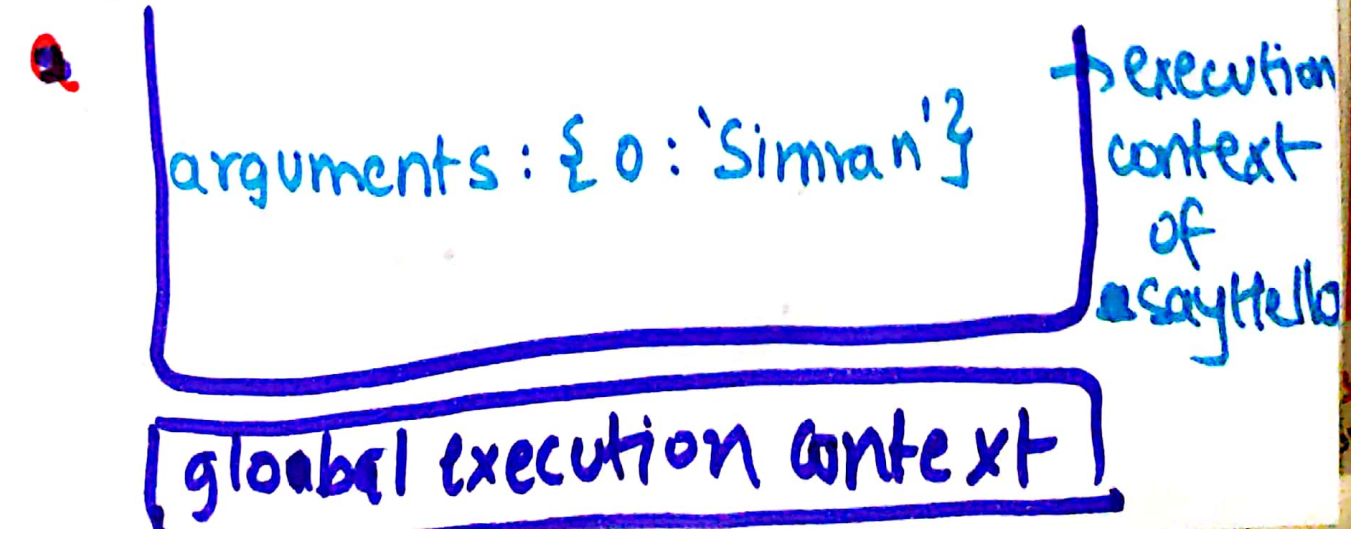
Arguments keyword.

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

@codewithSimran

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 spread 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

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

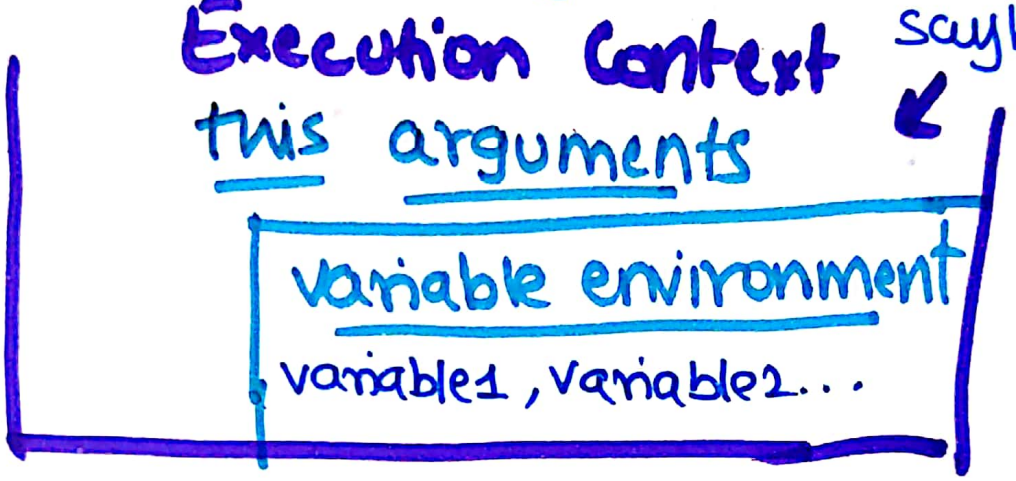
→ There can be multiple execution context,

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

→ Execution context of every functions has an arguments object

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

→ 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()
 {
  :
 }
```


→ Both sayHello and sayBye have access to the variable name

[All functions have access to global scope] **

In Javascript Static ~~set~~ scope defines (~~what~~

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

[[Scope]]

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 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 dosent let you create variable without actually declaring them (using var, let or const)

Okay.... 'use strict' is a nice friend. Dosen't let you go in the wrong direction :D