Javascript –

- popular
- weird gotchas
- may be using it for a while; but how does it work under the hood?
- javascript is nothing like C++, Java, etc. It's a different concept
- Deceptively powerful and a beautiful prog. language

UNDERSTAND ? will help

DON'T JUST COPY !!! ~~will help~~

- what if none very hard problem you encounter in your work?

1 ✓ CHROME, 2 ✓ VS Code

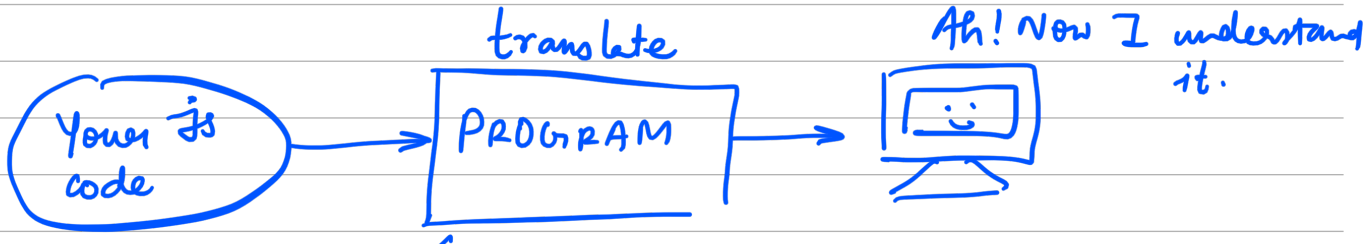
F12 → Developer tools

3 ✓ Live Server <Extension>

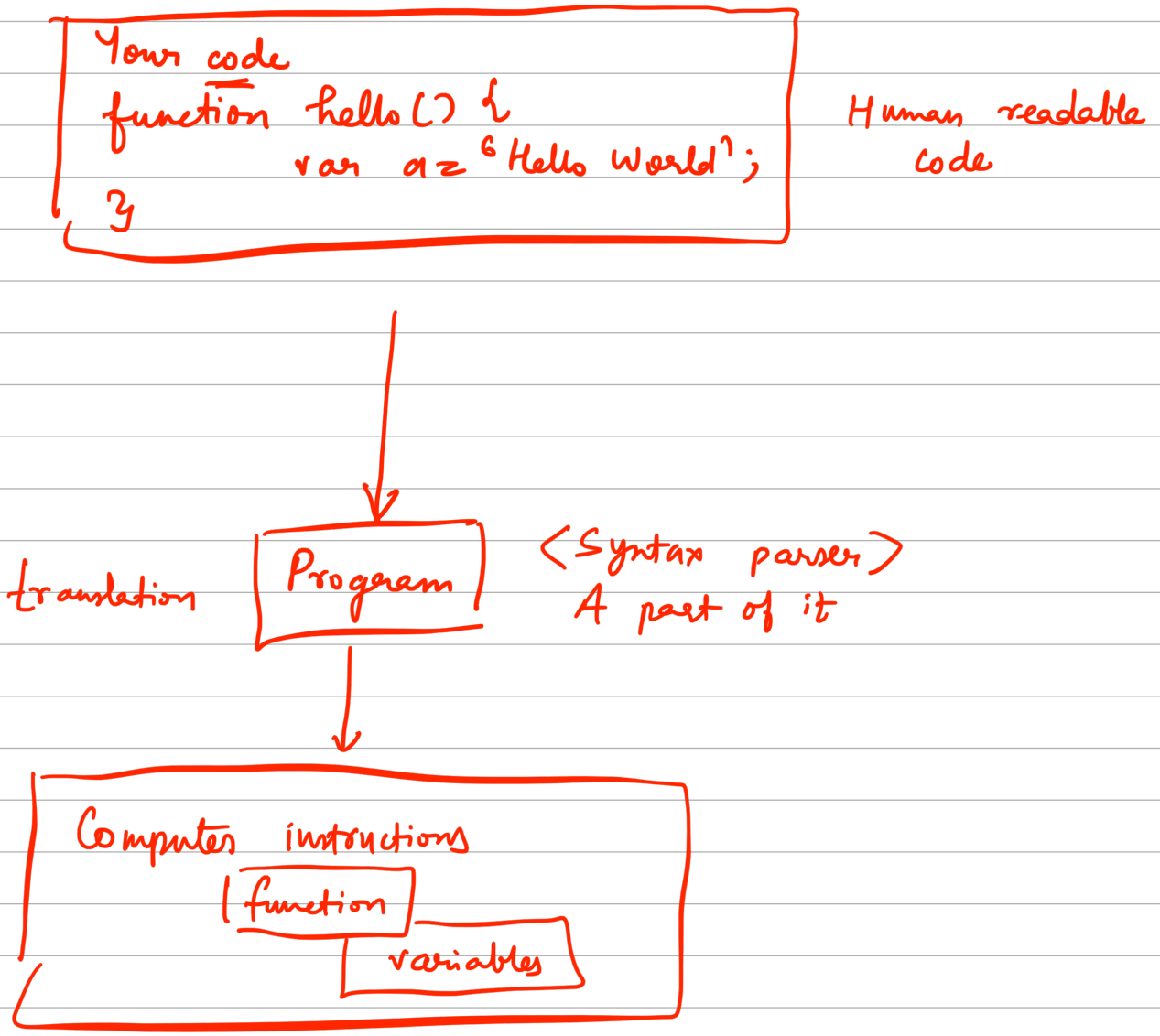
- abstraction over how the computer understands it...
- jQuery, React, Angular are just javascript code...

Syntax Parsers –

A program that reads your code and determines what it does and if the grammar is valid



<Compiler, Interpreter>
<Syntax Parser is a part of it>



Lexical Environment-

- where something sits physically in the code you write
- exists in programming languages in which where you write something is important

Execution context-

- A wrapper to help manage the code that is running
- There are lots of lexical environments. Which one is currently running is managed via execution contents. It can contain things beyond what you have written in your code.

Name/Value pair -

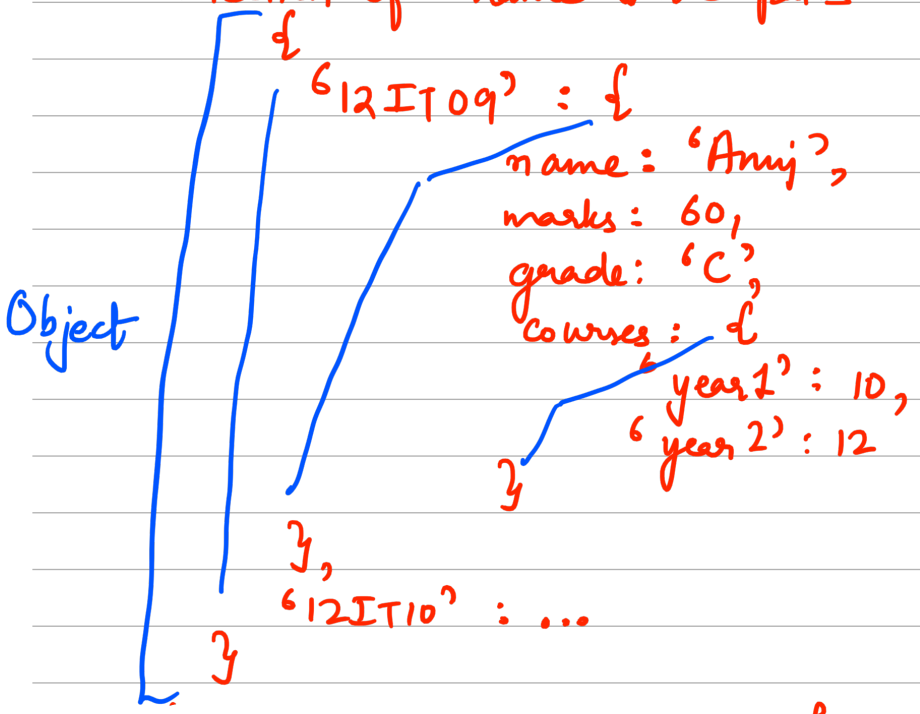
A name which maps to a unique value

The name may be defined more than once, but only can have one value in any given context
The value may be more name/value pairs

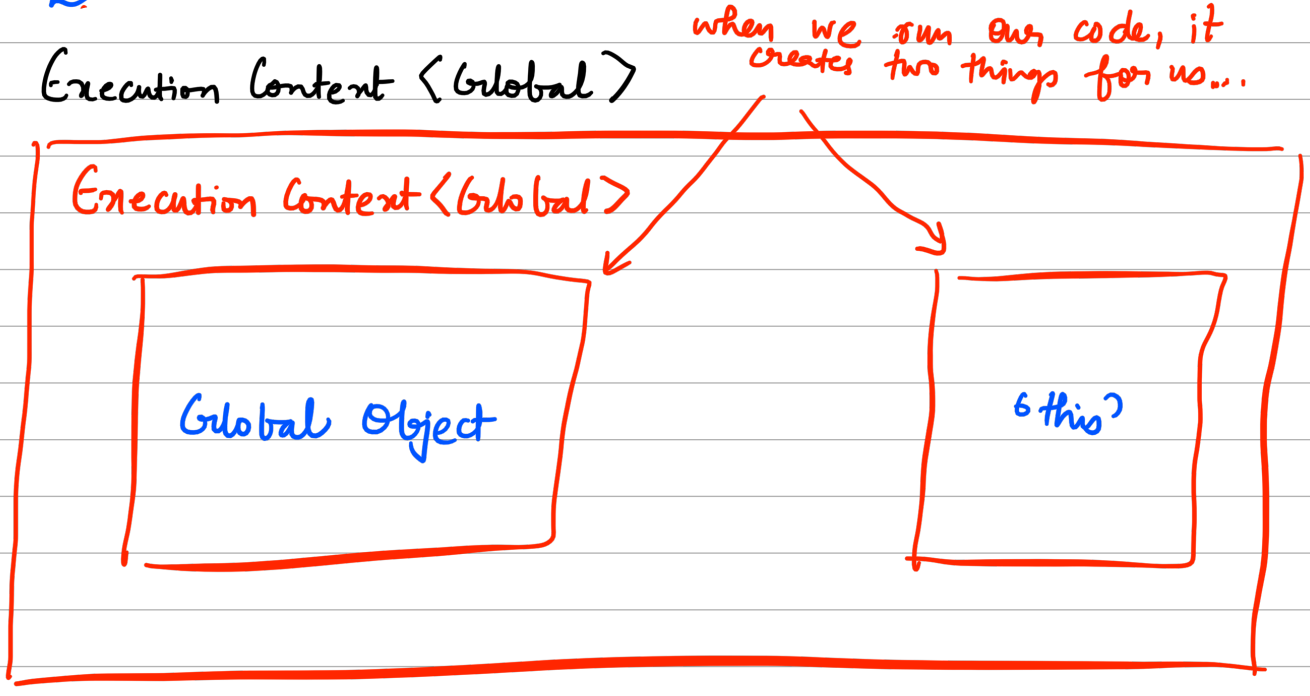
```
my_name = 'anuj';
```

Object -

Collection of name value pairs



Execution Context <Global>



Execution context was created by the javascript engine...

global This
this → window
window → current tab

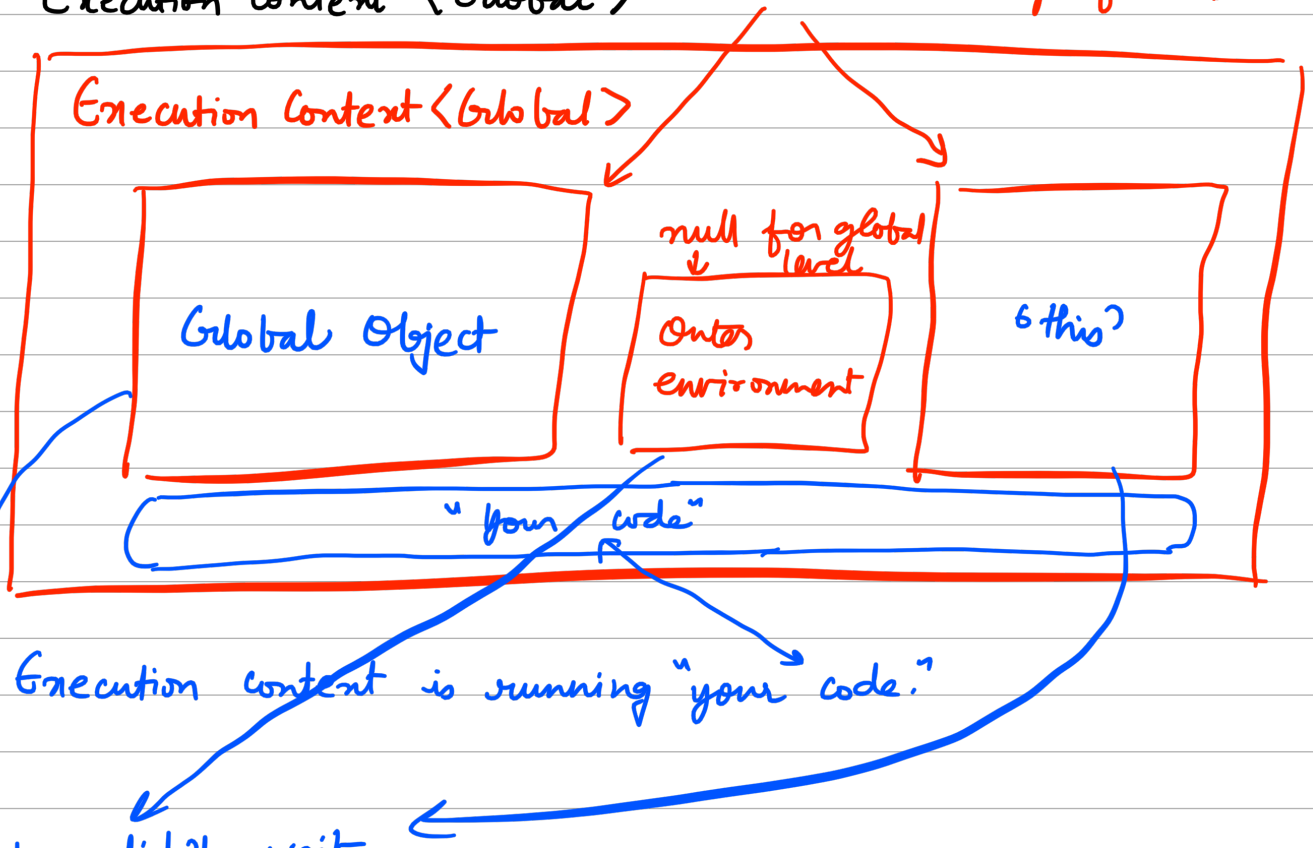
Each window has its own execution context.
1*

Global

- "Not inside a function"
- 2*
- when variables and functions are not lexically inside a function, they are at a global object

Execution Context <Global>

when we run our code, it creates two things for us...



Execution context is running "your code."

we didn't write

CODE →

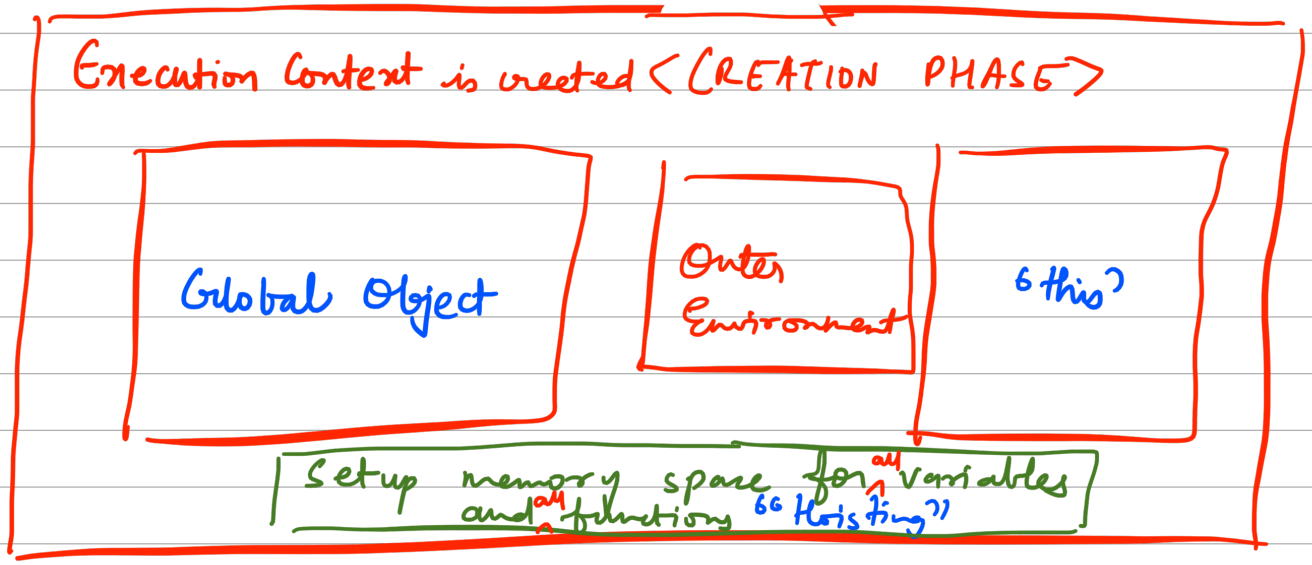
```
printName();
console.log(name);
var name = 'Hello World';
function printName() {
    console.log("Anuj");
}
```

Global Exec^n Context has evolved.

o/p →

Anuj
undefined

Why ? It's not moving code. Hoisting?
Execution content is created in two phases
- Creation
    Global Object, this, Outer Environment



- functions are sitting in its entirety in the memory...
- variables are there but it's undefined* (or default value)
  [[Global execution context → evolved]]

It's only when the code is executed line by line, that it assigns the value 'Hello World' to name

- Do not write code like above. It's a bit confusing!

Hoisting - variables setup (and set equal to undefined)
- functions setup

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

undefined → means that the variable has/it been set

```
a
```

- ReferenceError: a is not defined
- never set up the memory space. Looks in memory space, not found. Hence, error.

- Never set your variable to "undefined"... why?

JS ⟶ meant it as "something that has/it been set"

1. CREATION PHASE
2. EXECUTION PHASE
   → setup completed
   → runs your code "line-by-line"

4*

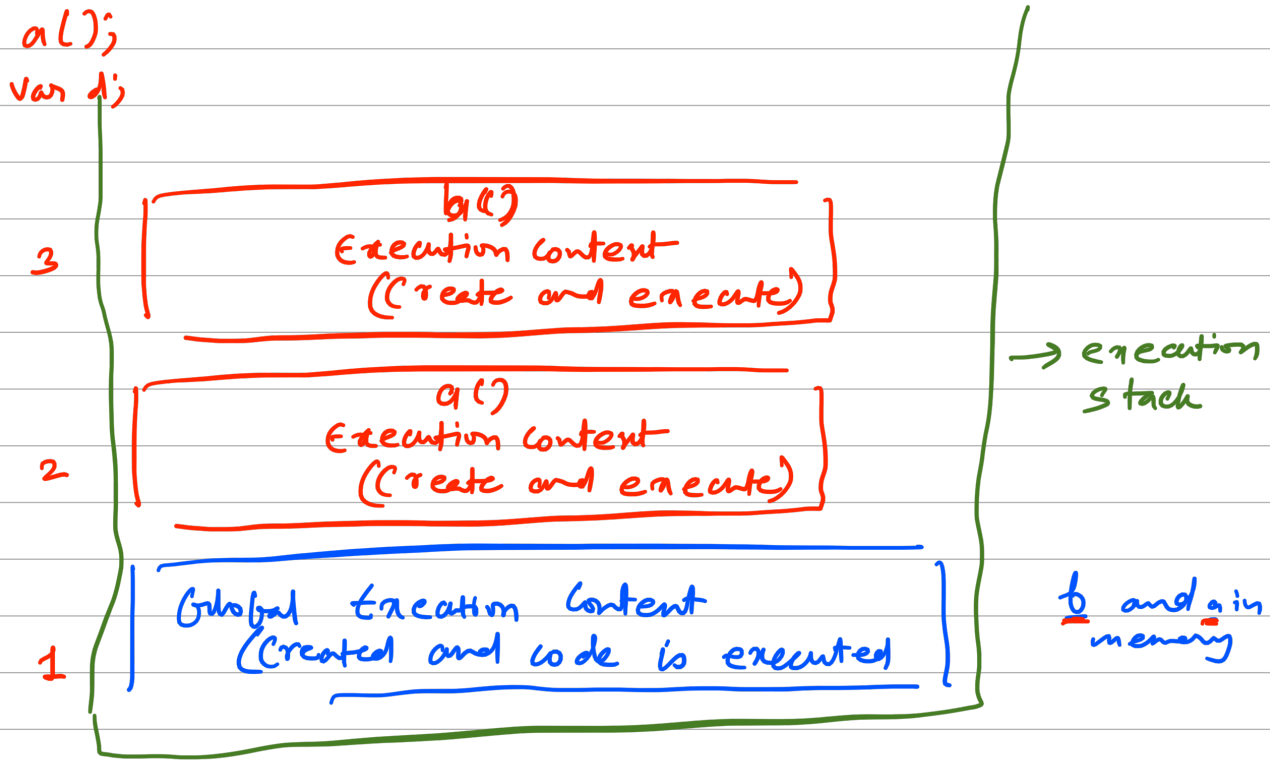
Single thread -
- execute one command at a time

under the hood of browser, may be not

Synchronous -
- one at a time
- in order

function invocation →
- running/a function
  calling
- by using parenthesis ()

```
function b() {
    var d;
}
function a() {
    b();
    var c;
}
a();
var d;
```

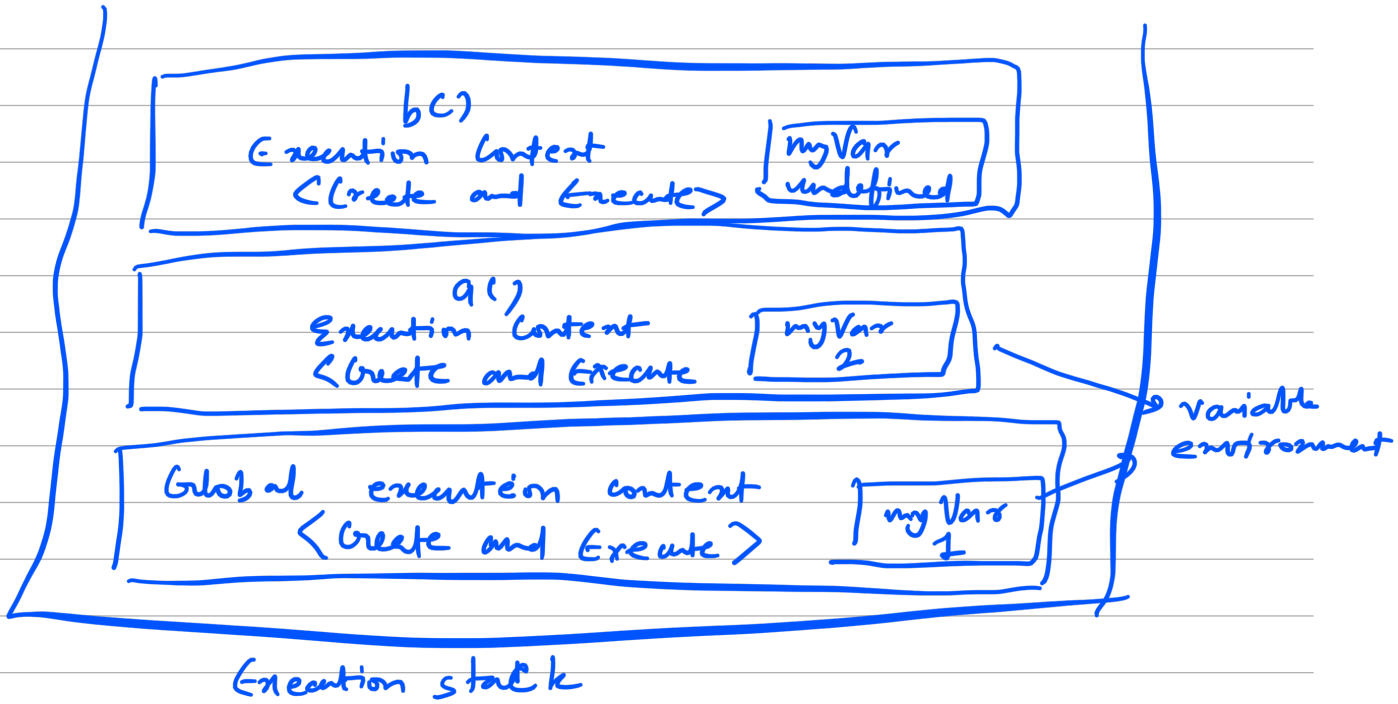


- Only when b → finishes; it goes about finishing a.
  <execution context of b pops off>
- A new execution content is created for that function that is executed.
- whatever at top, is actually running. Run it line-by-line

Variable environment –
- where the variables live
- how they relate to each other in memory.

```
function b() {
    var myVar;
}
function a() {
    var myVar2;
    b();
}

var myVar1;
console.log(myVar);
a();
console.log(myVar);
```



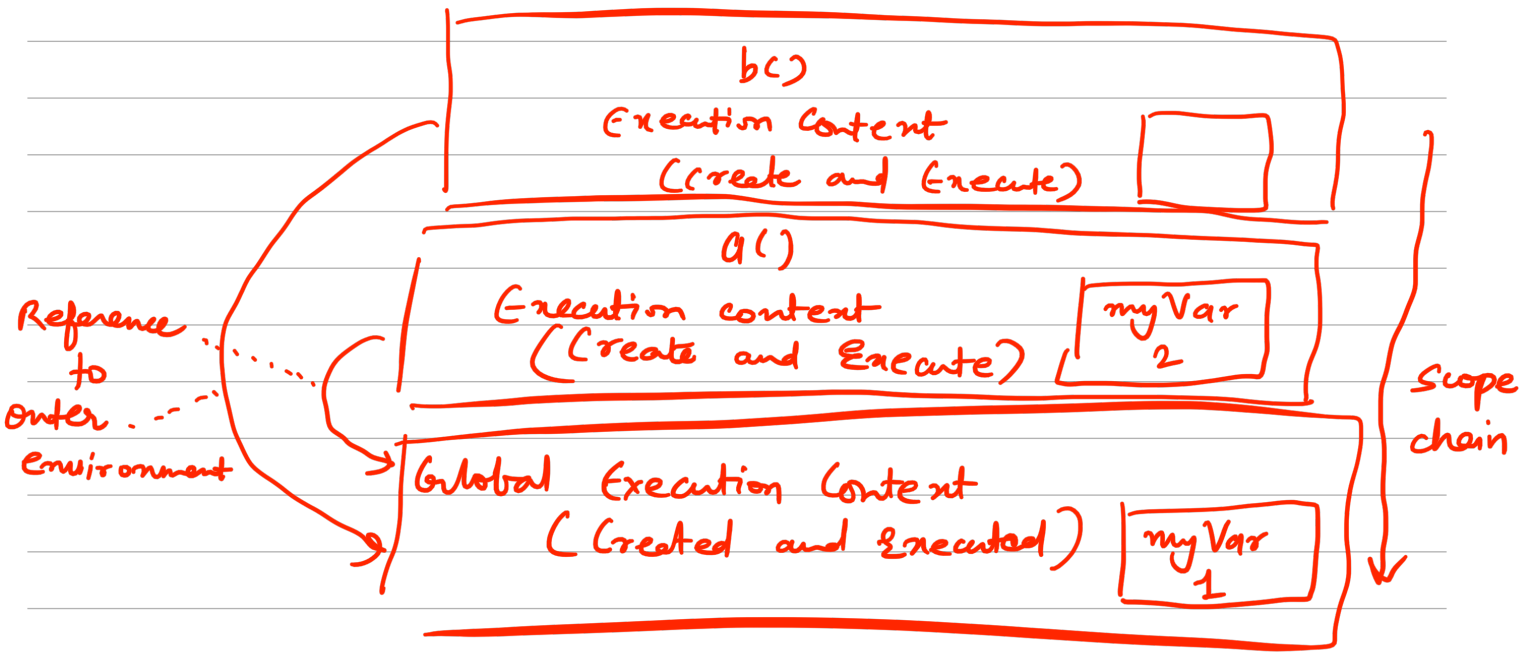
Each execution context has its own variable environment.

5*

6*

```
function b() {
    console.log(myVar);
}

function a() {
    var myVar = 2;
    b();
}
var myVar = 1;
a();
```

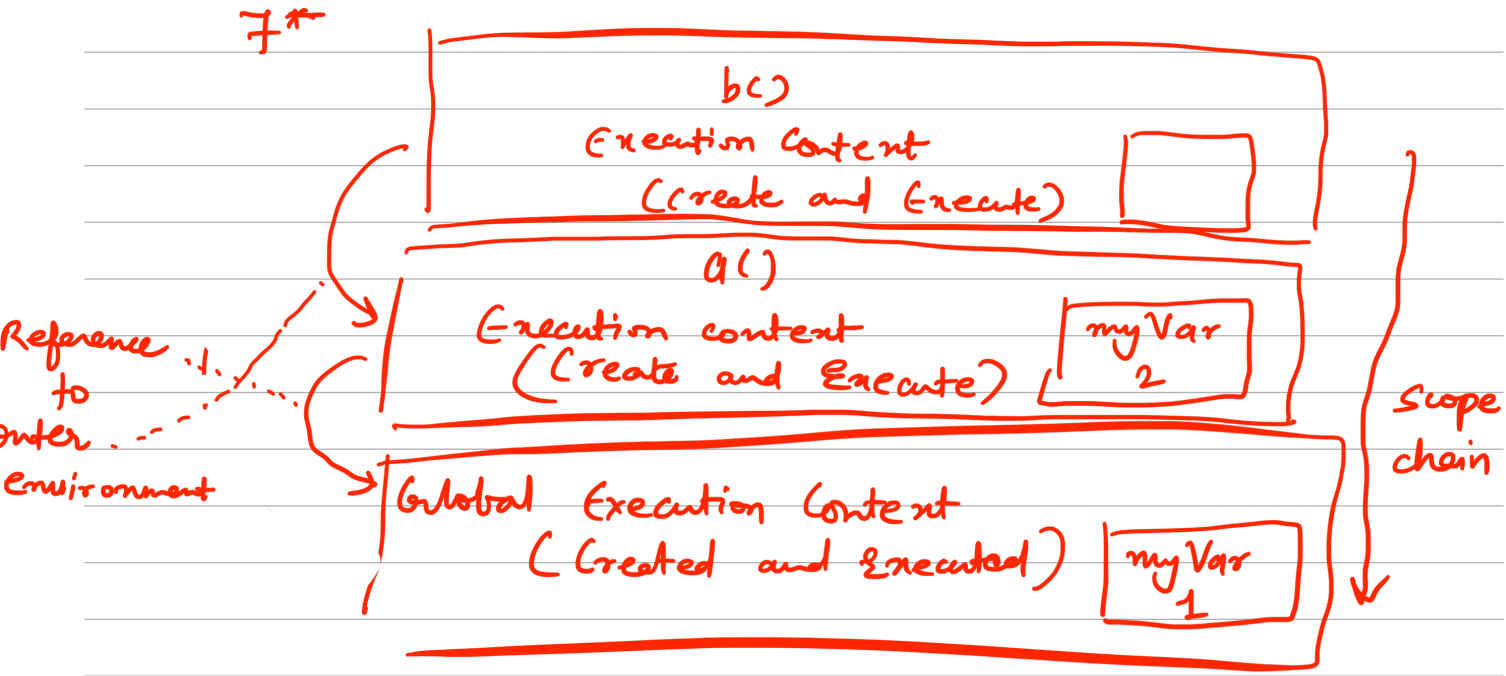


b cannot find myVar. Will look at its outer environment. find myVar in the global execution context.

- b sits at the global level
- b sits lexically at global level. So its outer environment is global.

"WHERE IT SITS" is important

- Outer reference is created based on where the function is actually sitting in your code.
- Go through outer references to check if it's sitting there or not. Keep going... called Scope chain.



→ where something sits physically... | who created me...

→ is my outer reference

Scope–

- where a variable is available in your code
- and if it's truly the same variable, or a new copy

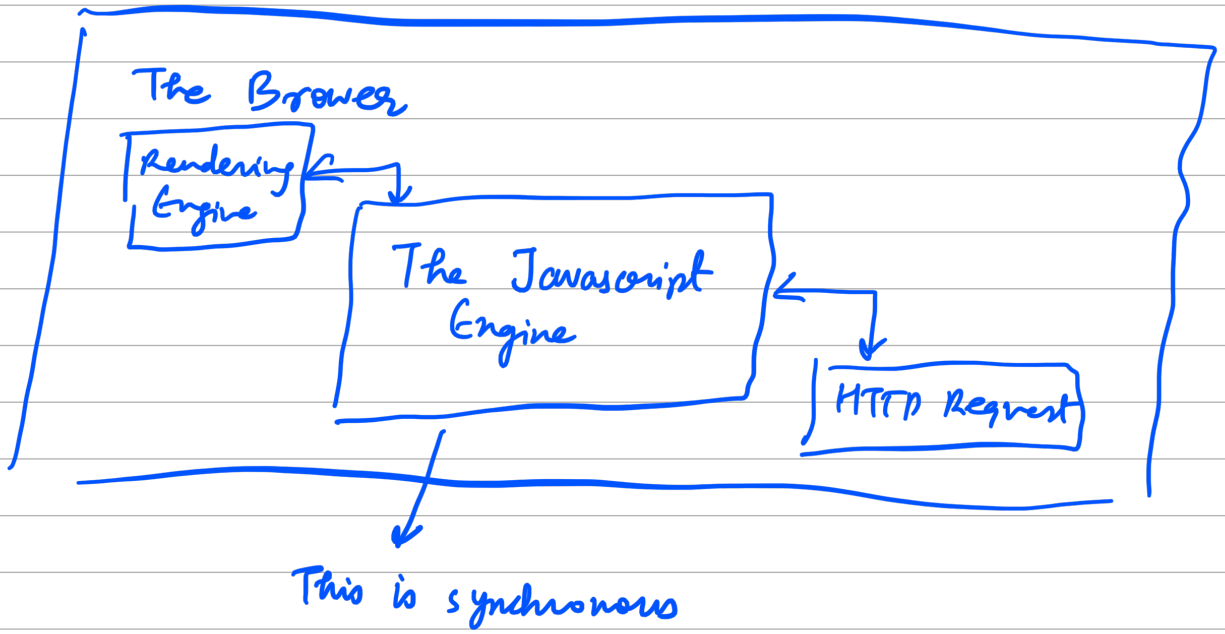
let- ES6

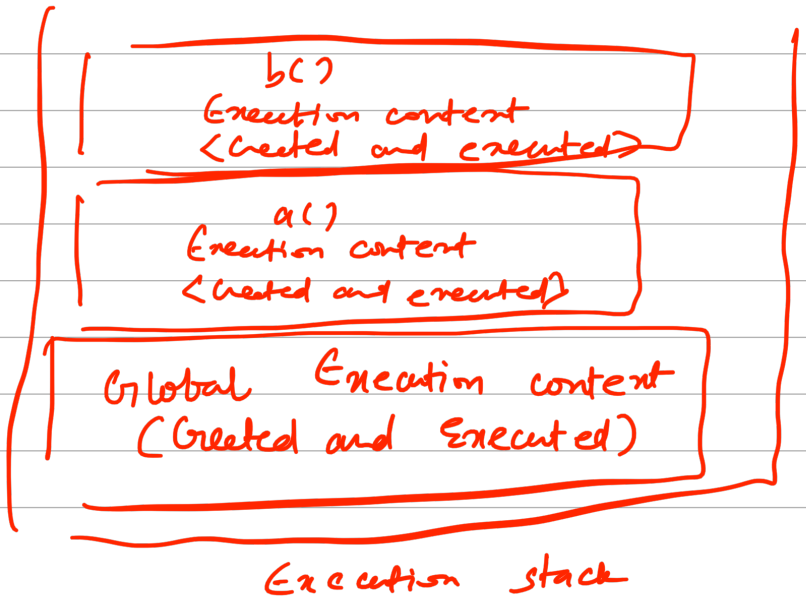
- allows the javascript environment and have block scoping { }

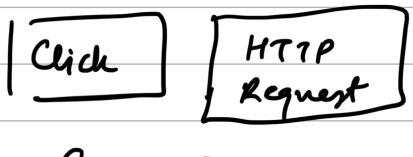
q*

Asynchronous–

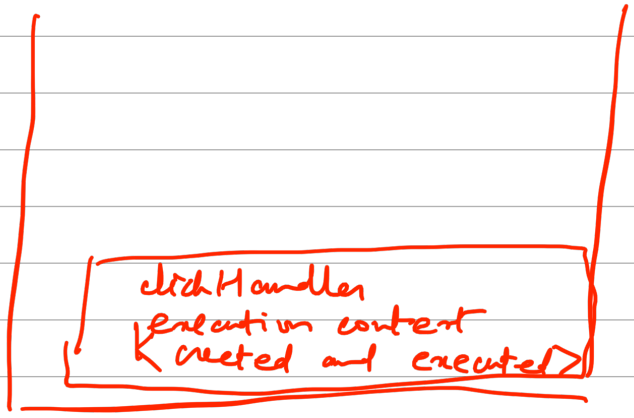
- More than one at a time
- Javascript is synchronous







It pops off the execution content one by one. And only when it is empty, it checks event queue and removes it from queue and puts the click Handler on execution stack.



q*

- Long running function can interrupt events
- It will continue to watch event queue in loop. Event loop... Process the events in the order it happened...

Dynamic typing-

- You don't tell the engine what type of data a variable holds, it figures out while your code is running
- Variables can hold different types of values because it's all figured out during execution.

Static typing- C#, Java, etc.

bool isNew = "Hello"; // an error

Dynamic typing-

Javascript figures it out on the fly...

var isNew2 true; // no errors
isNew2 'hello';
isNew2 100;

Primitive type -
- type of data the represents a single value
- that is, not an object.

undefined - lack of existence
- you shouldn't set a variable to this

null - lack of existence
- you can set a variable to this...

Boolean - true or false

Number - floating point number <there is always some decimals>. Unlike other programming languages, there is only one 'number type'... and it can make math weird.

String - a sequence of characters
<'', "" can be used>

Symbol - used in ES6 <the next version of javascript>

X

→6 primitive types
→Engine is figuring out them on the fly...

OPERATORS -
A special function that is syntactically <written> differently

Generally, takes two parameters and return one result

10*
Syntax parser and other parts of engine...

function +(a, b) {
    return // add the two no.s
}
+(3, 4) → cumbersome

Instead, we use infix notation. | Prefix
3 + 4 | + 3 4
[human readable] ↓ | Postfix
3 4 +

- Operators are functions.

Operator precedence-
- which operator function gets called first
- functions are called in order of precedence (Higher precedence wins)

Associativity:
- what order operator functions get called in:
    Left-To-Right
    Right-To-Left
- when functions have the same precedence

11*

Coercion-
- converting a value from one time to another
- happens a lot in javascript because its dynamically typed

12*

Comparison operators-

13*

- Coercion in practice...
- Can cause surprising results...

== → will try to coerce the types <Equality operator>
→ 3 == 3 // true
→ "3" == 3 // true
→ false == 0 // true
→ null == 0 // false... null doesn't coerce to 0 for == comparison
- Can cause problems that are hard to debug

→ "" == 0
→ "" == false

- Code becomes very difficult to anticipate

=== → strict equality // doesn't try to coerce the
!== → strict inequality // values...

3 === 3 // true
"3" === "3" // true
"3" === 3 // false. === says they are different types
So, they are not equal...

```
var a = 0; b = false;
if(a == b) {
    console.log("They are equal");
} else {
    console.log("They are not equal");
}
if(a === b) {
    console.log("They are equal");
} else {
    console.log("They are not equal");
}
```

→ use === when making equality comparisons...

We are observing the coercion is bad. Then is there any use case for that?

14*

- Boolean(undefined) → false
- Boolean(null) → false
- Boolean("") → false
- Boolean(0) → false

They implies the lack of existence

Coercion is useful here...

15*

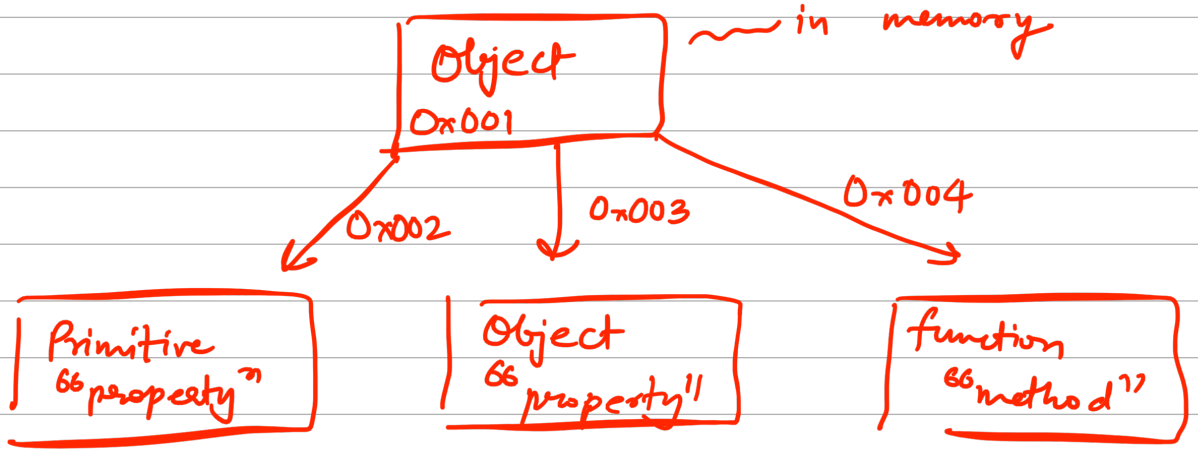
|| operator coercision.

Default values-

16*

Objects and functions→

- In javascript they are pretty much related
- Object can have properties and methods



Accessing those properties and methods?

17* →

- Prefer using the . operator. It's neat and clean.

18* → Object literals

Namespace -
- container for variables and functions
- trying to keep variables and functions with the same name separate

In javascript, there is no concept of namespace. We need to fake it.

19*

JSON - - Javascript Object Notation
- inspired from object literal.

20*

Why wrapped in quotes?

°o° to make it simpler and to avoid having another method to sanitize it from reserved javascript keyword.

JSON → {

var : 123

}

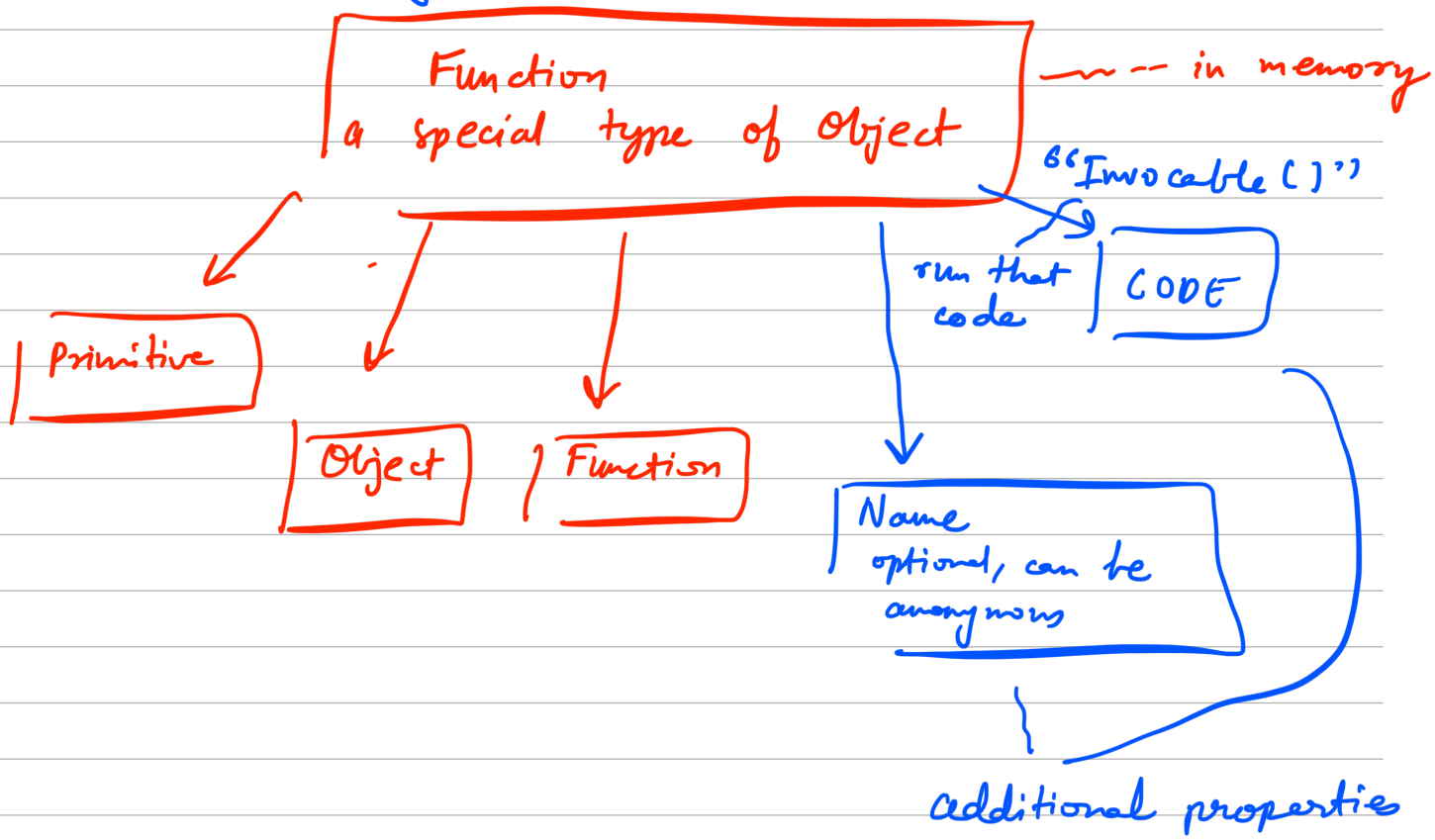
↓

a reserved keyword. Then we would need a mechanism to tell that hey! it's not a keyword but a key in JSON... phew!!!

fundamental and important concept in javascript—

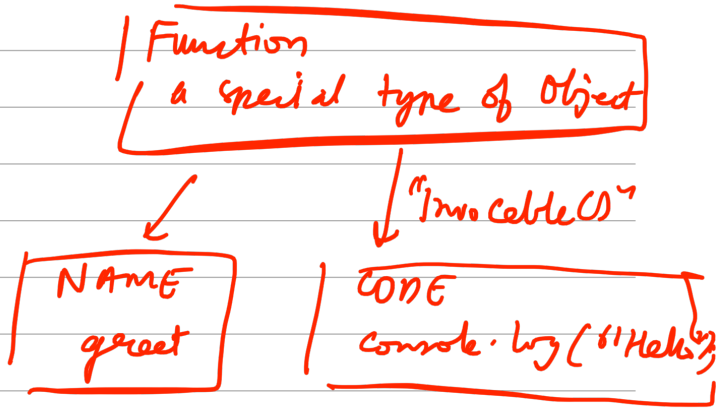
- first class functions →
  - Everything you can do with other types, you can do with functions
  - assign them to variables, pass them around, create them on the fly...

function are objects in javascript



21*

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



In Javascript,

functions are objects.

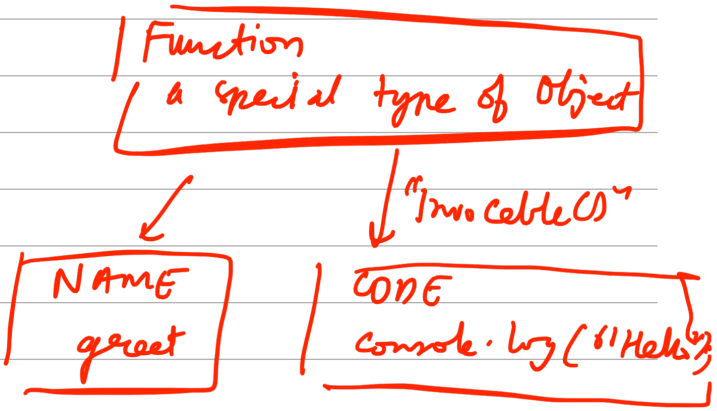
- functions are more than something that only executes code.

Expression -
- a unit of code that results in a value.
- it doesn't have to save to a variable

22*

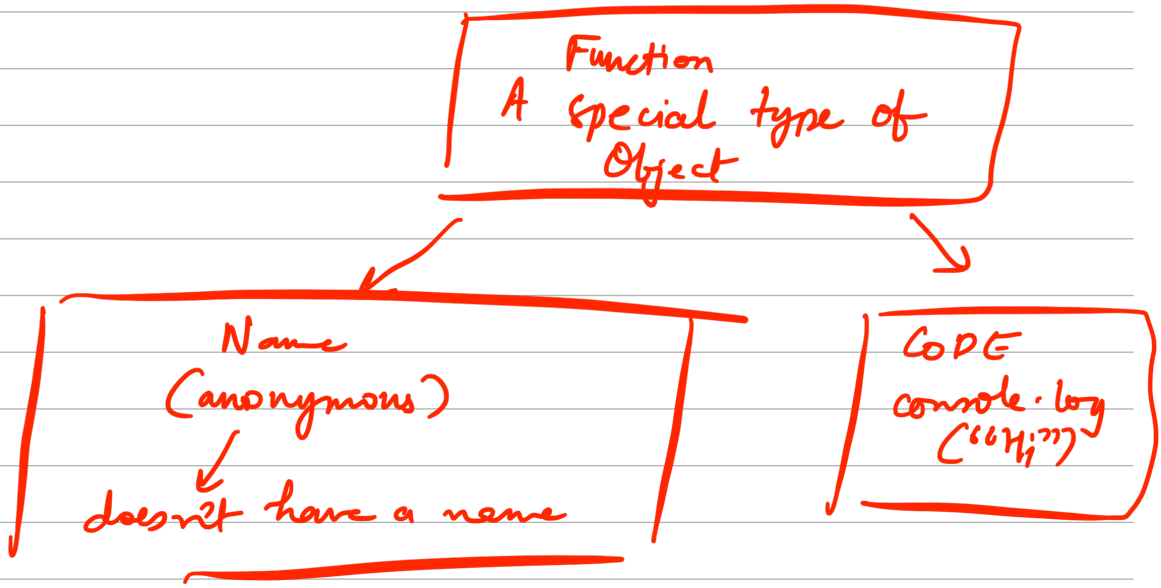
1+2
3

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



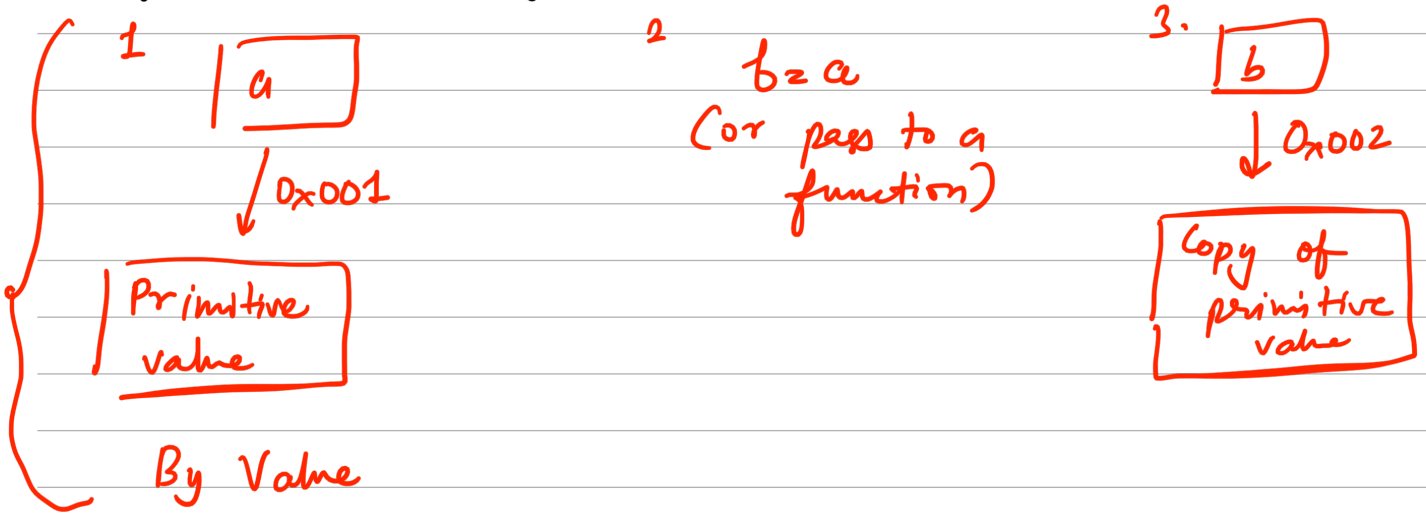

```
var anonymousGreet = function() {
    console.log("Hi");
};
```

// function expression
∴ it results in a value



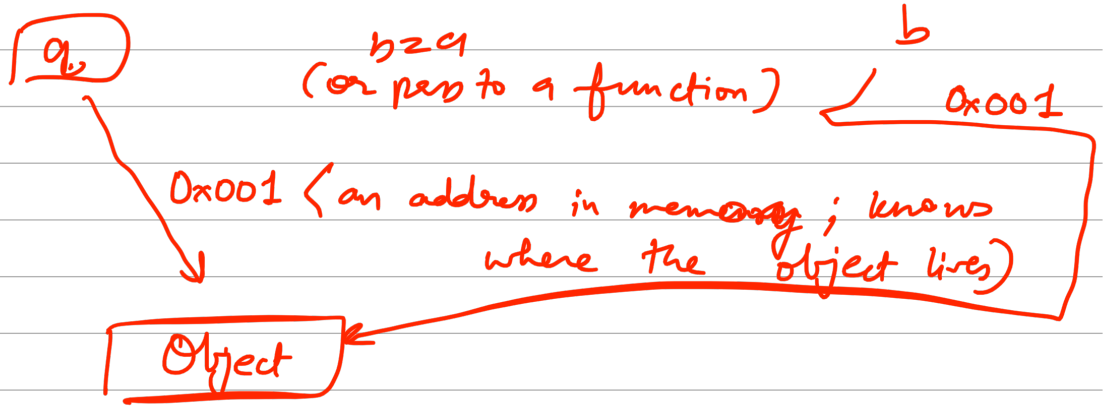
anonymous Greet();

By Value vs By Reference–



By Value

Two separate spots in memory.

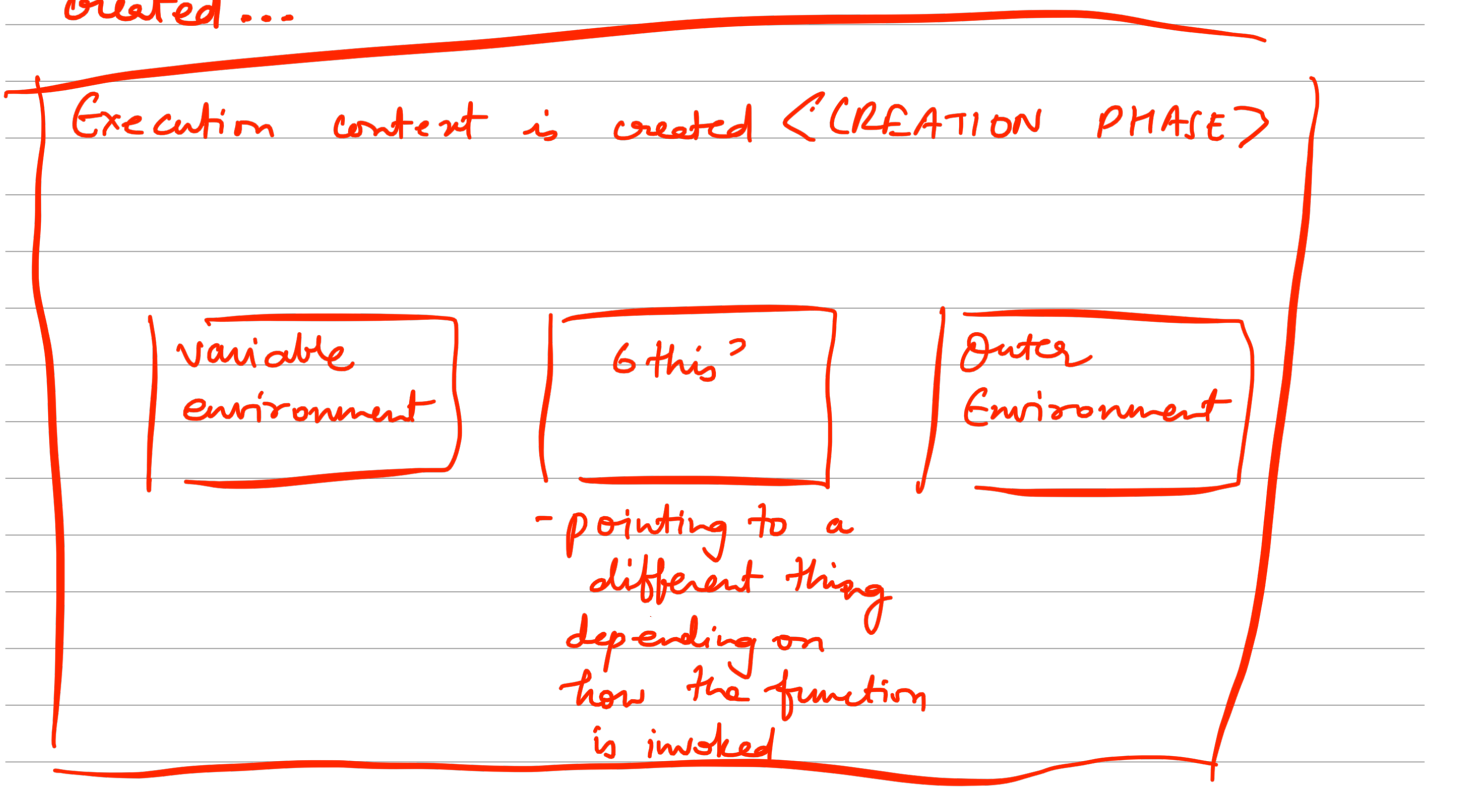


- like an alias
- a and b have the same value
- By reference
- all objects behave by reference way..

23*

Objects, functions and this-

When a function is executed, an execution context is created... <running the code in the code property>



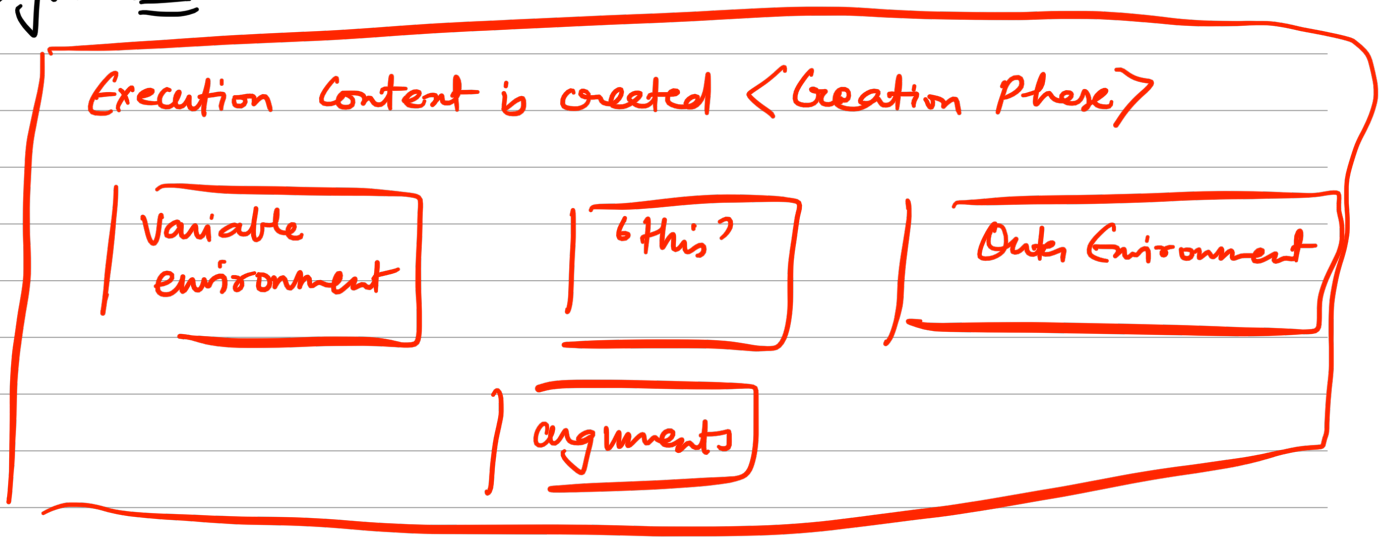
24*

Arrays-

- Collections of anything

25*

Arguments-



- The parameters you pass to a function
- Javascript gives you a keyword of the same name which contains them all...
- Javascript engines sets up 'arguments' for us.

26*

Function Overloading-
- Javascript doesn't have function overloading
- In javascript, functions are objects.
- Having first class functions, gives us a lot of other things

27*

Syntax Parsers-

Code → translates → Program → Computer

- different aspects.
- One of them is syntax parser

V ← ← r

- character by character to check if its valid... using a set of rules... and deciding what is it that you want to do?

Syntax parsers-

- Automatic semicolon insertion-

r ← e ← t ← u ← r ← n ↓ (carriage return)

It's actually a character.

Syntax parser inserts a semicolon there...

- Always put your own semicolons...

- In case of return, semicolon automatic insertion can cause a problem

28*

White space -
- Invisible characters that create literal 'space' in your written code

- Carriage returns, tabs, spaces.
- Javascript is very liberal about whitespaces

29*

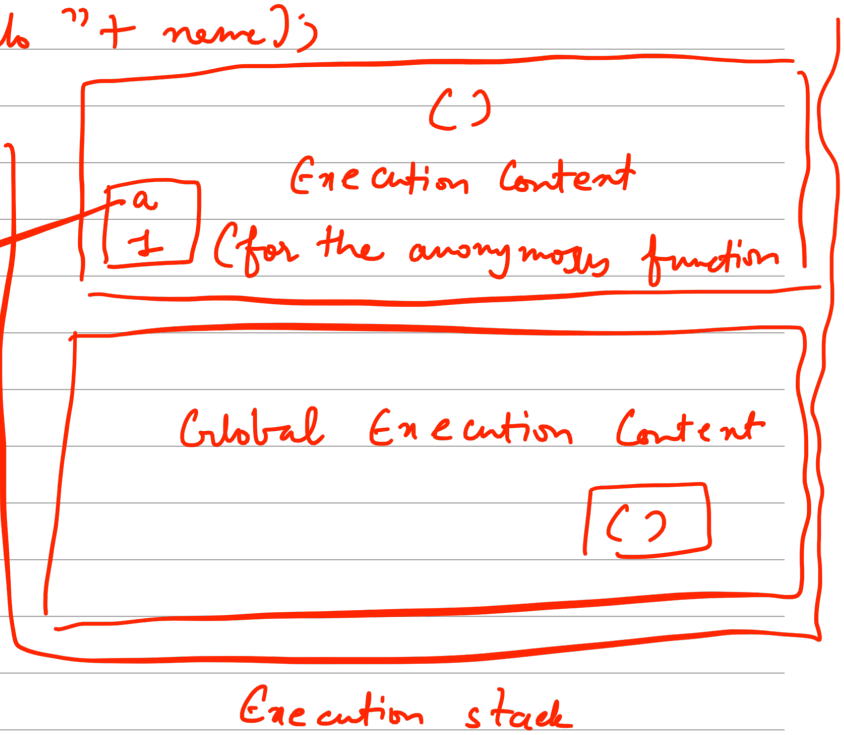
(IIFE)s - Immediately Invoked Function Expression -

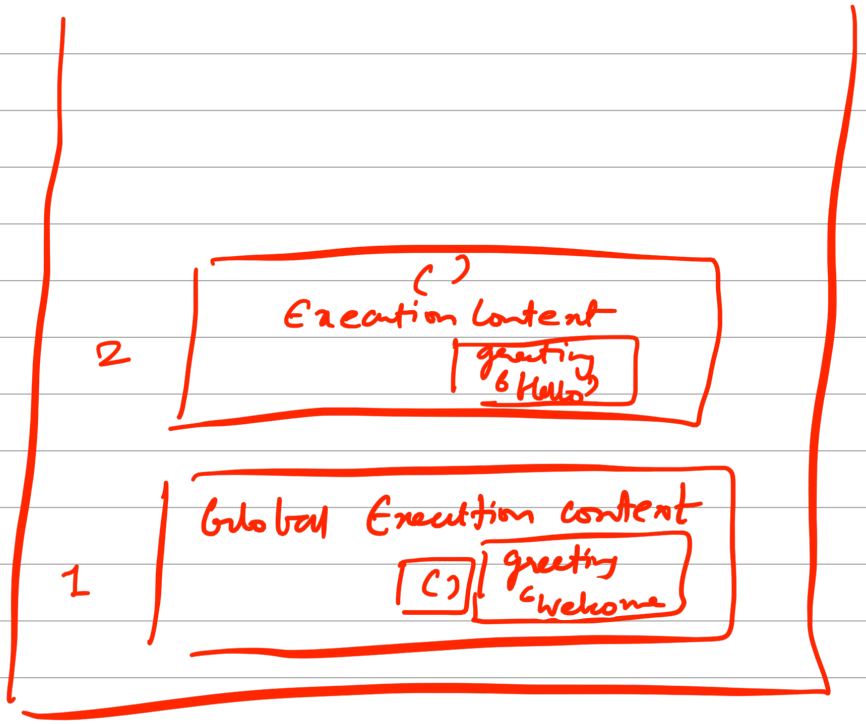
30*

31*

```
(function(name){
    var a=1;
    console.log("Hello " + name);
})("Ami");
```

→ no interference for this a
→ not putting something into the global object. Doesn't pollute the global object




2
Execution Context
greeting
"Hello"
1
Global Execution context
greeting
"Welcome"