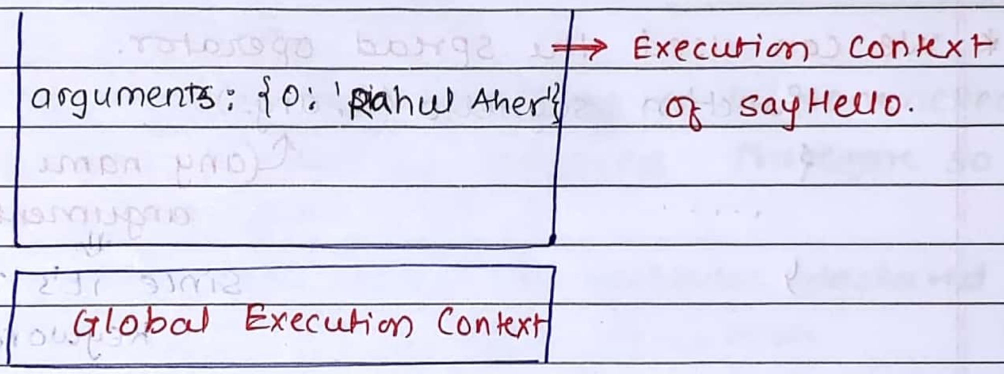
- JavaScript - Day 4

- Arguments keyword :-

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

sayHello ("Rahul Aher");
```

→ We already know calling a function creates a separate execution context for it.
→ If you see the execution context of sayHello you will see an object name arguments.


arguments : {0: 'Rahul Aher'}
→ Execution context of sayHello.
Global Execution Context

→ If we passed 2 arguments we had get

```
arguments : { 0 : 'Firstargument',
              1 : 'Second argument'
            }
```

→ So arguments is an object, it's not an array !!!
All the keys of arguments object are 0, 1, 2, 3, ...

* How do we iterate the arguments object?
→ You can use the new Array.form.

Rahul Aher

Page No. Date ②

* What does Array.form do?

→ let's say our arguments object is
```
{ 0: 'Rahul'
  1: 'Tina'
  2: 'John' }
```

> Array.form (arguments)

output> ['Rahul', 'Tina', 'John'];

→ It return us an array of all values :)

→ We can now use any array methods on it.

OR

* We can used the spread operator.
```
function sayHello (... args)
{
   .....
}
```
↑(any name but not arguments) ⇓ Since it's reserved keyword.

→ now we can do args[0], args[1] and so on.

* What happend when you don't pass any arguments to a function.

→ you will still gets arguments object but it will be empty {}

● SUMMARY :-

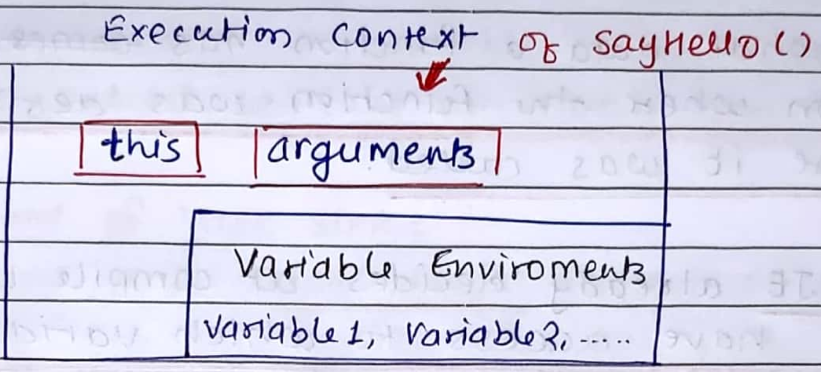
→ There can be multiple execution context.

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

→ Execution context of every functions has arguments object.

- **Variable Enviroment :-** i.e. Memory space of each function.

→ Whenever we create an execution context (like calling a function) we so far know we get this and arguments object. But there's one more thing



→ Since variable declared inside a function are local to that function, they resides in the variable enviroment or Local Enviroment.

→ Once the function stop executing or is done executing it's execution context is removed. Therefore so is the variable enviroment.

→ So we can't have access to variables declared inside a function when it is done execution.

- **Scope Chain :-**

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

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

```
var name = 'Rahul';
function sayHello()
{
   :
}
function sayBye()
{ :
}
```

→ Both sayHello & sayBye have access to the variable name.

*** [All functions have access to global scope]

→ So what data a function has ~~comes~~ access to depen ds on where the function was defined and not where it was called.

→ So JE already decides at compile time what funct will have access to which variable because it knows where a function is defined. (when it scans the file) but it doesn't care about where the function is called.

⇒ Go to the console define a function.

```
> window      ⇒ (this show have your functn,
                 sayHello() for example)
  :
  :
  sayHello : :   ⇒ [say-Hello will have [[scope]]
             :      and it will tell you it's scope
             :      in this case global.]
    [[scope]]
```

* Exercise

```
function leakage () {
        name = "Rahul Aher";
}
```

- where is the name in the Execution context?

→ It should be in the execution context of leakage

Rahul Aher

function right?

→ But, it's not. Since we didn't declare name with var, let, const etc, leakage function say 'Hey I don't have the variable name and passes on to global context, global context says I don't 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 doesn't let you create variable without actually declaring them (using var, let, const or etc).

→ Okay.... 'use strict' is a nice friend. Doesn't let you go in the wrong direction ;D.