JAVASCRIPT Execution Context

(The execution behind your javascript code)

There are always certain factors which are a given when you execute your Javascript files

Here's the crux of the Matter:

(*) $\boxed{\{\}}$ → Global Execution Context (GEC)

↳ whenever you write some Javascript, A global EC is defined.

'this' refers to the GEC.

Remember we tried printing the GEC ~~out~~ through 'this' keyword in node? We got an empty object.

GEC depends on the runtime environment of JS.

For browsers, window is the GEC.

(*) (2) → Function Execution Context

- Yes, Functions have a different EC.
- Basically why 'this' prints a lot of context when logged inside a function.

(3) → Eval execution context

A property of GEC.

Good to Know Kind of Stuff

Okay Sherlock, Now tell me How is my code actually executed (-_-)

Alright, Let's See

After the global EC has been created

JS engine executes your code in two phases/cycles

① → Memory creation phase

what does the name tell you? Exactly, That's it.

I'll write a code down, we'll refer to it and allocate memory, we are the JS engine for now :P

```
1   let var1 = 10
2   let var2 = 5
3   function addTwo (var1, var2) {
4       total = var1 + var2
5       return total
6   }
7   result1 = addTwo (var1, var2)
8   result2 = addTwo (10, 6)
```

Now Pay close attention,

1. line 1 encountered, define a var & allocate memory
   var1 = undefined
2. line 2 encountered, define a var & allocate memory
   var2 = undefined
3. addTwo, This is a function, let's store the function definition
   addTwo = func^n definition
4. line 7 is result1, A variable, allocate memory & initialize
   result1 = undefined

5. line 8, Another result variable, Allocate memory & initialize
result2 = undefined

(*) okay by this time you might be having a few questions, Let's answer them

(0) why
A] Next question

(1) This GEC you're talking about, IS created before the creation phase or during?
A] During the creation phase, And it is the first thing that is created during the creation phase

This is also called 'Global object setup' and 'this' Keyword refers to the global object
This object is the ultimate, The Shiva of JS.
It is the outermost object in the scope and holds all the global variables & scopes

(2) Why undefined to variable & function definition to functions?
A] To answer it shortly; Javascript's choice & some logical reasons
Reasons?
→ If not undefined, then what?
Remember, No execution has occured yet. JS doesn't know what you're defining in the first phase.
So, It takes a safer bet and assigns undefined.
→ A clear default value
undefined clears all confusion, Allows you to distinguish ~~values~~ ~~and~~ that are declared but not yet assigned a value variables
→ Using other values like 0 OR null can be a pain in the ass
Suppose a student scored hundred. & you assigned & printed a 0 instead, ffs (-_-)

what about the functions then?

A] functions are assigned their definitions. why? what in the fucking world is that question?
What else would you assign to it? null? 0? undefined?
This will f*ck up, the function calls all around the scope, No response then.

(*) Have you heard of hoisting?
All declared functions and variables are moved to the top of your scopes, I mean their respective scopes in the creation phase itself.

So no matter wherever you call em, They will respond if the call is in their scope.

(3) what does a function definition contain?
Interesting, The following stuff:

i] Function name
ii] Parameters
iii] function body

Okay! That clears things up!
At the end of phase 1, we have

var1 = undefined
var2 = undefined
addTwo = function definition
result1 = undefined
result2 = undefined.

Execution [phase two]

```
let var1 = 10
let var2 = 5
function addTwo(var1,var2) {
    total = var1+var2
    return total
}
result1 = addTwo(var1,var2)
result2 = addTwo(10,6)
```

Your declared variables are ready and the code is ready to be executed now.

Let's see,

Val 1 ← 10

Val 2 ← 5

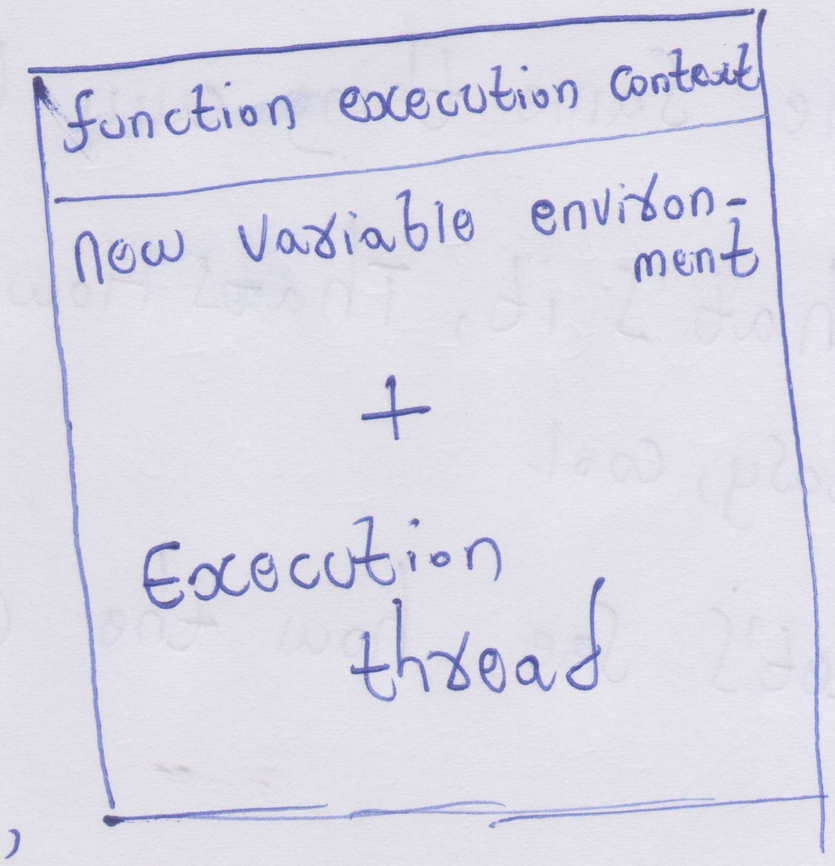
function? we already noted the definition, Go ahead

result1 ... Here's where things get interesting

This is a function Call, And JS engine recognises it.

Here's what happens, A new execution context is created.

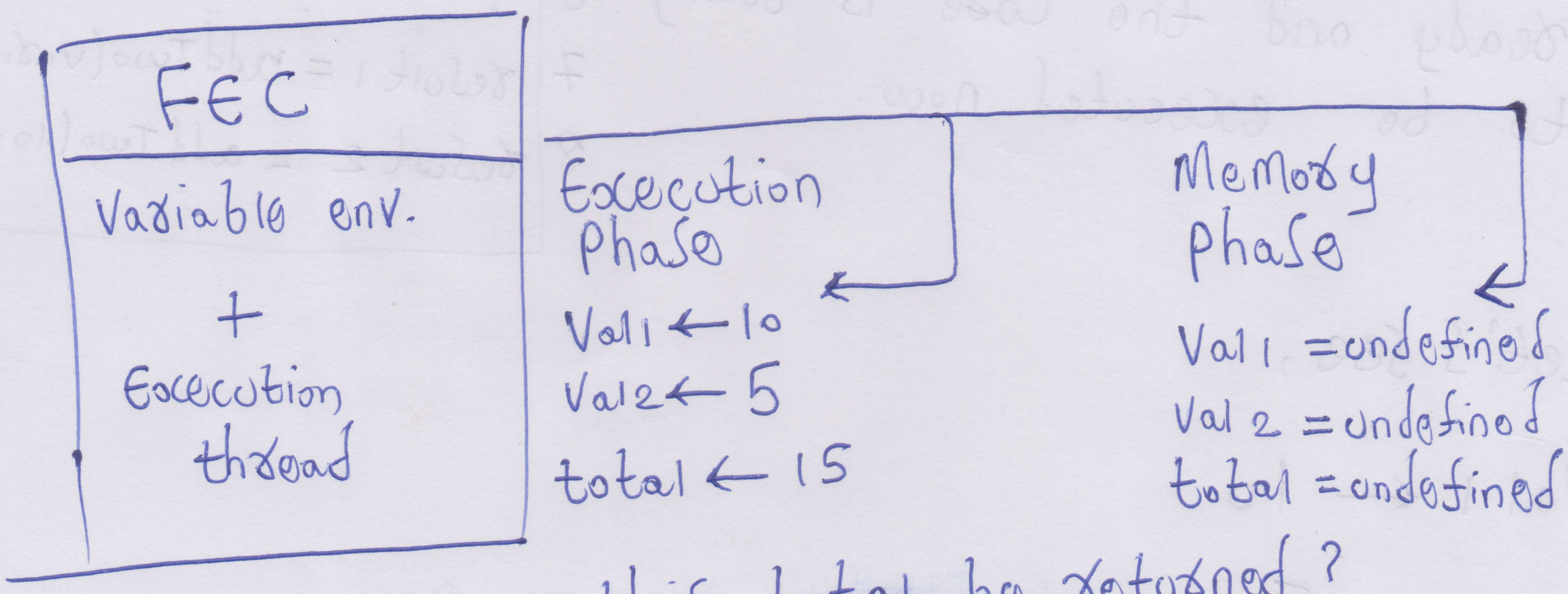
result 1 → Calls addNum →



and this function execution context (FEC) is created every-time a function is called.

(*) Good to Know: Don't you think, For variables & functions inside FEC, FEC win actually be the Global context? Think of it as a mini global context, This is the reason why this behaves differently in functions. Context is different!

Then the Ramayana starts again, Again the memory phase and execution phase for the function context



(*) Now where will this total be returned?
To its global execution Context, Basically the parent Context

And what happens to the FEC post returning?,
It is deleted, existence is erased, as soon as the value is returned

The same thing will be done for all function calls.

That's it, Thats How javascript runs your code.

Easy, cool.

Let's see how the call stack works

CALL STACK

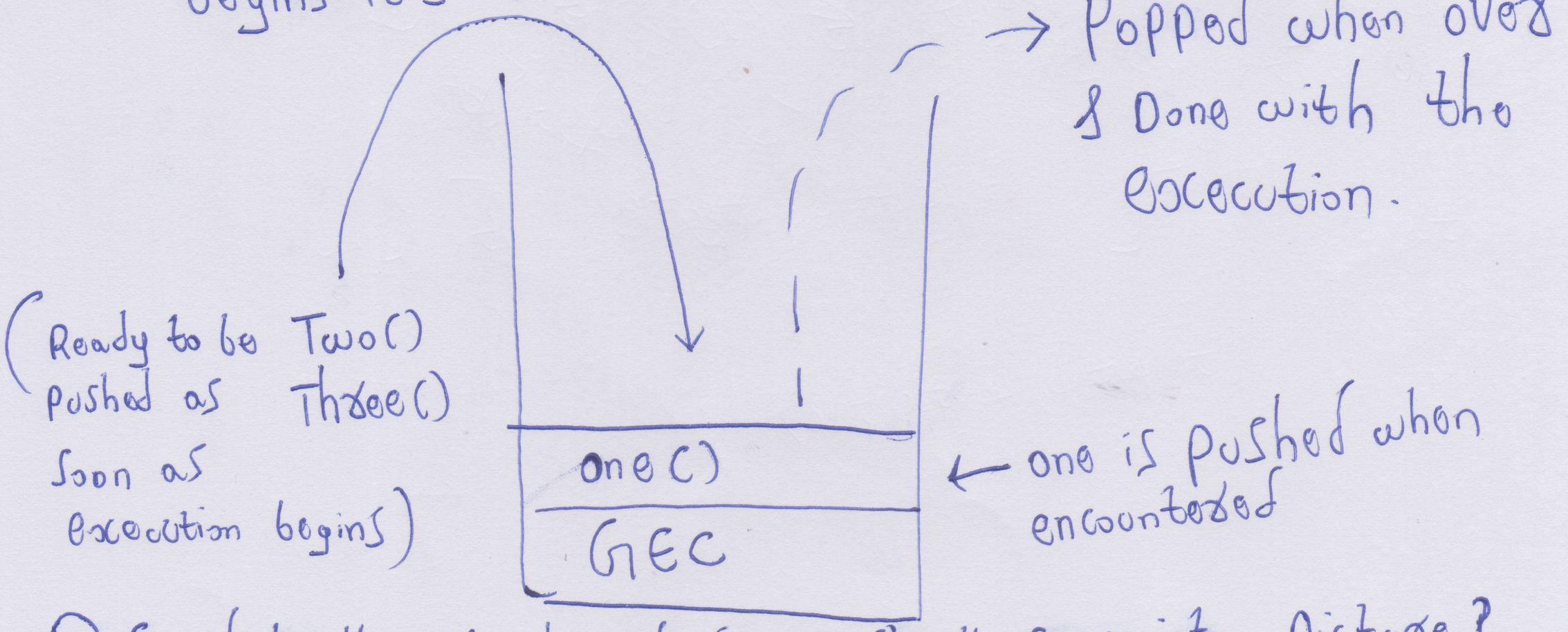
(No function escapes my watch)

Let's take a small example

```
function one() {
  ...
}
func two() {
  :
}
func three() {
  :
}
```

(*) you already know how recursion works? You know how call stack works then!

Execution phase starts, And the call stack begins it's work



(*) Good to Know: when does call stack come into picture? It's present In both the phases, but is mostly (totally) utilized in the execution phase.