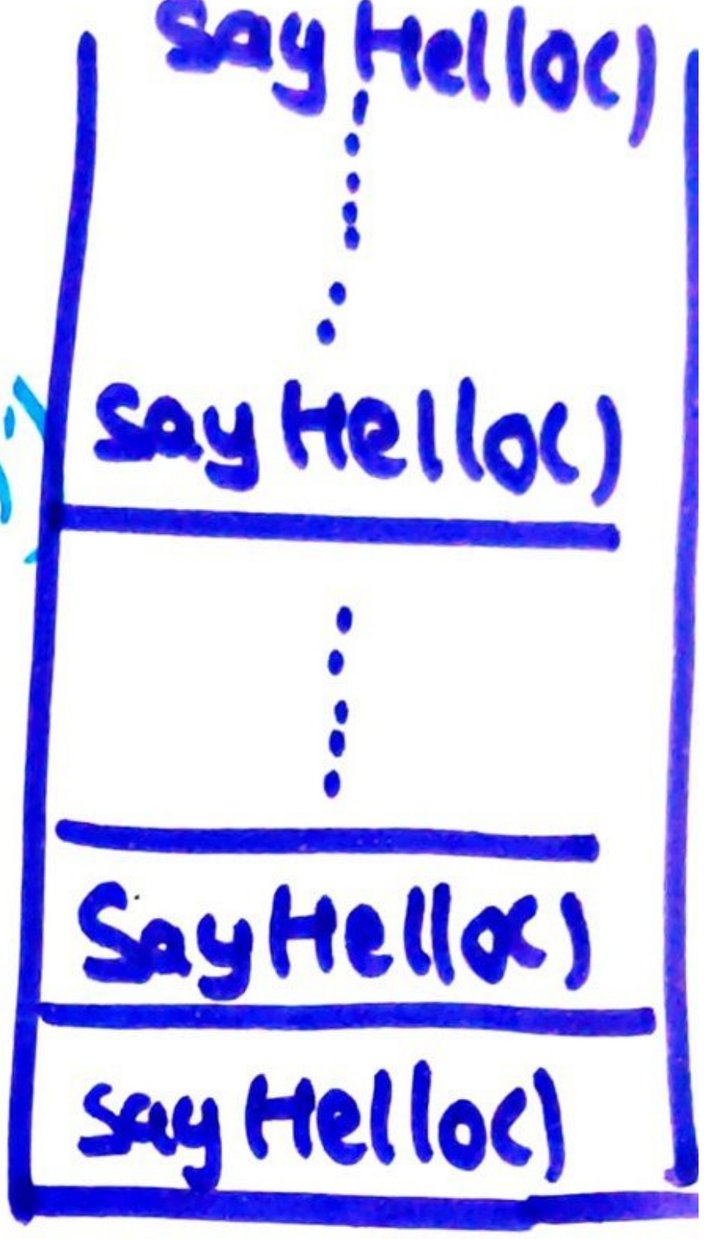
A memory heap has limited space. When you have too much of unused memory that you dont free up the spac gets filled.

No wonder, global variables are bad (They remain throughout the execution of the code)

You must've heard of stack overflow!!

Well that's when your call Stack overflows as it also has limited spac.

```
function SayHello()
{
    console.log('Simran');
    SayHello();
}
```



well that went into an infinite recursion and we have stack overflow

JavaScript is a single threaded language?

Well that means it has only ONE CALL STACK and therefore it can only execute one task at a time

Okay But Why single threaded!

It's quite easy and no complications

codeWithSimran codeWithSimran_

Okay... Wait! I've heard of asynchronous programming. If JavaScript can do that, how is it single threaded?

Let's take an example!

```
setTimeout ( () => {
        console.log("setTimeout is asyn"
    }, 2000)
```

→ wait for 2 seconds

setTimeOut is given to us by Web APIs (It gives us various APIs) It's technically not a part of Javascript.

codeWithSimran

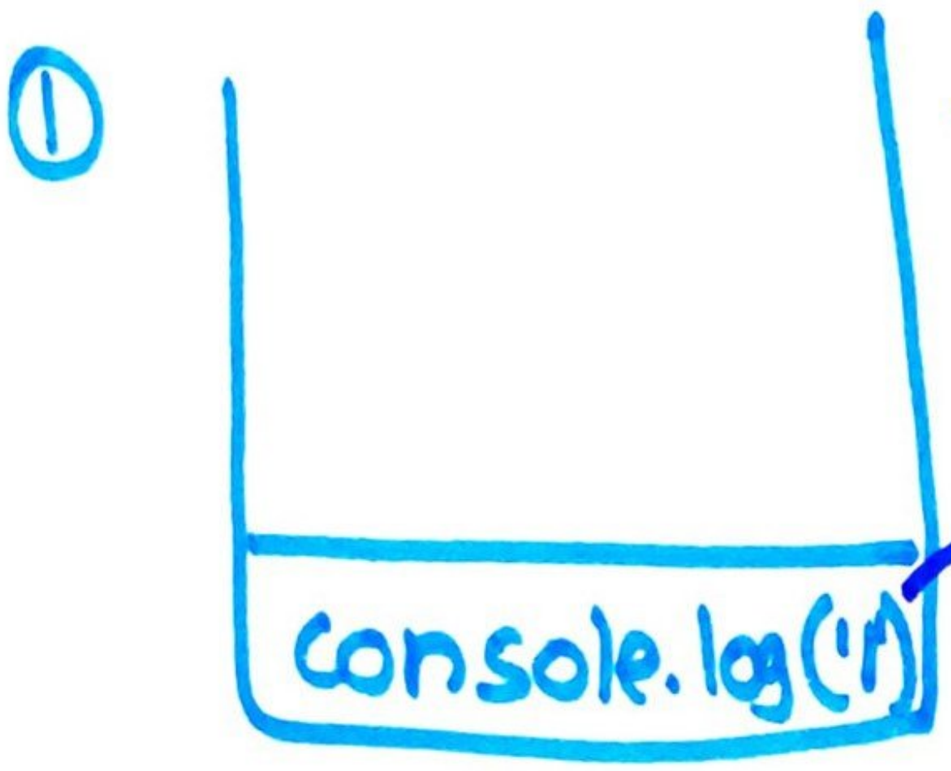
codeWithSimran_

```
console.log('1')
setTimeout(() => {
    console.log('2');
},
console.log('3');
```

Output : 1
3
2

[since setTimeOut waits for 2 secs it's printed in the end]

Behind the scenes



→ Call Stack

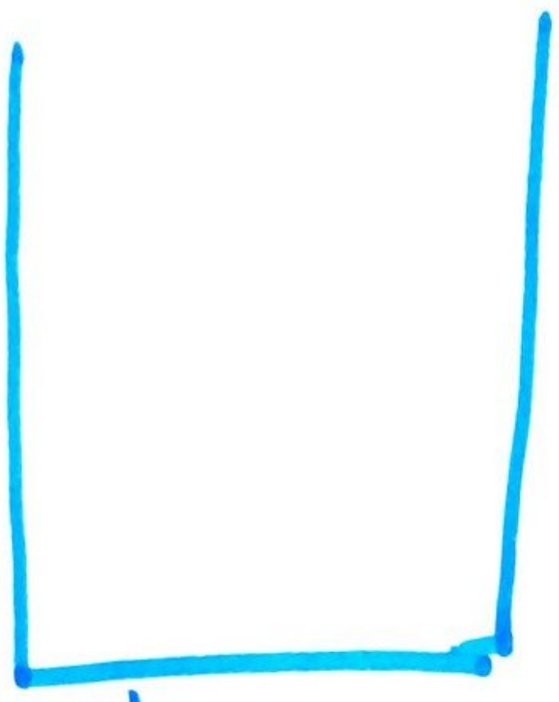
→ gets executed and gets popped so the stack is now empty and output is 1

codeWithSimran codeWithSimran_

(2)

OH! this is given by the web API Let me send it to web API

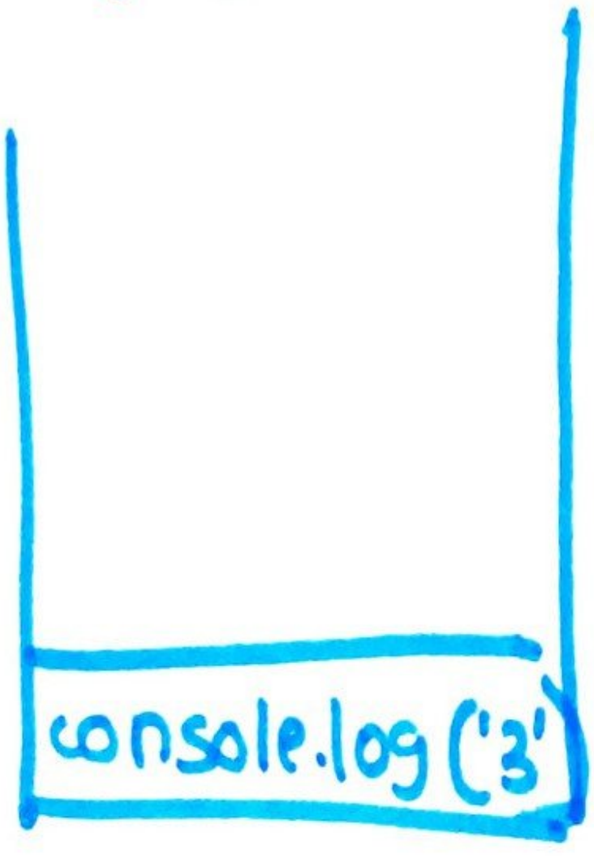
(3)



Web API

I've setTimeout with me and I should execute it after 2 seconds

(4)



> Output will print 3

So far

1

3

Web API is still waiting

(5) After 2 seconds are over WEB API ← On its console.log(2 that should be executed. This is basically a callback that is executed after 2 secs.

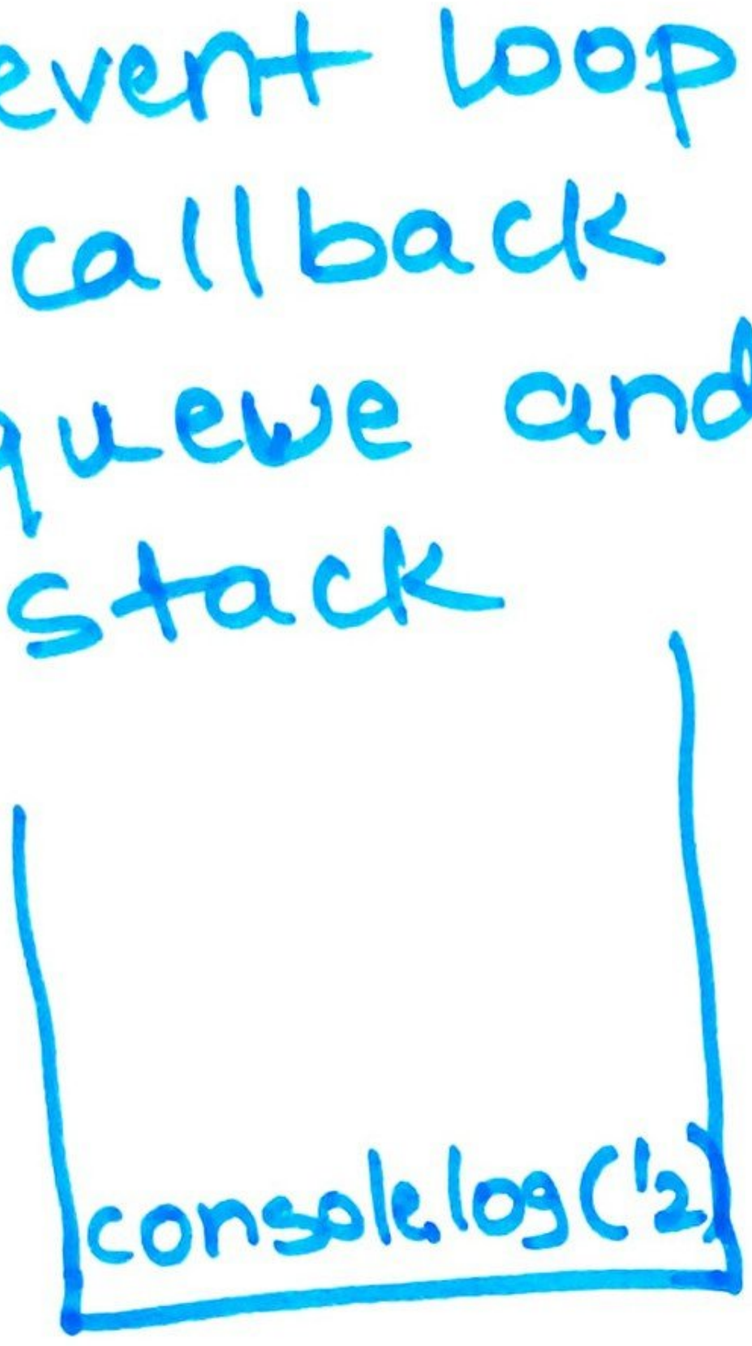
WEB API will send this to callback Queue saying there's a callback please proceed.

callback queue

This queue basically keeps track of all callbacks that need to be executed.

Now, there's something called as event loops which keeps checking if stack is empty

Well now it's empty so the event loop will take a callback from callback quewe and put it in the stack



> prints 2

so finally we have

1
3
2

Recap of setTimeOut

① Pushed to stack → ② Passed to WEB API

④ Pushes callback ← ③ waits for 2 seconds

to callback queue → ⑤ Event Loop ckeck if stack empty and pushes to stack

Execution Context

```
function getMessage()
 {
    return 'Hi Simran';
 }

function SayHello()
 {
    return getMessage()
 }
sayHello();
```

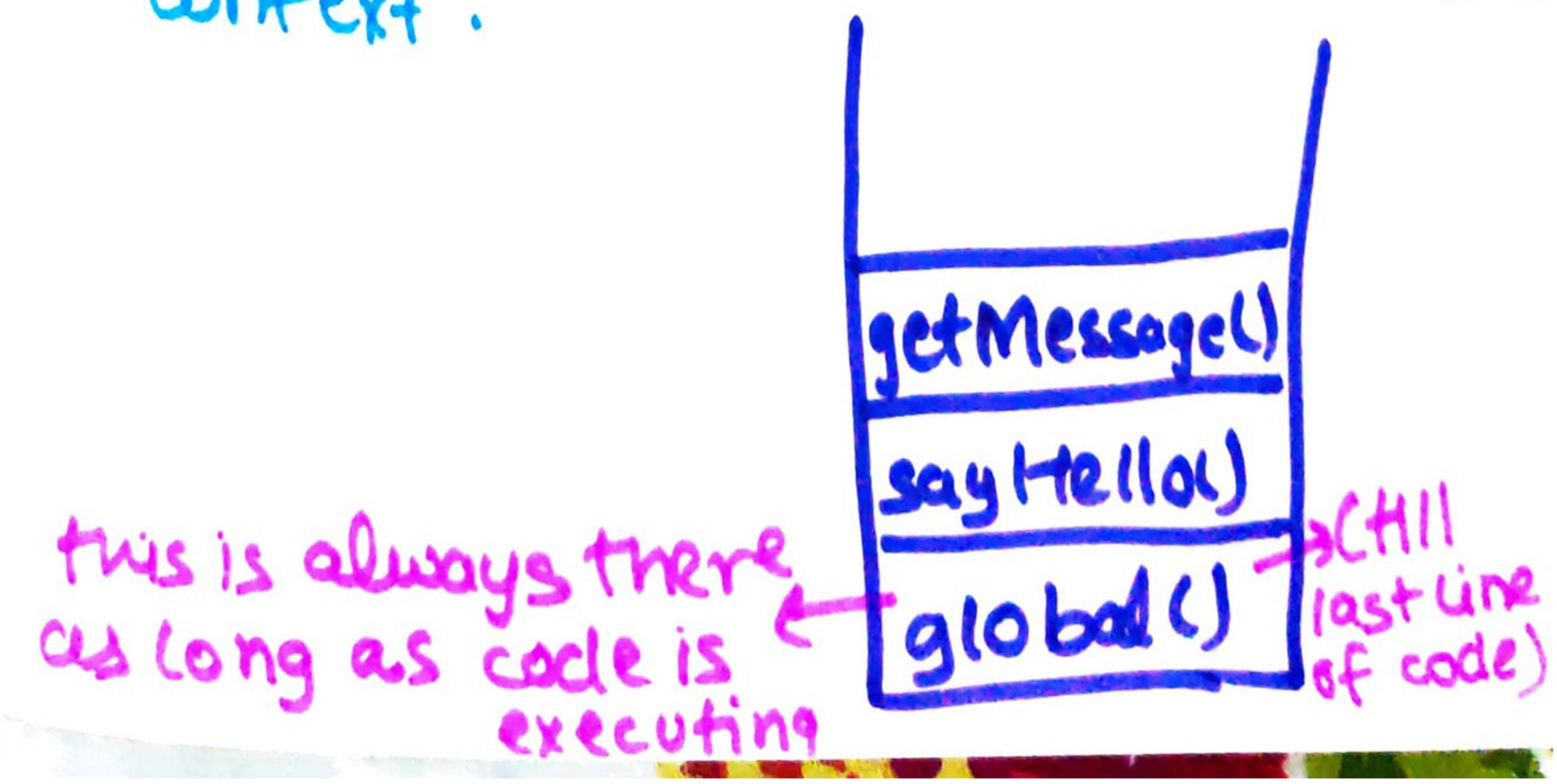
↓

Javascript Engine (JE) sees these brackets and says 'OH I need to execute this function, let me create an execution context for it.

Okay... What is execution context ???

BUT there is a base execution context (global context)

So basically the JE will always first create the global execution context.



codeWithSimran codeWithSimran_

That means every line of code is part of an execution context (eg. global), ora function)

So what do I do with this global Execution context?

↳ It gives us 2 things

Global execution context

- Global Object (which is the window object)
- this keyword in JS (the one that everyone loves:)

GO AND CREATE AN EMPTY JS FILE

Go to console and type

> this

→ Window object

So this prints the global object. In a browser global object is nothing but window object.

SO CHECK

```
this === window // true
```

Are they always going to be same? :/

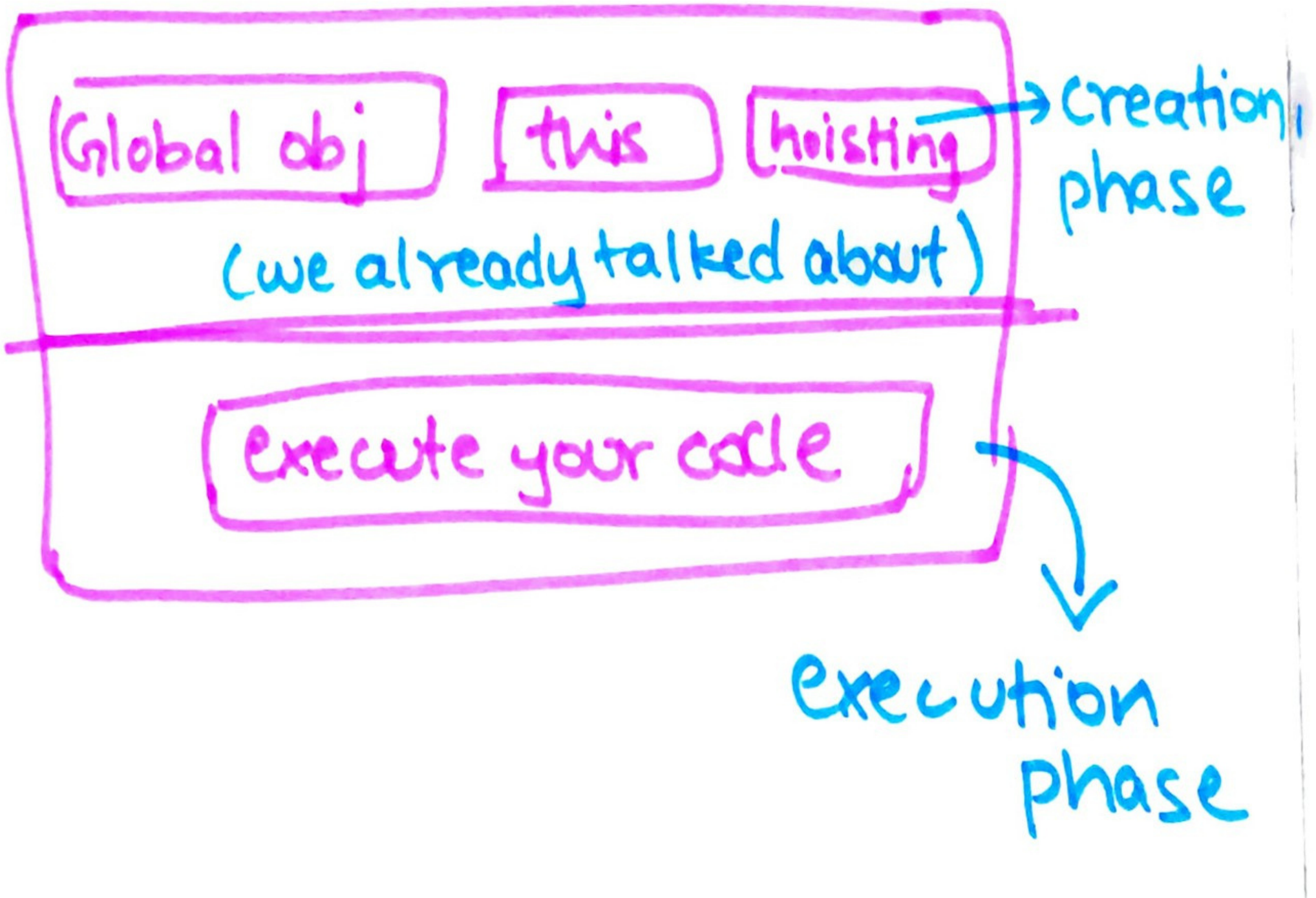
↳ NO !!! We have a whole new section to understand this keyword. Hold on :D

GO TO CONSOLE AND DECLARE A VARIABLE

```
> var name = "Simran"
> window
{
  name : 'Simran'
}
```

→ try to see window [window object will have name variable]

There are 2 phases of an execution context



Creation phase → global obj (window)
(happens first) this
hoisting

execution phase → run code
(happens next)

What is hoisting?

@codeWithSimran

Ex 1

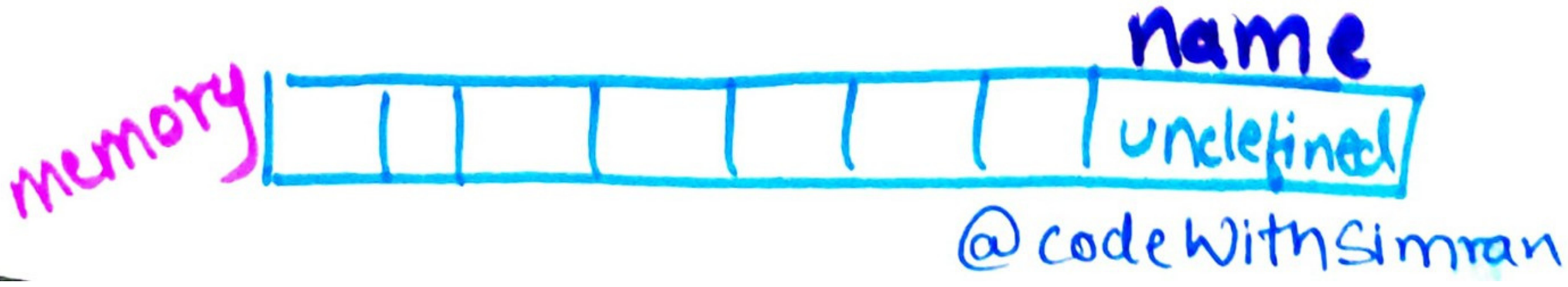
```
console.log(name)
var name = 'Simran'
```

> undefined

Why undefined? Should'nt it say name 'is ~~no~~ dosen't exist (Reference error)

It's because of hoisting!!!

During the creation phase JE sees this variable (name) and allocates memory for it and keeps undefined as its value.



@codeWithSimran

So since hoisting happens in creation phase there is already a variable named name in the memory with value undefind. @codeWithSimran

So during execution phase when it runs the file from top to bottom thus is what it sees [not physically we'll see that in a while]

var name = undefinded

sonsole.log(name)

name = 'Simran')

this is hoisting

note this is an assignment and does not have var

@codeWithSimran

codeWithSimran codeWithSimran_

NOTE: Variables are partially hoisted

@codeWithSimran

Meaning: Jhey are allocated memory but not assigned actual value what we give. They're simply assigned undefined.

Example 2

```
var name = 'Simran'
var name = 'Simran'
```

What about this? How many times will name be hoisted?

ONCE!! Why! Since the value dosent matter, JE will say I already have hoisted name.

Example 3


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

> Hello

Just like variables, functions are also hoisted.

BUT!! Functions are completely hoisted.
Meaning: The entire function with defination is hoisted.
So it wont give us undefined.

Example 4

@codeWithSimran

Just a clarification

JE doesn't physically move variables and functions to top of file. It just allocated memory for them & before ✦ running the code.

```
sayHello();  → function expression

var sayHello = function(){
    console.log('Hello')
}
> undefined
```

It was treated like a var and given undefined as it doesn't care about the value

Examples

```
sayHello();
```

@codeWithSimran

~~sayHel~~

```
function sayHello(){
    console.log("Hello");
}
function sayHello(){
    console.log("Bollo");
}
```

> Bollo

Since functions are completely hoisted, when same function comes 2nd time, JE will say I've already allocated memory for sayHello, let me replace its content from console.log("Hello") to console.log("Bollo")

Example 6

@codeWithSimran

TRICKY ONE

Just remember when a function is called a new execution context is created for that function.

```
var name = 'Simran'
var changeName = function(){
    console.log("Name", name);
    Var name = " John";
    console.log ("Changed name",
                    name);
};
changeName();
```

@codeWithSimran

what you probably expect

> Name Simran

> Changed Name John

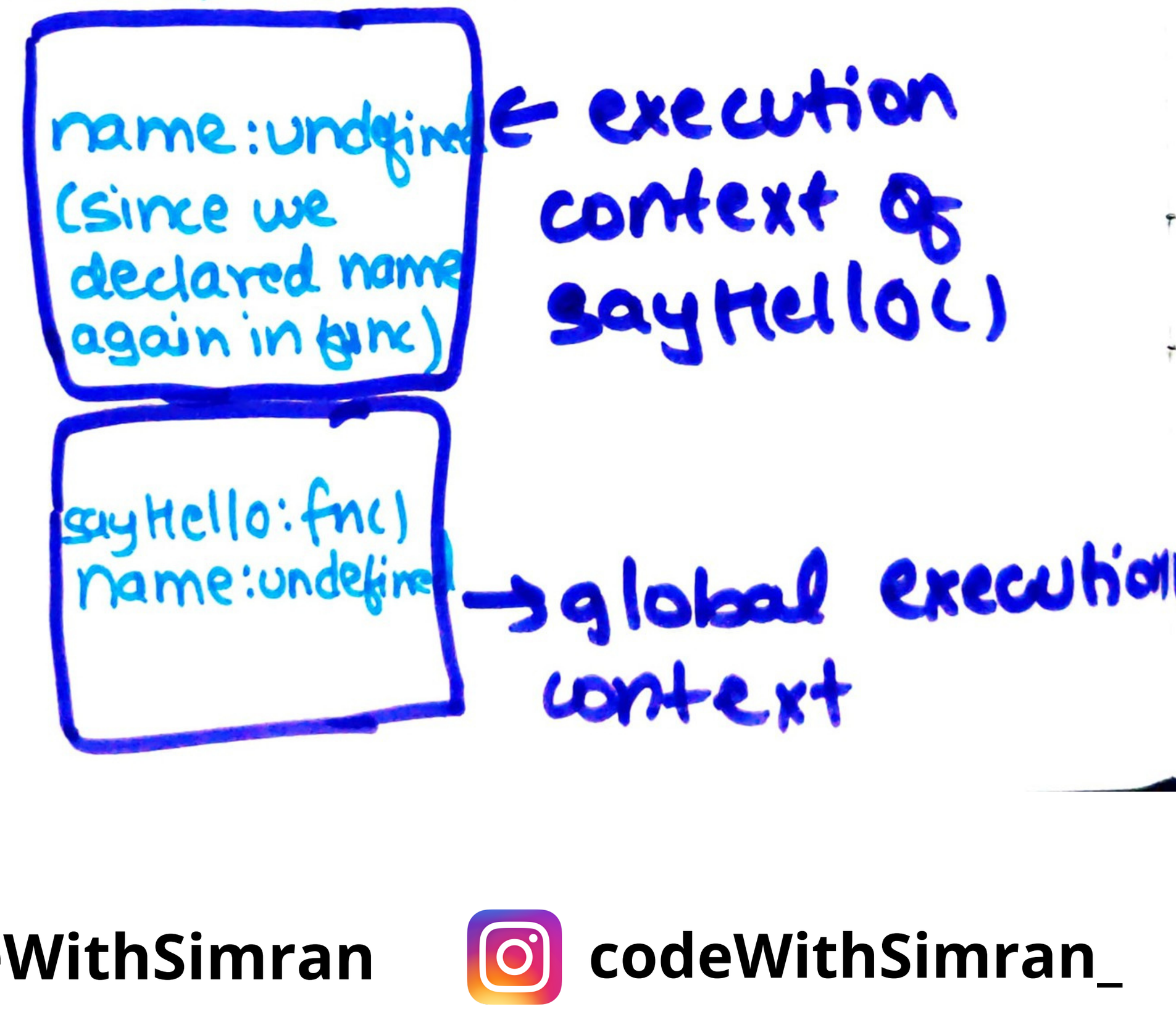
Wrong ✗

> Name undefined
> changed Name John ✓

@codewithSimran

What ?? Why?

When you called a new execution context was created on top of global execution context.
This execution context is created for the function when it is called



Go and look at the problem again, you'll get it. :)

BUT STILL, JS is tricky and weird :/

Oh! guess why they introduced let and const :)

They are [redacted] hoisted !!!

Even though they're hoisted they're not assigned anything (not even undefined) and we get reference error if we try to access them before using them

Arguments keyword.

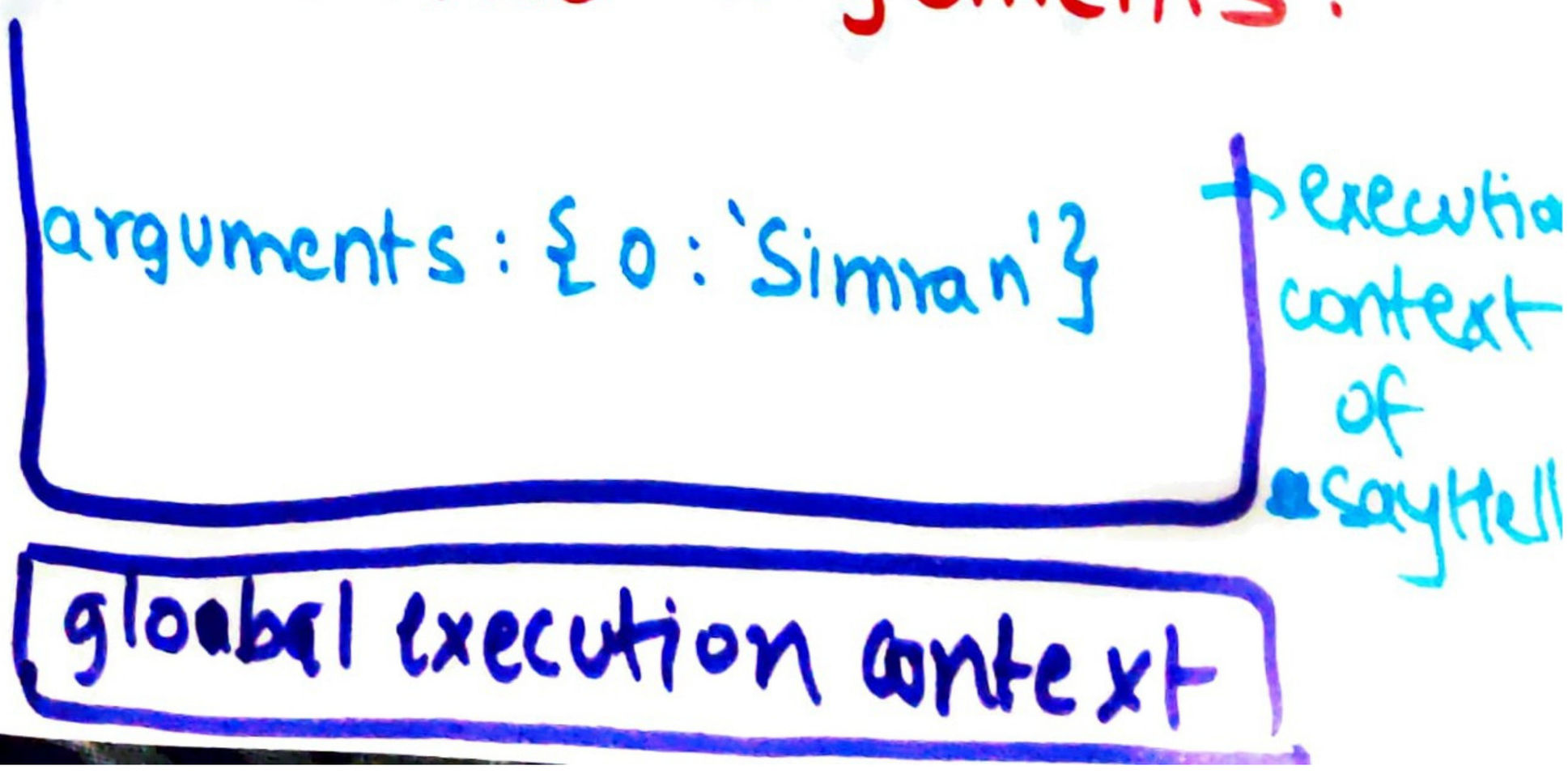
```
function sayHello(name)
 {
     console.log("Hello" + name);
 }
```

@codewithSimran

```
sayHello('Simran');
```

We already know calling a function creates a separate execution context for it.

If you see the execution context of sayHello you will seean object named arguments.



if we passed 2 arguments
we'd get

@codewithSimran

arguments: {0: 'first argument',
1: 'second argument'}

So arguments is an object,
it's not an array!
All the keys of arguments
object are 1, 2, 3....

How do we iterate the
arguments object?

You can use the new Array.from
what does Array.from do?

Let's say our arguments object
is {0: 'Simran', 1: 'Tina',
2: 'John'} @codewithsimran

> Array.from(arguments)

→ ['Simran', 'Tina', 'John']

It returns us an array of all values :)

We can now use any array methods on it.

@codeWithSimran

OR

we can use the spredd operator (any name but not arguments)

```
function sayHello(...args)
{
  ....
}
```

Since its reserved keyword

Now we can do

args[0], args[1] and so on

@codeWithSimran

codeWithSimran codeWithSimran_

What happens when you don't pass any arguments to a function

You will still get arguments object but it will be empty {}

@codewithSimran

Summary

→ There can be multiple execution context,

→ An execution context is created every time we call a function

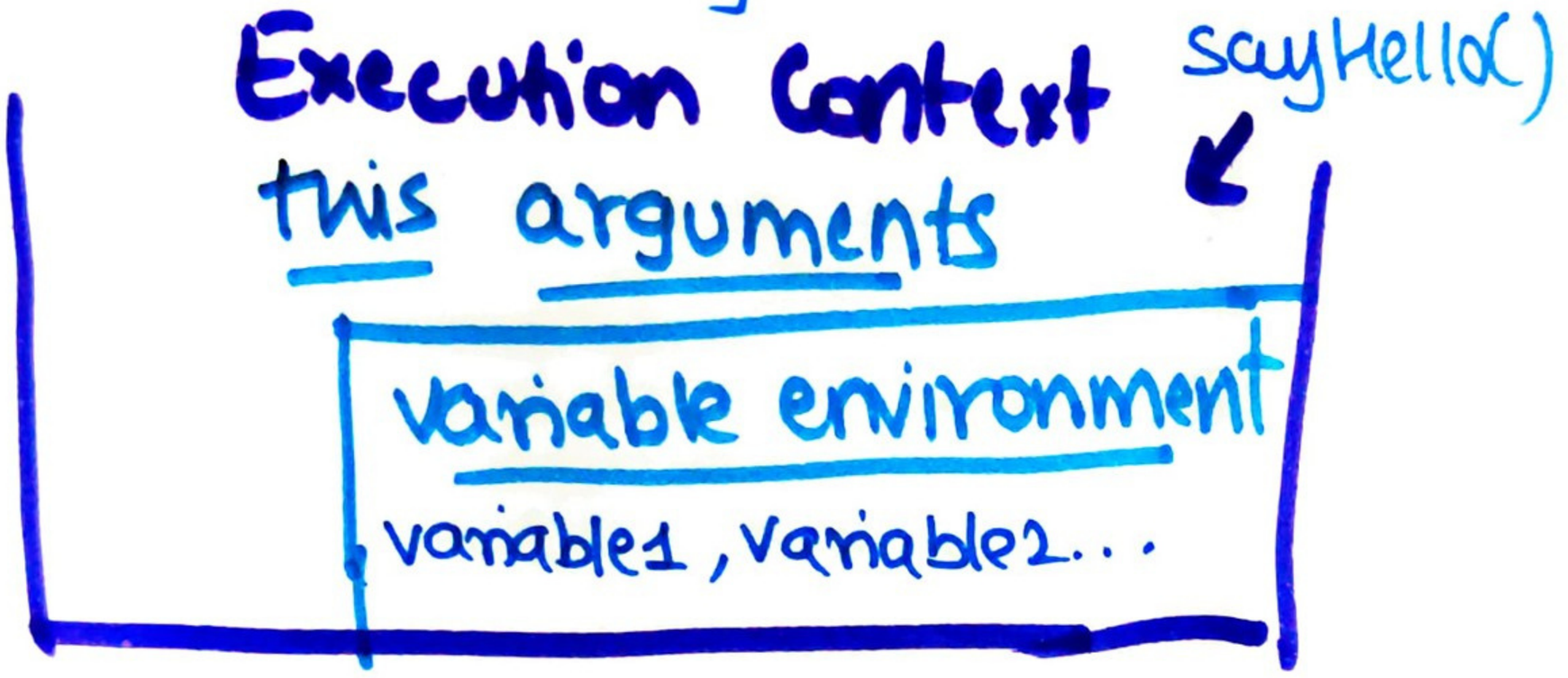
→ Execution context of every functions has an argument object

codeWithSimran

codeWithSimran_

Variable environment

whenever we create an execution context (like calling a function) we so far know we get this and arguments object. There's one more thing @codeWithSimran



Since variables declared inside a function are local to that function, they reside in the variable environment or local environment.

* Once the function stops executing or is done executing its execution context is removed. Therefore so is the variable environment. So we can't have access to variables declared inside a function when it is done executing

Scope Chain

@codewithSimran

→ Each execution context has a link to it's parent.

→ The parent is decided by where this function is lexcially (where is it in the code)

```
var name = 'Simrah'
 function sayHello()
  {
    :
  }
 function SayBye()
  {
    :
  }
```

@codewithSimran

→ Both sayHello and sayBye have access to the variable name

[All functions have accets to global scope] **

So what data a function has access to depends on where the function was defined and not where it was called.

@codewithSimran

So JE already decides at compile time what functions will have access to which variables because it knows where a function is defined (when it scans the file) but it dosent care about where the function is called.

Go to console define a function

>window → (this show have your function, sayHello) for example

sayHello:: → [[Scope]] → [sayHello will have [[Scope]] and it will tell you its scope in this case global)

Exercise

```
function leakage() {
    name = 'Simran'
}
```

Where is name in the execution context?

It should be in the execution context of leakage function right? BUT, it's not. Since we didnt declare name with var, let, const etc, leakage function say 'Hey I dont have the variable name and passes on to global context, global context says I dont have it either :/. So the JE creates a variable in the global scope.

This is called leakage of global variables

Ever heard of 'use strict'?

When you write 'use strict' on top of your file, it dosent let you create variable without actually declaring them (using var, let or const)

Okay.... 'use strict' is a nice friend. Dosent let you go in the wrong direction :D

codeWithSimran

codeWithSimran_

Function Scope V/s Block Scope

We already know about function scope. Everytime we call a function we create a function scope, and we can only access variables declared inside a function inside it. We can't access them outside the function. Because they're within the function scople

So what's block scope?

Block scope is the scope created inside {...} (curly brackets)
JavaScript follows ~~block~~ ~~so~~ function scople. And didnt actually have the concept of block scope until ES6 was introduced.

@codeWithSimran

```
if(true)
{
  var name = 'Simran'
}
console.log(name)
```

> Simran

Because if is not a function right? so name is still in global scope and is accessible.

So if we had

```
function sayName()
{
  var name = 'Simran'
}
console.log(name)
```

> Reference error

Explanation
Basically when name is declared inside function a function scope is created and that's why global scope does not have name it's in function scope)

However, with the introduction of let and const, they also come under block scope.

@codeWithSimran

```
if(true)
{ let name = 'Simran' }
console.log(name)
```

> ~~Simran~~ Reference Error

Because since let and const are block scoped using if statement is like block scope ({...}) there those variables are only accessible inside the block scope.

** Var still follows function scope.

Let and const follow block scope.

→ Therefore in most situations it's better to avoid & var

Immediatly invoked function expressions ~~EE~~ (IIFE)

Before that let's understand why global variables are bad and then how IIFE can help us :)

-> we already spoke about memory leaks

@codeWithSimran

Okay, if space is the problem, can I use a few global variables

Let's say we have included multiple script tags in a file

<script> var count = 1 </script>
<script> var count = 2 </script>

* Both files have a global variable named "count" so now your count will be overwritten by the lastest count value which is 2

THAT'S BAD RIGHT?

As your codebase gets larger it'll get hard to track these name collisions.

* All script tags get combined to one execution context so count 'is used across all and obviosly it can't have different value for different files.

@codeWithSimran

IIFE will help us here

IIFE is a function expresion that woks like this

This brack says this is not a function delcation, its a function expression

```
( function () {
})();
```

→ anonymous func (no name)

→ we are immediatly calling this function

Summary

It's an function expression (we already learnt function expresions don't get completely voisted as JE does not look at them as function declaration) That means whatever code we put inside this function will not be a part of global execution. context since this function itself is not part of global ex

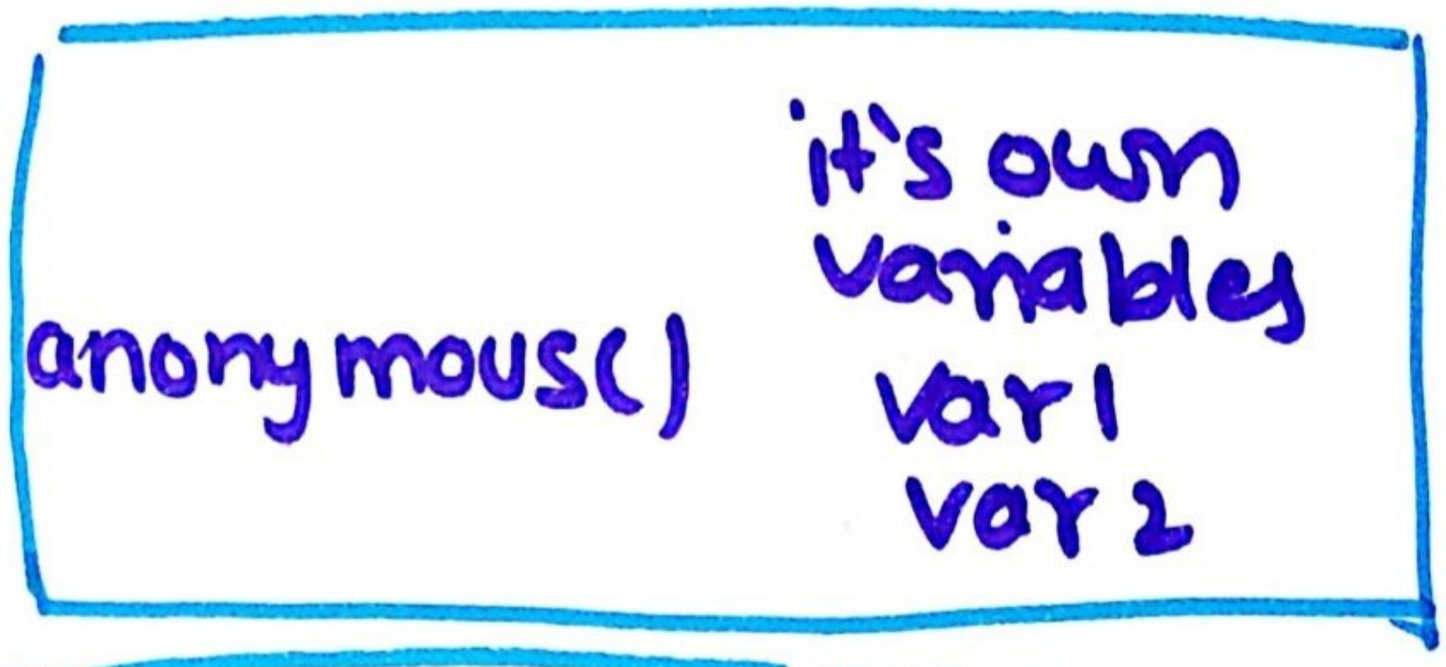
@codeWithSimran

codeWithSimran codeWithSimran_

So far we know whatever code (variables) we declare inside this IIFE are going to be ~~global~~ local to this function.

@codewithSimran

This is how the execution context looks



So a new execution context is created for this anonymous function and it has it's own local variables

@codeWithSimran

How doe we use IIFE to use data (variables or functions) throughout the file?

file is @codewithSimran

```
var file1 = (function() {
    var name = 'Simran'
    function sayHello() {
        return 'Simran'
    }
    return {
        name: name,
        sayHello: SayHello
    }
})()
```

** Since we're 'immediatly invokiy 'calling this IIFE file1 variable has the returned data or return value of this IIFE so we can simply do

file1.name

file1.sayHello() etc

But again, we ~~s###~~ still have a global variable called file1 so there is a better solution in modern Javascript called <u>modules</u> we will study about it later.

@codewithSimran

THIS KEYWORD

This refers to the object on which we call our function:

Example

② sayHello has 'this' (just like all other functions) and 'this's value is obj

```
obj.sayHello( this )
```

① obj is the object on which we're calling sayHello

@codeWithSimran

* Inside of a function the value of 'this' is the object we used to call the function.

What if we simply say
> sayHello(), we're not calling sayHello on any object.
> In this case, 'this' refers to the window object.

* So just saying sayHello() is like saying window.sayHello()

① Look before the dot what's there?

@codeWithSimran

SomeObject . SomeFunction()

② It's someObject → the value of 'this'

③ If there is not . means directly calling function, then look at it as

window . someFunction()

what's b4 before the dot is window internally.

@codeWithSimran

* On global execution context the value of 'this' is window

Example 1

```
function sayHello() {
    console.log(this)
}
sayHello()
```

> window [Because sayHello is called without any object, iternally it's window (window.sayHello())

Example 2

@codeWithSimran

```
const obj = {
    name : 'Simran'
    sayHello : function(){
        return 'Hi' + this.name
    }
}
```

If we want access to name in sayHello this is how we can do it. Since sayHello is a part of obj, that value of 'this' inside sayHello is 'obj'

To run it we will use

obj.sayHello()

And what's before that.? It's 'obj' ,that means value of 'this' inside sayHello should be 'obj'

Example 3

@codeWithSimran

* Execute same code for multiple object (Reuse)

```
function sayHello() {
        console.log(this.name)
}

const obj1 = {
        name : 'Simran
        sayHello : sayHello
    }
const obj2 = {
            name : 'John'
            sayHello : sayHello
    }
```

```
const name = 'Lara';
```

> window.sayHello() OR sayHello()
→ Lara

> obj1.sayHello()
→ Simran

> obj2.sayHello()
→ John

@codeWithSimran

Conclusion

* This gives methods (functions inside your object) access to the object they're defined in
* Let's us execute same code (sayHello) by multiple objects (obj1 and obj2)

Example 4

```
Const a = function() {
    console.log('a', this)
    const b = function() {
        console.log('b', this)
        const c = {
            sayHello : function() {
                console.log('c', this)
            }}
        c.sayHello()
    }
    b()
}
a()
```


- a → Window
  (because there is nothing before a()
  its effectively window.a())
- b → Window
  (same case as a)

Note: value of 'this' inside b is window means it DOES NOT MATTERS WHERE WE DEFINED FUNCTION B, what matters is how it's called

But Javascript follows lexical (static) scoping right? Scope of a function is decided by where it is defined and not where it's called. @codeWithSimran

No wonder 'this' is like an alien, that follows dynamic scope, value of this depends on how we called the function not where it is actually defined.

▶ c → c 'this'
sayHello

Inside ▲ the value of 'e is the object c because we called sayHello as

c.sayHello()

@codeWithSimran

Example 5

```
const obj = {
    name: 'Simran'
    sayHello: sayHello()
        console.log('Hello', this);
```

Example 5.

```
const obj = {
    name: 'Simran'
    sayHello: function() {
        console.log('Hello', this)
        var sayBye = function() {
            console.log('Bye', this)
        }
        sayBye()
}
```

> obj.sayHello()

- Hello obj
- Bye Window

So how do we solve this problem and make it work as lexically scoped, value of this should depend on where the function was defined because that's how rest of javascript works

Soln Arrow functions 

if you convert sayBye to an arrow function out will be

- Hello obj
- Bye obj

Why? Because inside arrow function value of this depends on where that arrow function is present or defined And its defined inside obj right? So value of this will be obj

Wait!! Arrow functions were introduced in ES6 right? What did people do before that?

* To get the same result with normal function as arrow function you can use the bind method

@codeWithSimran

~~sayBye()~~

sayBye.bind(this);

↓

you're basically binding the current value of this which is (obj) to sayBye as well.

In order to play with 'this' keyword, we have 3 methods.
call(), apply(), bind()

* Internally every function uses call() when it's invoked.

→ so when you call a function by sayHello()
you're basically doing
sayHello.call() [Internally]

Let's take an example of a game where you've a battery life (of player) and in some situations the player can charge itself to 100%.

```
const player1 = {
        name: 'John'
        battery: 62
        charge: function(){
                this.battery = 100
        }
}
```

@codeWithSimran

So basically player1 is an object with a method charge (to charge 100%.)

To simply charge Player 1
we can do

→ player1.charge()
[battery: 100]

Now lefs introduce a second
player, player2

const player2 = {
name: 'Lara'
battery: 20
}

OKAY, player2 does not have
a charge method, so can it
borrow the same from player1
Also not we dont want to
repeat code that can be reused

→ call the charge method of player1

player1.charge.call(player2)

But call it on player2

We're esentially able to pass reference to player2 (which will be value of this) inside battery method.

So argument to call is overding what the value of 'this' will be when that method is 'invoked.

*Also you can pass arguments using call which will be recieved by battery method.

player1.charge.call(player2, 20,30,40)

arguments

and your battery function might look like this

battery: function (arg1, arg2, arg3)

NOTE: You don't need to accept player2 as an argument. 'this' already contains player2.

apply

It's pretty much same as call but the way it takes argument is different.

It takes an array of arguments

```
player1.charge.apply(player2,[20,30,40])
```

takes an array of arguments

bind()

call and apply immediatly invoke the function, whereas bind returns a new function binded with a new `this` which we can run later

```
const callLater =
player1.charge.bind(player2,20,30,40)
[syntax same as call]
to actually call it we can later do
  callLater()
```

Functions are objects (JavaScript)

@codeWithSimran

```
function sayHello()
{
    console.log("Hey")
}

sayHello.property1 = 'Bye';
```

well if functions are objects we should be able to add properties to it right?

How functions look internally

```
const functionObj = {
  property1 : Bye
  name : sayHello
   () : console.log("Hey")
}
```

So, that basically means, every function is represented as an object. It has all the properties of that function for example property1 that we just defined It also has a name property that is the name of the function. (Optional since function can be anonymous. It has () which is basically used to invoke the function.

@codeWithSimran

Note: This example is just an illustration to understand, the object could look differently It's important to understand this because we say functions can be passed as arguments (They're objects)

Types in JavaScript

* Primitive type
  → number (5, 6 ...)
  → boolean (true, false)
  → undefined (undefined)
  → null
  → Symbol (new in ES6)
  → String ('hi', 'hey'...)

* Non Primitive type
  → object
  → {}
  → [...]
  → function

@codeWithSimran

What's the difference?

→ Primitive type is directly stored in the memory (var a = 5) a holds the values.
→ Non-primitive type does not directly hold the value, it holds the reference to that value in memory (like pointers)

Built in Objects (come with the language)

we already have primitive types like number, boolean etc

So why are there built-in objects like **Number**, **Boolean** given by the language?

→ Every primitive type (number, string etc) have wrappers around them called Number(), String() etc.

Okay but why complicate?

> false.toString(); @codeWithSimran

→ 'false'

Wait!! false is a primitive type how can be run toString() on it?

What JS does internally

Boolean(false).toString()

Oh so these built in objects like Number, Boolean etc exist for us to run other methods on the primitive types. Because we cant directly run the method on primitive type.

So not everything in JS is an object, there are primitive types too that are not Objects

Let's now talk about non-primitive types (Objects)

* Arrays are objects
  Umm.... wait what?
  Internally arrays are treated like objects as follows

```
let array = [1,4,5]
```

internally

```
var array = {
  0 : 1
  1 : 4
  2 : 5
}
```

@codeWithSimran

codeWithSimran codeWithSimran_

So if you check
>typeof [1,2,3] @codeWithSimran
>object.

So how do we differentiate and find if its an array if typeof says its an object.

Array.isArray([1,2]) → true

↙

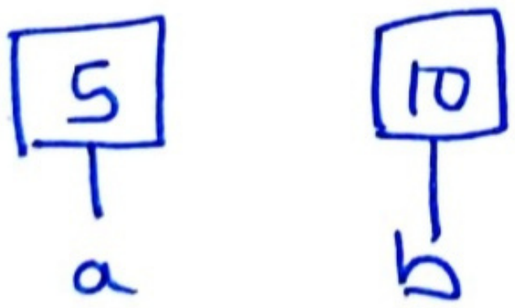
Built in Object
that has isArray property [modern JS]

Pass by value & Pass by reference

* Pass by value

let a = 5; let b = 10

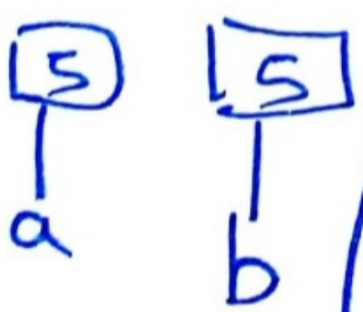
memory



That means a and b actually hold the values we assign them

* when we do

let a=5; let b=a;



a has no contact with b later
a copy of a is assigned to b and
b is now independent.
This is pass by value.

* Pass by reference

Objects are passed by reference

```
let obj1 = { name: 'John',
             favFood : 'Burger' }
let obj2 = obj1;   @codeWithSim
obj2.favFood = 'pizza';
console.log(obj1, obj2)
```

obj1
↳ { name: 'John', favFood : 'pizza' }

obj2
↳ {name: 'John', favFood: 'pizza'}

Wait, what? obj1 and obj2 are same but we only changed favFood property of obj2 right:

what happens internally

① obj1 → name: John, favFood: 'burger' ← obj2 ② obj2 = obj1

memory

codeWithSimran codeWithSimran_

That means obj1 and obj2 have the same data, because they're both pointing to the same memory location.
So changing obj1 or obj2 will change data for both.
So objects are basically references to a some memory location and obj2 = obj1 does not create a new memory space for obj2.
We're saving memory space 🙂
But also it's confusing, what if someone wants to keep them separate?

@codeWithSimran

Also remember, since arrays are objects, that have the same behaviour

Arrays soln	Object soln
let a = [1,2,3] let b = [].concat(a)	let a = {'a':0,'b':1} let b = Object.assign({},a)

Exercise:

Try doing the same with spread operator.

soln let b = {...a}

What if we have nested objects?

```
let obj1= {
        a : 1
        b : 2
        c : {
            nested : true
            }
    }
```


Try the above methods and do

obj2 = ~~obj1~~ {...obj1}

obj2.c.nested = false;

You'll se the value of nested will change to false in both objects.

Opps... We have a problem. Whatever approach we tried on top does not work with nested object because every object

is passed by reference and we only cloned the top layer

This is called <u>shallow cloning</u>.

But what we need now is

<u>deep cloning</u> (all levels)

→ let obj2 = JSON.parse(JSON.stringify(obj1));

This does deep cloning.
However if obj is too large, there will be performance issue as parsing the whole object to all levels can take time.

@codeWithSimran

codeWithSimran codeWithSimran_

Functions are first class citizens in JS

① We can assign functions to variables

```
var sayHello = function() { }
```

② We can pass functions as arguments

```
function sayHello(message) {
    message()
}
sayHello(function() { console.log("Hello") })
```

↳ accepting function as an argument

passing function as an argument

③ Return functions from another function

```
function sayHello() {
    return function message() { console.log("Hey") }
}
sayHello()();
```

① Has the return value of sayHello

② The returned value is a function and we want to call that function that is returned.

We're able to achieve all this because, well functions are just objects right? And we can pass objects to as arguments, return trem etc

@codeWithSimran

Higher order functions @codeWithSimran

A function that takes a function as an argument, returns a function, or both

Let's say we want to multiply 2 numbers, but before that we want to check if they're numbers

```
function multiply(num1 ,num2)
 {  if(num1 && num2 && typeof num1 === 'number'
                        && typeof num1 === 'number')
      {
         return actualMultiplication(num1,num2);
      }
}
```

@codeWithSimran

So basically instead of doing everything in mutiply we first check if arguments are valid and then call the multiplication method.

Now let's say we decide that we want to add both the numbers, will we write another function doing the same check? Can we pass add or multiply as arguments? [assume add & multipy exist]

```
function operation(num1, num2, performOperation)
 { if(checkIfValid(num1, num2)) {   // basically checks if valid
      performOperation(num1, num2)
 }

operation(5,2, multiply)   // we can now pass a func as argument to decide what
operation(5,2, add)        // we want to do at runtime
```

CLOSURES

@cocleWithSimran

Closures allow functions to access variables from the enclosing scope even after it leaves the scope in which it was declared.

Confusing right?

```
function first(){
        let breakfast = 'breakfast'
        return function second(){
                let lunch = 'lunch'
                let random = 'xyz'
                return function third() {
                        let dinner = 'dinner'
                        return '${breakfast}→${lunch}→
                                ${dinner}'
                }
        }
}
first()()();
```

@codeWithSimran

→ breakfast → lunch → dinner

① When we called first() → it returned second
→ first got pushed on the call stack and when it returned the result, it got popped off the stack

② We invoked the result of first() [which is a function] by first()() so second got invoked
→ Second was pushed on the call stack and returned function three. So second is popped off the stack

codeWithSimran codeWithSimran_

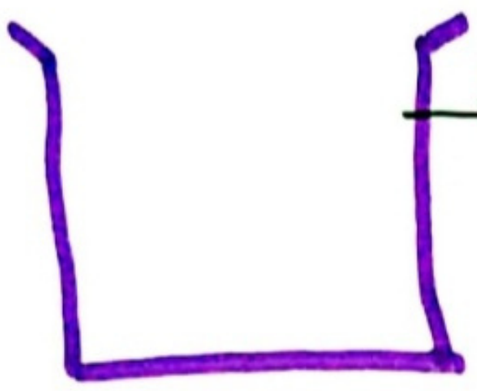
③ We finally invoked function three by first()()()

Function third printed breakfast, lunch and dinner

But wait!! How did function third get acces to breakfast and lunch when the functions owning them were popped off the stack?

Well, the magic lies in closures @codeWithSimran

Closures is actually a feature by Javascript



* Let's say we have a closure box somewhere, when JE sees that breakfast in accessed somewhere down the line, it puts <u>breakfast</u> into the closure box

* Even though first is done executing, it keeps variable breakfast in the closure box
* Similarly if sees lunch is being accessed later, and puts it in closure box
* Function third will now get acces to breakfast and lunch from closure box

Well, we also declared a variable called <u>random insicle second</u>.

Why is it not in the closure box?

Because it's not being referenced later and can be cleaned up the garbage collector :)

@codeWithSimran

So what enables this features? @codeWithSimran

① Functions are first class citizens
function can be returned from another function

② Lexical scope
what variables we have access to depends on where the function was declared.

So using the concept of higher order functions and scope chaining, we can enable closures

Note

① first and second are both higher order functions as they both return functions.

② Before we actually execute the code, the JE knows what variables your code has access to

@codeWithSimran

Prototypal Inheritance

Inheritance is an object getting access to the properties and methods of another object.

We already discussed that arrays and functions in javascript are basically objects

Object {}

Array []

Function()

So the array object and the function object get access to the properties and methods of Object {}

const array = []

array.__proto__

▶ [concat: f, fill f, find: f, ...] ← array

These are basically the methods we use on array right?

Now let's go up the prototype chain, what's on top of Array[], it's Object {} (see above diagram).

▶ array.__proto__ • {}.__proto__

going one chain up (Object {})

→ {hasOwnProperty: f, toString: f, valueOf: f, ...}

▶ array.toString()

▶ ""

How are we able to use toString method on array? Well we just said inheritance is one object (array) can access methods (toString) of another object (Object {})

So whatever is on top of the prototype inheridance chain, you'll have access to it.

Try the same thing with functions and objects

To understand why this concept is important let's take an example

Let's say we have 2 students, who say hi when they come to class and once they finish the assignment, they can leave by saying Bye

```
let student1 = {
        name : 'John',
        assignmentDone : true,
        sayHi : function(){
                    } console.log("Hi");
        sayBye: function(){
                if(assignmentDone)
                  {
                   console.log("Bye")
                  }
}
```

@codeWithSimran

```
let student2 = {
        name : "Ria"
  }
```

And now we dont want to repeat code, so if student2 wants to use method of student 1, we learnt we can use bind.

```
const sayBye = student1.sayBye.bind(student2)
```

we want to use sayBye method of student 1

we want to use it for student2.

But wait, we do have access to sayBye from student1, but sayBye needs assignmentDone variable and student2 does not have it.

So we need to find a sdution that not just lets student2 have access to sayBye but also assignment Done.

We basically want student2 to inherit all functions and variables [properties] of student 1.

@codewithSimran

Sol1

~~Linear~~ student2.__proto__ = student1

- student2.sayBye()
  → Bye
- student2.sayHi()
  → Hi
- student2.assignmentDone
  → true

- student2.name
  → Ria

Jhat means whatever properties student2 already has (name) will be taken from student2. itself, but whatever is new (sayHi, sayBye, assignmentDone) will be inherited (taken) from student1.

Recap

student2.__proto__ = student1

__proto__ : create a prototype chain and

student1 : inherit properties not present in student2 from student1

Exercise

① for(let property in student2)
    console.log(property)

② for(let property in student2)
    if(student2.hasOwnProperty(property)){
        console.log(property)
    }

After you execute these, you'll know that student2 does not actually have properties of student1 copied, instead we just have a reference to them through prototypal inheritance (It looks up the protype chain if property is present)

Functional Programming

Motivation : keep data separate from functions ♥

Pure functions

* The function should always return the same output for a given output
* The function should not modify anything outside of itself. (No side effects)

@CodeWithSimran

1. No side effects.

Example of a function with side effects

```
const obj = { name : 'Simran', language : 'JavaScript' }
function modifyName( objInput )
 {
    objInput.name = 'John';
 }
modifyName(obj)
console.log(obj)
▶ {name : 'John', language : 'JavaScript'}
```

@codeWithSimran

Since this function is modifying something outside of itself (obj) it produces side effects and hence is not a pure function

Imagine multiple functions modifying this obj, it can get really hard to keep track of current value of obj and who modified it.

Let's change the same example to have no side effects.

```
function modifyName(objInput)
{
    let objTemp = Object.assign({}, objInput)
    objTemp.name = 'John'
    return objTemp;
}
```

we now created a local copy inside modifyName and modified that instead. So obj outside does not change.

2. Return same output for same input

```
function allNum (num1, num2)
{
    return num1 + num2
}

allNum (8,2)
```

@codeWithSimran

If you run this function a 100 times, it's always going to return 8+2 → 10. That means given the same input it always returns the same output

Why is that important?
Well, it makes our code more predictable right?

So, are side effects bad? I mean at some point a function will have to interact with the browser, manipulate the DOM etc

@codeWithSimran

To write some meaningful code we will end up having side effects, but can minimize them and organise our code well enough so that it's predictable and we know what is happening where.

Currying

@codeWithSimran

Currying is a mechanism where we can translate the evaluation of a function that takes multiple arguments into evaluating a sequence of functions that take a single argument.

one function taking multiple arguments → transform to → multiple functions taking one argument each

```
function add(a,b){
    return a+b
}
```

```
function addNum1(a){
return function addNum2(b){
        return a+b
    }
}
```

** The reason why we're able to achieve this is because function addNum2 has access to variable a due to closures

So now, how do we use this function?

- addNum1(5)(3) and not addNum1(5,3)
  because addNum1 only takes one argument and it returns a function so result of addNum1(5) is a function (addNum2) and we want to pass the second argument to addNum2

Alright, why would someone even do this in the first place??

Let's say we just call

→ addNum1(3)(5)

this will return us a function, so let's store it for future use

→ const addToNum5 = addNum1(5)

// sometime in future

→ addToNum5(3)

→ addToNum5(4)

we're trying to run less code in future by storing ~~addNumTo~~ addToNum5 for future and not running it everytime

Compose

* The data processing that we do should be obvious

Example:

Let's say we want to make a paste from 5 ingredients mixed together.

item1 —add→ item2 —blend→ newItem —add→ item3

Final Product ← blend ←

So final product is made from item1, item2, item3 and the order doesn't matter

What compose says is we should be able to do the following

item2 —add→ item3 —blend→ newItem —add→ item1

Final Product ← blend ←

* Composability is a system design principle

A highly composable environment components can be assembled in various combinations and still get the right output

@codeWithSimran

Code example :- @codeWithSimran

Let's say we have a negative number, we want to multiply it with another number and return the absolute value

-2 * 3 → -6 absolute 6

Let's compose these together

1) const composedResult =

compose(multiplyBy3, returnAbsolute)

↓

we need to define our own compose func.

~~const compose = (f, g) =>~~

2) const compose = function(funcA, funB) {
3) return function (data) {
4) return funcB(funcA(data))
}

↘ we can also do funcA (funcB(data))

>> composedResult(-2)
→ 6

Let's assume multiplyBy3 and returnAbsolute are defined.

On line one we're calling compose function that takes 2 functions as arguments and returns a function (line 3), this function is stored in composedResult. We can now call composedResult with any data (-2) and it applies funcB on result of funcA

Asynchronous JavaScript

@codeWithSimran

↓

we don't have the data right away

JavaScript is a single threaded. language, it knows nothing of the Outside world

Promises

A promise is an object that may produce a single value sometime in the future.

- either a resolved value
- or a reason why it's not resolved / rejected

3 states of a promise

* fulfilled
* pending
* rejected

But, we already have callbacks, why promises?
Promises were introduced in ES6 and are a bit more powerful, let's see how

Create a promise :-

create new promise

takes 2 parameters either resolve / reject

```
const promise = new Promise((resolve, reject)=>{
        if(someCondition) resolve("worked")
        else reject("something went wrong")
    });
});
```

How to run the promise?

② get the result ↑ @codeWithSimran

```
promise.then(result => console.log(result));
```

① once promise is resolved or rejected

③ use the result ↓

Output : worked (assuming some condition is true)

Chaining in Promises :-

→ same as `{return result1 + 😍}`

```
promise.then(result1 => result1 + 😍)
.then(result2 => console.log(result2))
```

> worked 😍

Explanation :- the first .then() gave us the result and it got passed on ~~th~~ to second .then() This in chaining in promises.

What if an error occurs in any of .then() ?

@codeWithSimran

```
promise
  .then(...)
  .then(...)
  ⋮
  .then(...)
  .catch(() => console.log('error'));
```

→ you can catch the error using .catch

.catch will only catch error of .then() before it. If you have .then() after .catch() it wont catch the error [Try adding throw Error in .then()]

Promises are great for asynchronous programming

* we can store a promise in a variable
* we can do.then() on a promise which can get executed when the promise returns

@codeWithSimran

Combining promises :-

```
const promise1 = new Promise((resolve, reject) => {
        setTimeout (resolve, 500, 'Hi P1')
    }

const promise2= new Promise((resolve, reject)=> {
        setTimeout (resolve, 1000, 'Hi P2')
    }

const Promise_3 = new Promise ((resolve, reject)=> {
        setTimeout ( resolve, 5000, 'Hi P3')
    }
```

To combine all these promises, we can use Promise.all()

```
Promise.all([promise1, promise2, promise3])
        .then((values => { console.log(values); })
```

* It takes an array of promises as an argument
  > ["Hi P1", "Hi P2", "Hi P3"]
* Returns an array of resolved values
* This result is returned after 5000 ms

Async & Await (ES8)

* It is an extention to promises itself.
* Async also returns a promise
* So it's basically syntatic sugar, does the same thing as promises but code looks much more readable.

// with promises [following do the same thing]

```
performTask(task1)
    .then(() => PerformTask(task2)
    .then(() => Perform Task(task3)
    .then(() => PerformTask(task4)
```

// async await

do some async task — inside this function

```
async function playGame(){
    const task1 = await performTask(task1)
    await performTask(task2);
    await performTask(task3);
    await performTask(task4);
}
```

await keyword basically indicates we're waiting for something to give us a response.
Instead of chaining with .then() we're awaiting for every task to perform before next one

Practical example

The fetch() methods that let's you make an API call, is actually a promise, so we can do the following

```
fetch('https://jsonoutput/users')
    .then(response => response.json())
    .then(console.log(response))
```

Because fetch is a promise we're able to do .then() on it

* Note :- response.json() , this json() to convert to json format is also a promise hence again a .then() after it.

→ Let's convert this to async await

```
asyn function fetchStudents(){
    const response = await fetch('https://.....')
    const data = await response.json()
    console.log(data);
}
```

Error handling in async await.

```
async function fetchStudents() {
    try {
        await fetch(url)
        :
        }
    catch (error) {
            console.log(error)
        }
}
```

→ Error handling with async await can be achieved using try catch block.
Put your fetch call (await) inside try and it any error occurs within try then it will be caught in catch block

Job queue

Earlier we said that setTimeout is not a part of JavaScript, it's a part of WEB API

However, with the introduction of Promises Promises let us handle asynchronous code and they are actually a part of Javascript

So we need something to handle these promises just like we need Callback queues to handle setTimeout callback.

// Job queue - Microtask queue

Job queue is a bit smaller compared to callback queue buthas higher priority

So the event loop is going to check the job queue first and then callback queue if it is empty

(guess the output)

```
Promise.resolve('Wohoo').then(()=> console.log('2'))
setTimeout(() => console.log('1'), 0)
setTimeout(() => console.log('3'), 10)
console.log('4')
```

@codeWithSimran

→ 4
2
1
3

How to execute parallel, sequence and race using promises

@codewithsimran

① Parallel

In order to execute promises in parallel (all together) and get their result together we can use

```
Promise.all([promise1, promise2, promise3..])
```

② Race

When we have multiple promises and we only want to return the result of the promise that get's resolved first we can use

```
Promise.race([promise1, promise2....])
```

It will only give result of the promise that gets resolved first

③ Sequence @codewithSimran

What if we have n promises and we want to execute them one after the other ?

```
const p1 = await promise1
const p2 = await promise2
   ⋮
const pn = await promisen
```

In this case after promise1 gets resolved, promise 2 gets executed, once that resolves promise3 and so on....

ES2020 : allSettled. @codewithSimran

We already talked about Promise.all

```
Promise.all([promise1, promise2]).then
        (data => console.log(data))
```

Let's say promise1 gets resolved and promise2 gets rejected.

> Output → Uncaught (in promise) undefined

So we need to put a catch statement to properly handle this

So promise.all resolves only if all the promises resolves. Even if one of them gets rejected, its going to result in an error.

@codewithsimran

for that matter, we have Promise.allSettled which will result in an array of outputs from the promises, if they get resolved or rejected.

Promise.allSettled([promise1, promise2])

> [{status: "fulfilled", value: 'somevalue'},
{status: "rejected", value: undefined}]

Modules in JavaScript

→ Highly self contained and have their own dedicated functionality

→ We generally have multiple JS files for our whole project

→ So we need to be able to import functionali -ty that we want and export the same

Before diving in let's see what ways we already know?

① We can have functions (they can have their own variables), but for shared data we will need global variables (which is bad right?)

② We can have multiple script files (if they have global variables with the same name, it's going to override it)

@codeWithSimran

Modules give us a better way to organise variables and functions so when we group these together it makes sense

Where do modules come?

Global scope @codeWithSimran
↘ Module scope
↘ Function scope
↘ Block Scope

Ways to achieve module scope

Before

① IIFE (Immediatly invoked function expression)

↳ Let's say want to wrap your entire JS content inside an IIFE

```
var module1 = (function() {
    var name = 'Simran';
    :
    return {
        name: name
    }
})();
```

← anything you want other files to have access, you can return and keep the rest private to the IIFE

This is called the revealing module pattern

You only reveal (return) what you want from the module

The other file that now wants to use variable name can simply do

module1.name

@codeWithSimran

Note: after wrapping your code in an IIFE it's no longer a part of the global scope and we're not polluting the global scope.

But what about the variable module1 ?

Well it's still a global variable

* So we can still override module1 in another file (problem 1)
* We need to keep track of all the dependencies.

For example

@codewithSimran

```
<script src='./script1.js>
<script src='./script2.js>
<script src='./script3.js>
```

Now if script1 and script3 use a function sayName that is defined inside script-2, script1 wont have access to it (because script 2 is included after script 1) and we'll get an error.

Summary

@codewithSimran

2 problems

① Still a global variable (module variable)
② need to maintain the dependencies

In order to solve the problems discussed with module pattern, we have something new that was introduced.

CommonJS & AMD

★ CommonJS

① We can import different JS files using

```
var module1 = require('module1')
var module2 = require('module2')
```

module1 and module2 are the names of the JavaScript files

② To export a function from a file we can do the following

```
module1.exports = {
    function1 : sayHello,
    :
}
```


To import the same

```
var module1 = require('module1').function1
```

So no ~~impo~~ IIFE and global variable collision issues. Just simple 'import, export :)

We have a small issue here
The require statement to import a module is actually synchronous. It can take a lot of time to load right? Bad user experience on the browser

@codewithsimran

So then we had something called browserify that takes a script and converts it into a bundle (even webpack can do things like that).
This bundle file has all the javascript files combined in one single bundle with all the dependencies (whatever we import) and we actually use this bundled file to run things on the browser and since we already have all the dependencies, it wont slow things down

ES6 Modules

@codewithsimran

Finally we have something natively available

To import something

```
import module1 from 'module1';
```

module1 ↓ what we export

'module1' ↓ name of the JS file

To export something

module1.js

```
export function sayHello(){

}
```

@codewithsimran

```
import {sayHello} from 'module1'
```

Finally let's export multiple things

```
export function func1(){
}
export function func2(){
}
export
```

→

```
import {func1, func2}
from 'module1';
```

Errors in JavaScript

There is a native JavaScript constructor Error in JavaScript.

Go to console and type

> Error

f Error() { [native code] }

@codewithsimran

So you can create an Error with this constructor function as follows

new Error('any message')

↳ new error instance

But this dosent mean that we have an actual error, we need to actually throw an error

> throw new Error()

Uncaught Error → now we have an actual error

@codewithSimran

And when we throw something, the execution of our program stops.

```
console.log('1')
throw new Error()
console.log('2')
```

@codewithsimran

```
> 1
> Uncaught f Error()
```

2 is not even printed because the execution of our program will stop

When we create an error, we can access some properties on it

```
const someError = new Error('Damn Error!')
> someError.name
    → Error
> someError.message
    → Damn Error!
> someError.stack
  "Error: Damn Error
      at <anonymous>:1:17"
```

@codewithsimran

The error occured on the global execution context

Now let's define an error inside of a function

```
> function randomError()
  {
      const someError = new Error('Damn Error!')
  }
randomError();
> Error: Damn Error!
      at randomError (<anonymous>:...:)
      at <anonymous>
```

② we called randomError and the error occured inside it

① first we had execution context

We can also check @codeWithSimran

```
som randomError().stack
....
```

So we can trace where exactly the error occured

We also have other error in JS

- new SyntaxError → example £,
- new ReferenceError → when something is not defined

So how are errors handled in Javascript behind the scenes?

When an error occurs, it check the execution context on top of stack if there is a catch (to handle the error), if not it goes one level down and checks the next execution context if there is a catch

@codeWithSimran

If there is no catch anywhere, the onerror() function gets run on the browser

So the runtime catch : onerror() will handle the error for us

But again, this will stop the execution of the program and hence we should have our own ~~of~~ catch statements

TRY CATCH

```
function sayHello(){
    try{
        console.log('Hi Simran');
    }
    catch(error) {
        console.log('opps! Error!')
    }
}
```

execute all the code inside of this block

The error that occured

If any error occurs inside try { } block, handle it inside the catch block


```
function sayHello() {
    try {
        console.log('Hi Simran')
        throw new Error('Damn error!')
        console.log('executed success')
    }
    catch(error) {
        console.log('Error', error)
    }
}
```

> Hi Simran
Error Damn Error!

So it executes code inside of try { } block until it finds an error, and then moves to catch block. That's why executed success is not printed.

finally

@codewithSimran

```
function sayHello() {
    try {
    }
    catch (error) {
    }
    finally {
        console.log("Always printed")
    }

    console.log("outside try catch")
}
```

→ No matter what part of try and catch gets executed, whatever code we write inside finally will always get executed

↳ This piece of code will never get executed. (outside try catch)

* Try catch can be used to handle synchronous code
* You can have nested try catch blocks

Error handling - Asynchronous code

```
try {
    setTimeout(function() {
        random;
    }, 1000)
} catch (error) {
    console.log('error', error)
}
```

→ Since, setTimeout will take sometime this code would have already executed.

@codewithsimran

So in asynchronous code when something fails in this case, we don't even get an error

```
Promise.resolve('silentfail')
  .then(res => {
      throw new Error('fail')
      return res
  })
  .then(res =>
  })
```

→ Even though we are ~~getting~~ throwing an error, we dont get any error!

Because it silently fails! Not good!

** We should always have a .catch with promises, otherwise we won't know what might happen.

** Even if you have nested promises you should write a catch statement for each of those.

@codewithsimran

```
Promise.resolve('fail')
  .then(res => {
      Promise.resolve().then(() => {
          throw new Error('Big fail')
      }).catch(error) => {...}
  })
  .then(
  )
  .catch(err => {
  })
```

→ for the inner promise

→ for the outer promise

If we don't put a .catch for the inner promise, it'll still be handled by the outer catch (But in some environments) you'll get a warning.

Error handling with asyn await

```
const errorFunc = async function() {
    try {
        await Promise.reject('fail')
    } catch(err) {
        console.log(err)
    }
}

errorFunc();
```

@codewithsimran

Since async await let's you write code in a synchronous fashion (await line by line) we can use a try catch block to handle errors.

Object oriented programming

Let's say we have a game, it has players as follows

```
const player1 = {
        name : 'John',
        battery : 20,
        charge(){
            retro player1.battery = 100
        }
}

const player2 = {
        name : 'Lisa',
        battery : 30,
        charge(){
            player2.battery = 100
        }
}
```

So if the game has n players, we're going to create n objects.

* Above, we have bundled the data (name, battery) with the methods that operate on the object (charge) and this is nothing but encapsulation

However, we have to repeat code for every player even though they have the same charge method.

Soln: factory functions: Functions that create the object for us.

```
function createPlayer(name, battery){
    return{
        name: name,
        battery: battery,
        charge(){
            this.battery = 100
        }
    }
}
```

→ let player1 = createPlayer('John', 20)

player1.charge()

→ let player2 = createPlayer('Lisa', 30)

Here createPlayer is a factory function that creates objects for us. Now we don't need to repeat any code.

However, the function charge is going to be same for all objects we create, and using createPlayer everytime creates the charge method for every object in the memory.

One possible soln

Extract out all common functions outside like this

```
const createPlayerFuncs = {
        charge : function charge() {
                this.battery = 100
        }
}

function createPlayer (name, battery){
        return {
                name : name, battery : battery
                charge: charge
        }
}
```

→ let player1 = createPlayer('John',20)
player1.charge = createPlayerFuncs.charge
Player1 now has access to charge (A new copy is not created, it points to existing charge in memory)

→ player1.charge() (To use it)

We can repeat the same process for any numbers of players. They will be all pointing to the same charge function in the memory.

But this seems like a lot of manual work, so in the next session we will look at another way to do it via object.create() and using prototypal inheritance principles

Object.create()

```
const createPlayerFuncs = {
        charge: function charge(){
            this.battery = 100;
        }
}

function createPlayer(name, battery){
    let newPlayer = Object.create(createPlayerFuncs)
    newPlayer.name = name;
    newPlayer.battery = battery;
    return newPlayer;
}
```

Object.create creates a link from newPlayer to createPlayerFuncs object.

So now newPlayer object that we created via Object.create() has access to createPlayerFuncs object through prototypal inheritance

If you console.log(newPlayer) you won't see charge function, but if you log newPlayer.__proto__ you can see it because newPlayer is pointing upto createPlayerFuncs in the prototypal chain.

Another alternative → Constructor functions

```
function createPlayer(name, battery) {
        this.name = name;
        this.battery = battery;
}

const player1 = new createPlayer('John', 20)
> player1.name
> John
```

Any function invoked with the new keyword is a constructor function.
We don't need to return the object from the constructor function, instead it is automatically created and returned

Convention : Always start your constructor function with capital letter.
So CreatePlayer instead of createPlayer

** when it comes to constructor functions, 'this' keyword dosent point to global (window) object

instead it points to the object we want to create. In this case player1.

So this.name = name; is equivalent to player1.name = name

That's how we get these properties in player1

So all of it happens because we're using new keyword

Now how do we actually attach charge function when using constructor functions.

When it comes to constructor functions, we can attach any new properties using prototype that every function has access to when created

```
createPlayer.prototype.charge = function(){
    this.battery = 100;
}
```

ES6 classes

Classes are a template for creating objects They encapsulate the data and your methods (finally everything in one place)

So here's what our code will look like

```
class Player {
    constructor(name, battery) {
        this.name = name;
        this.battery = battery;
    }
    charge() {
        this.battery = 100;
    }
}
```

Classes have constructor functions that run every time we instantiate a class (use the new key word)

All the methods (charge) can also be encapsulated inside the class itself

Creating an instance of a class simply means we're creating an object from the class.

> let player1 = new Player('John', 20)

by using the new keyword we're creating an instance (Obj player1)

Note :- Classes are only syntactic sugar, the class keyword is just like using prototypal inheritance (behind the scene) So it doesn't have the real class concept like it exists in other languages.

Interview question : Why are methods not put inside the constructor?

→ When we use new keyword, an instance is created based on the constructor function (name, battery)

If we put the method (charge) inside constructor then it will be created by with every object which is memory inefficient. So by keeping it outside, all objects will reference to charge which is present in one place in the memory

Inheritance in OOP

Let's say in our game we have players, ~~they~~ there could be different types of players in the same game. Let's say our game has a hero who can have weapons like gun, knife etc and the villian who can destroy things around with a special power.

If we wanted to design this game, both hero and villian would have name, battery and charge function. But only hero has a weapon and villian has a method that destroys things.

How do we solve this?

Option 1: create 2 different classes each for hero and villian. But we would be repeating code, since they both have something in common. This is where inheritance comes into picture

* We first create a (common) base class that has the common functionality for all players.

```
class Player {
    constructor(name, battery) {
        this.name = name;
        this.battery = battery;
    }
    charge() {
        this.battery = 100;
    }
}
```

* Then we create a class for hero/villian that inherits all that Player class has and any new functionaly we need to add

```
class Hero extends Player {
    constructor(name, battery, weapon){
        super();
        this.weapon = weapon;
    }
}
```

* extends is a keyword that helps us inherit from ~~for~~ a base class (also called super class) ~~now Hero class~~

> const hero = new Hero('John', 80, 'gun')

Now a new hero is created with a weapon gun which a general Player doesn't have.

Also hero has access to data and methods defined in Player class

> hero.charge();

↳ hero has access to charge method

* We are not instantiating Player anywhere, we're directly instantiating Hero class, but Hero extends Player, so we need to pass the parameters (name, battery) required by Player class in order ~~to~~ for it to work.

So what about super()? Why is that required?

In order for a class (that extends from another class) to have access to the this keyword, we need to first always call super() because ~~super actu~~ this

* actually comes from super class

So if we try to do

this.weapon = weapon before calling super() it will give us an error that this is not defined.

And what does this contain after we call super?

> this

{name: 'John', battery: 80}

and by doing this.weapon = 'gun' we adding to the this coming from super

> this.weapon = 'gun'

> this

{name: 'John', batt-ery: 80, weapon: 'gun'}