

# Arguments keyword.

```
function sayHello(name)
{
    console.log("Hello" + name);
}
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 see an object named arguments.

arguments : { 0 : 'Simran' }

→ execution context of sayHello

global execution context



codeWithSimran



codeWithSimran\_

if we passed 2 arguments  
we'd get

@codewithSimran

arguments: { 0: 'first argument',  
3 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.from  
what does Array.from do?

Let's say our arguments object  
is { 0: 'Simran', 1: 'Tina',

2: 'John' } @codewithSimran



codeWithSimran



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



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

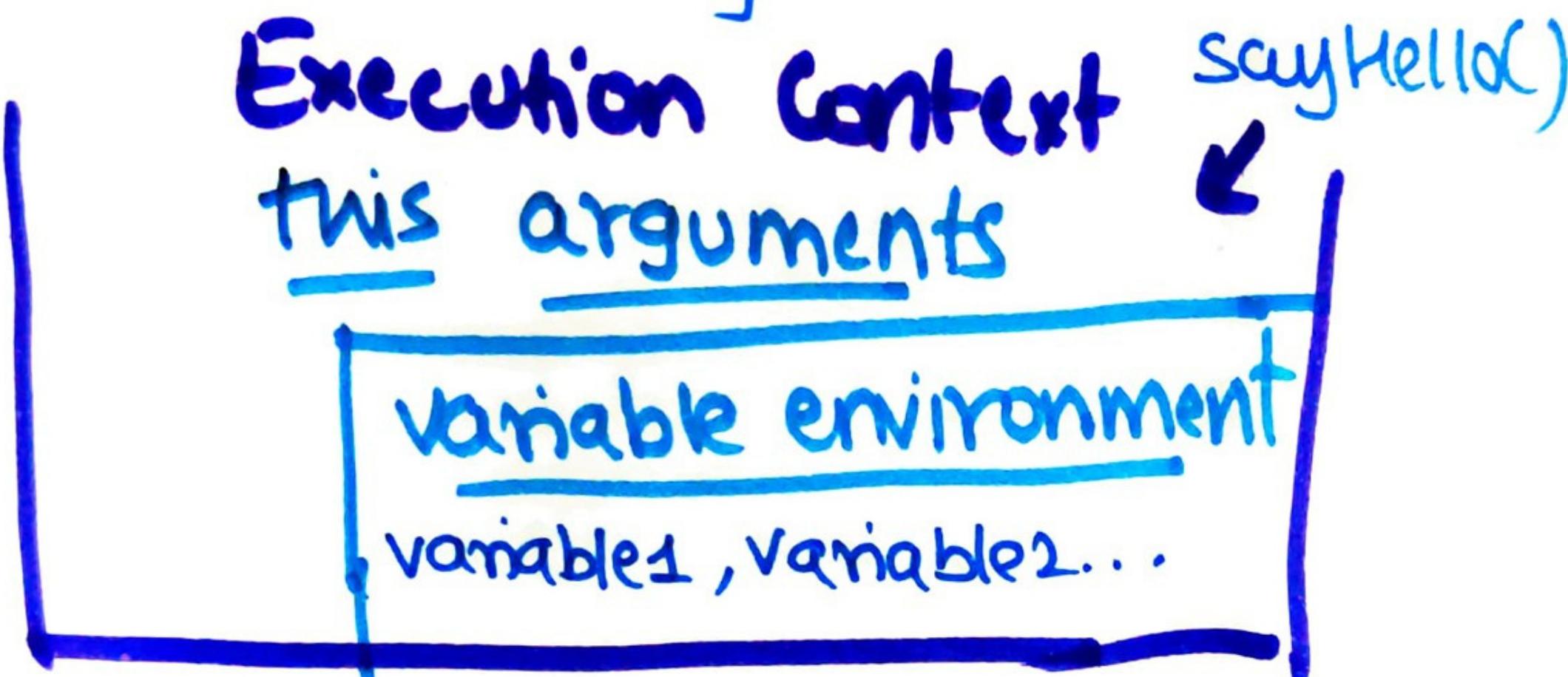
## Summary

- 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



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

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

```
function sayHello()
```

```
{  
:  
}
```

```
function sayBye()
```

```
{  
:  
}
```

→ Both sayHello and sayBye have access to the variable name

@codewithSimran

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



codeWithSimran



codeWithSimran\_

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 doesn't care about where the function is called.

Go to console define a function  
>window → (tws show have your function, sayHello) for example  
sayHello:: → [sayHello will have [[Scope]] and it will tell you its scope in this case global)  
[Scope]



codeWithSimran



codeWithSimran\_

## 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 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 JS 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 or const)

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

