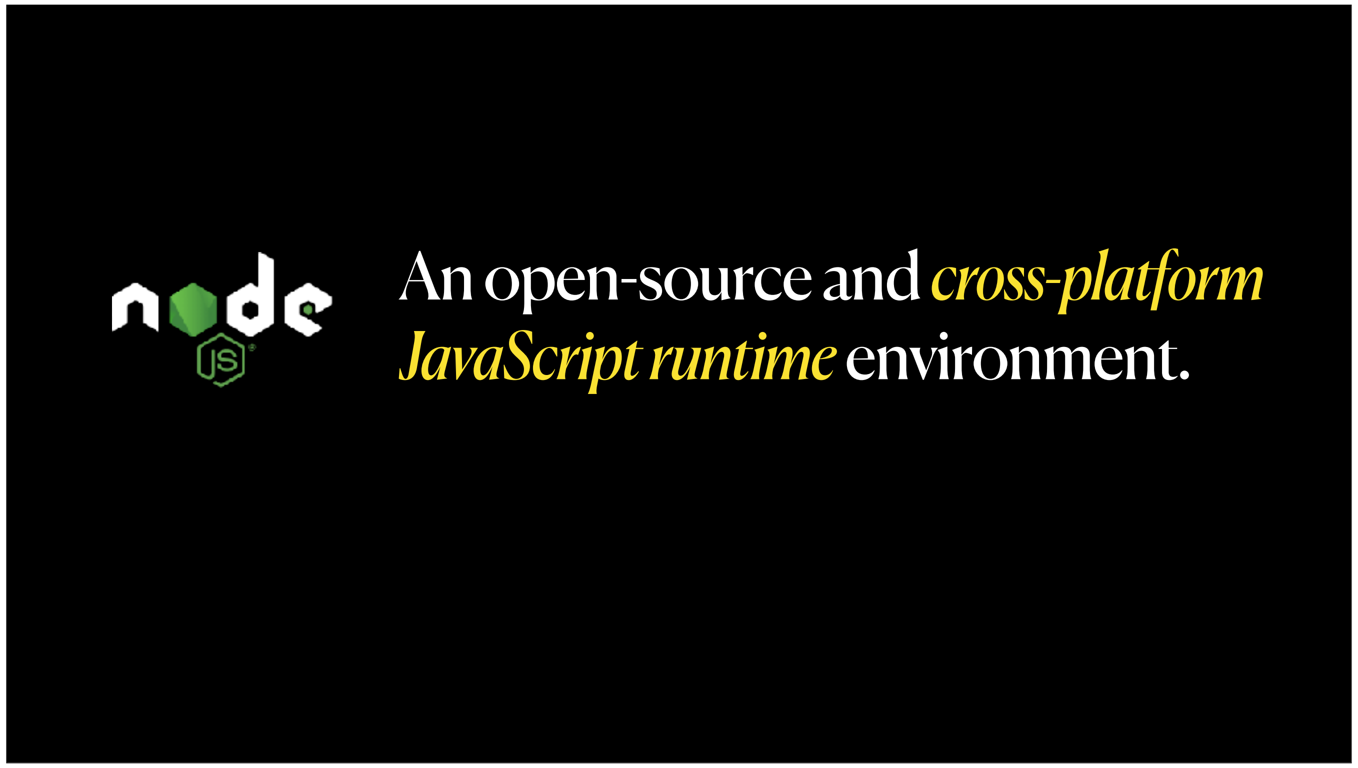
Presented by

CSE-23-16 CSE-23-28

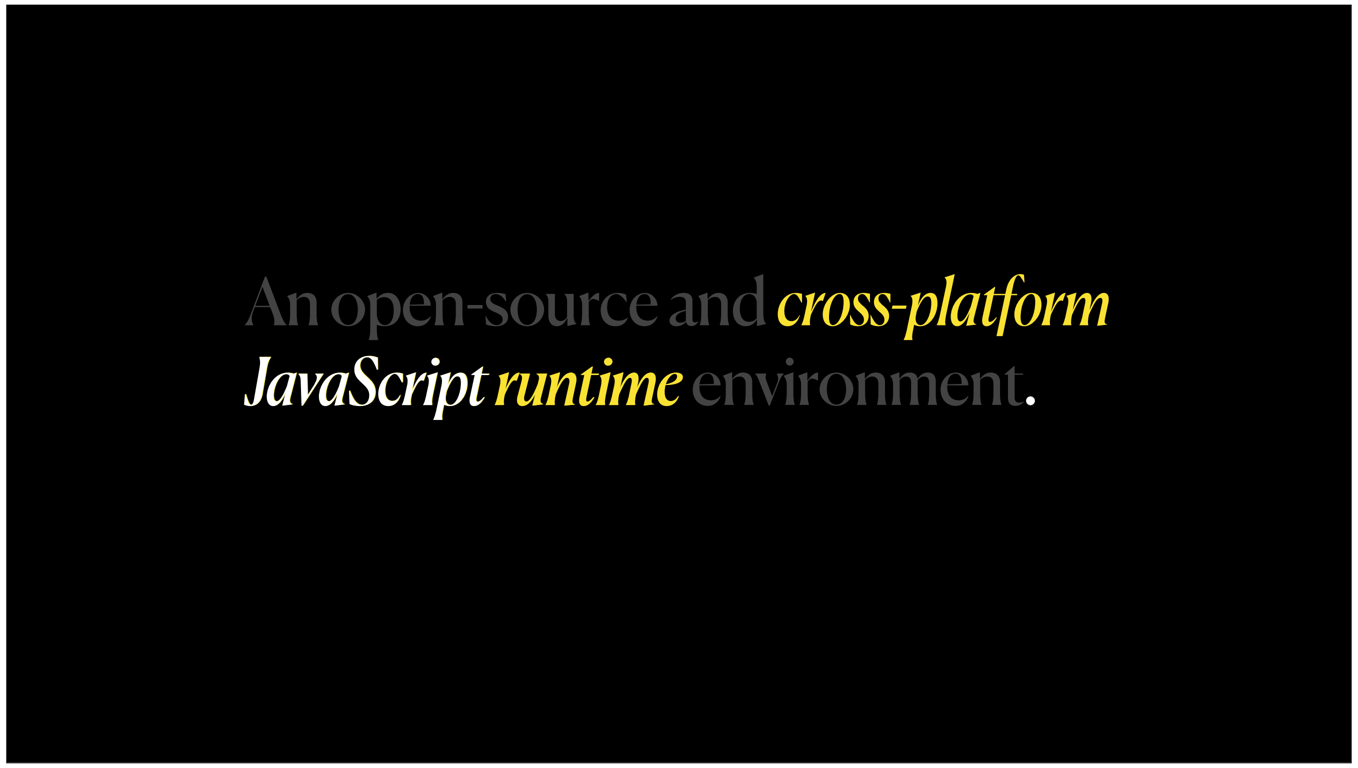
CSE-23-55 CSE-23-68

Greetings everybody;
>> Myself MohammadAzeem, and I will be introducing NodeJS



An open-source and *cross-platform JavaScript runtime* environment.

So it is An open-source and cross-platform Javascript runtime.
>> Which means,

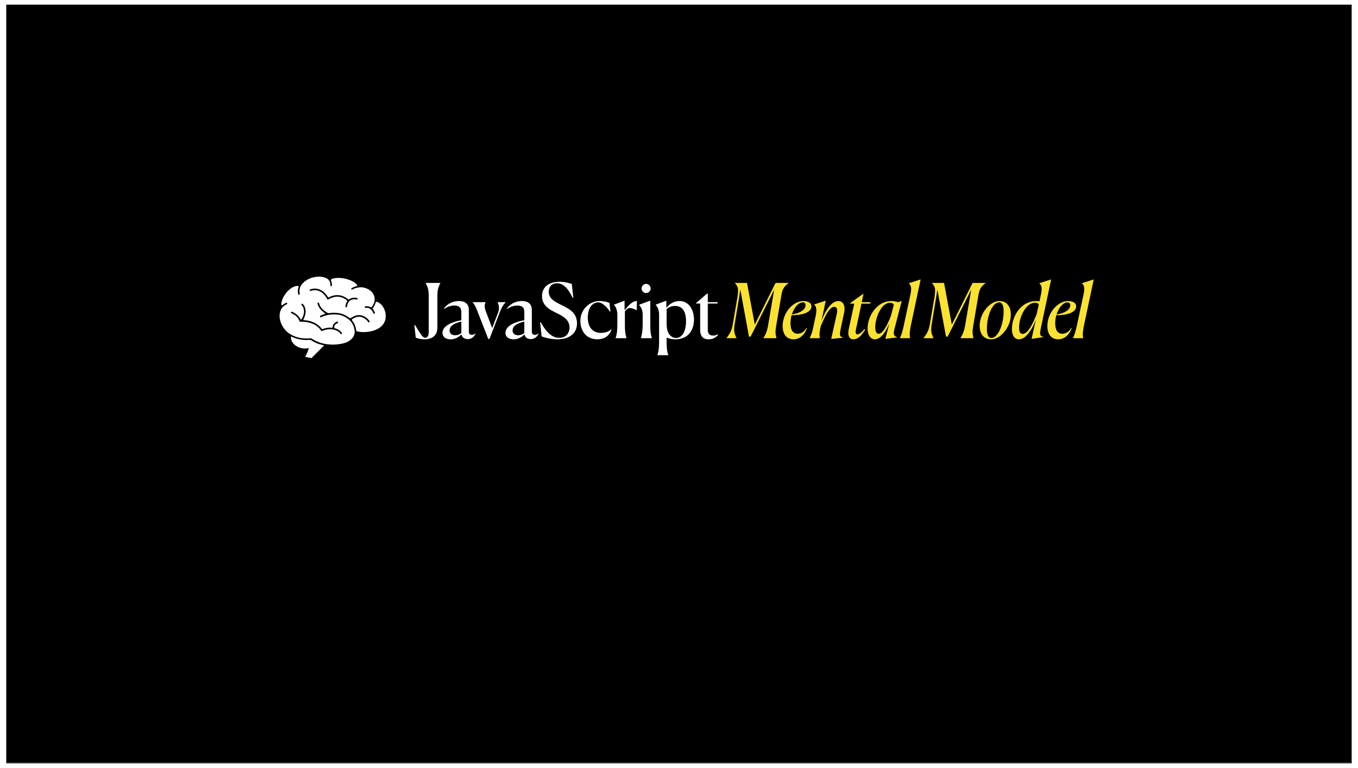

An open-source and *cross-platform*
JavaScript *runtime* environment.

whatever this is, it can be used on any platform; be it ARM, x64 (64bit), x32. And results will be consistent.
Now what it actually is, is a Javascript runtime.
>> Which means

JavaScript runtime

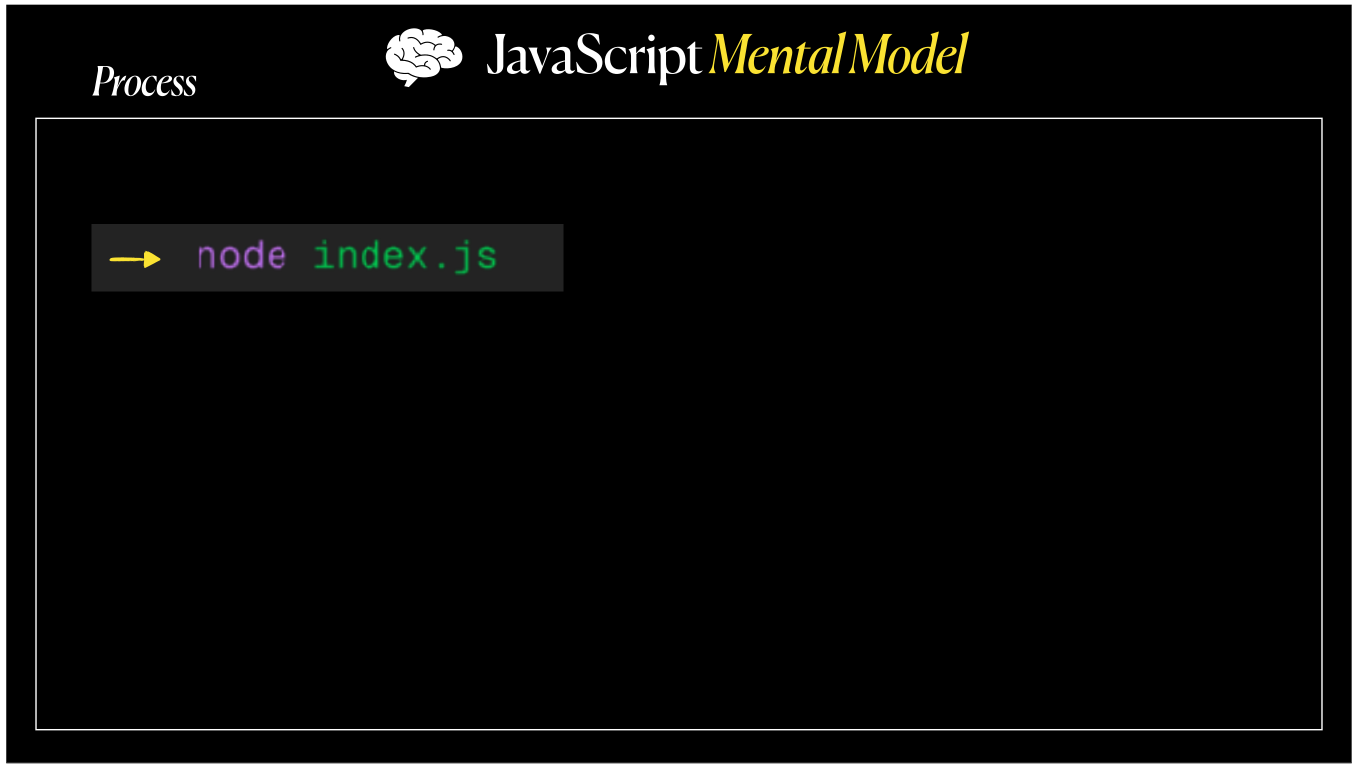
Is simply a computer program that executes JavaScript code. It's responsible for translating human-readable JavaScript code into machine-readable instructions that the computer's hardware can execute.

[read def]
JavaScript has multiple implementations, unlike most programming languages that typically have one primary version. While languages like Python usually run on the official interpreter (or binary) provided by their maintainers, JavaScript runs on various engines across different platforms - V8 powers Chrome and Node.js, SpiderMonkey runs in Firefox, and Safari uses JavaScriptCore.
>> Now we will go in a [em] little detail



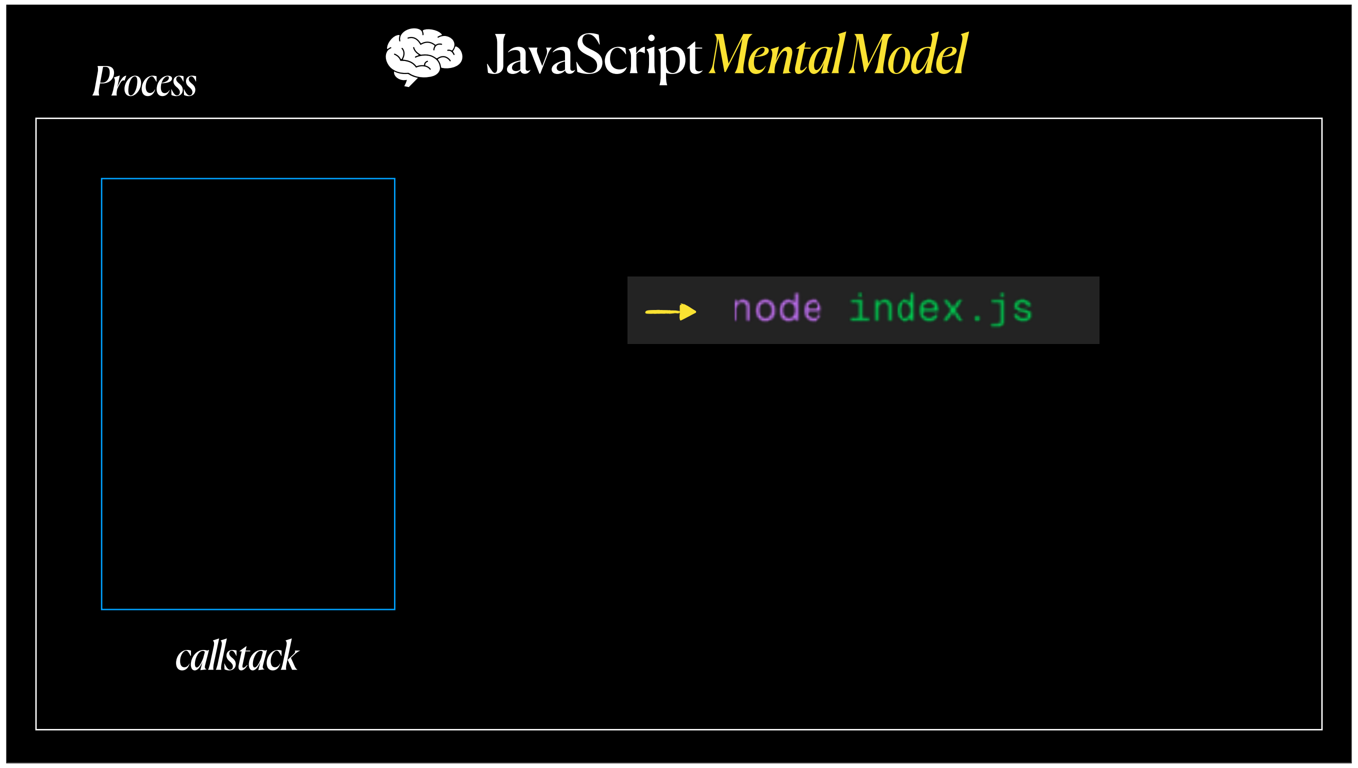
JavaScript *Mental Model*

Of how javascript is executed, and along the way, we will develop so called a *Mental Model* of the same. Now I promise that this will befit you; weather you are a beginner or a seasoned JS developer. So keep your ears up.
>> When a



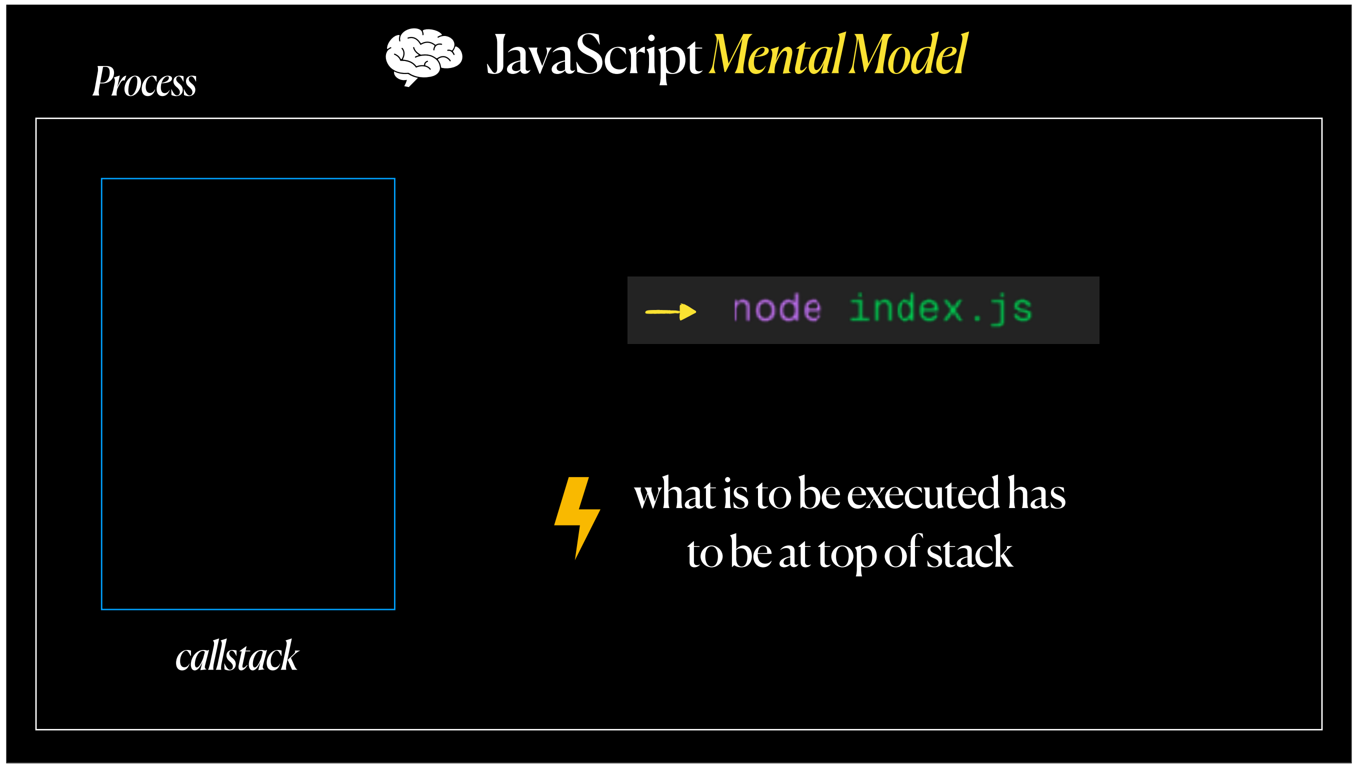
```
→ node index.js
```

JS program (file) is executed using nodeJS,
(when you do `node index.js`)
A process at the OS level is started. It is the same process as is created when you open any program (say notepad) and has a proper ProcessID of it.
>> When we hit enter on this, what nodeJS does is



```
node index.js
```

Creates a callStack
A call stack is a mechanism to keep track of what function is now to get executed.
And stack itself says, what is the last in (top of stack) is the first to get executed.
>> No*wwwwww* this might give you an idea



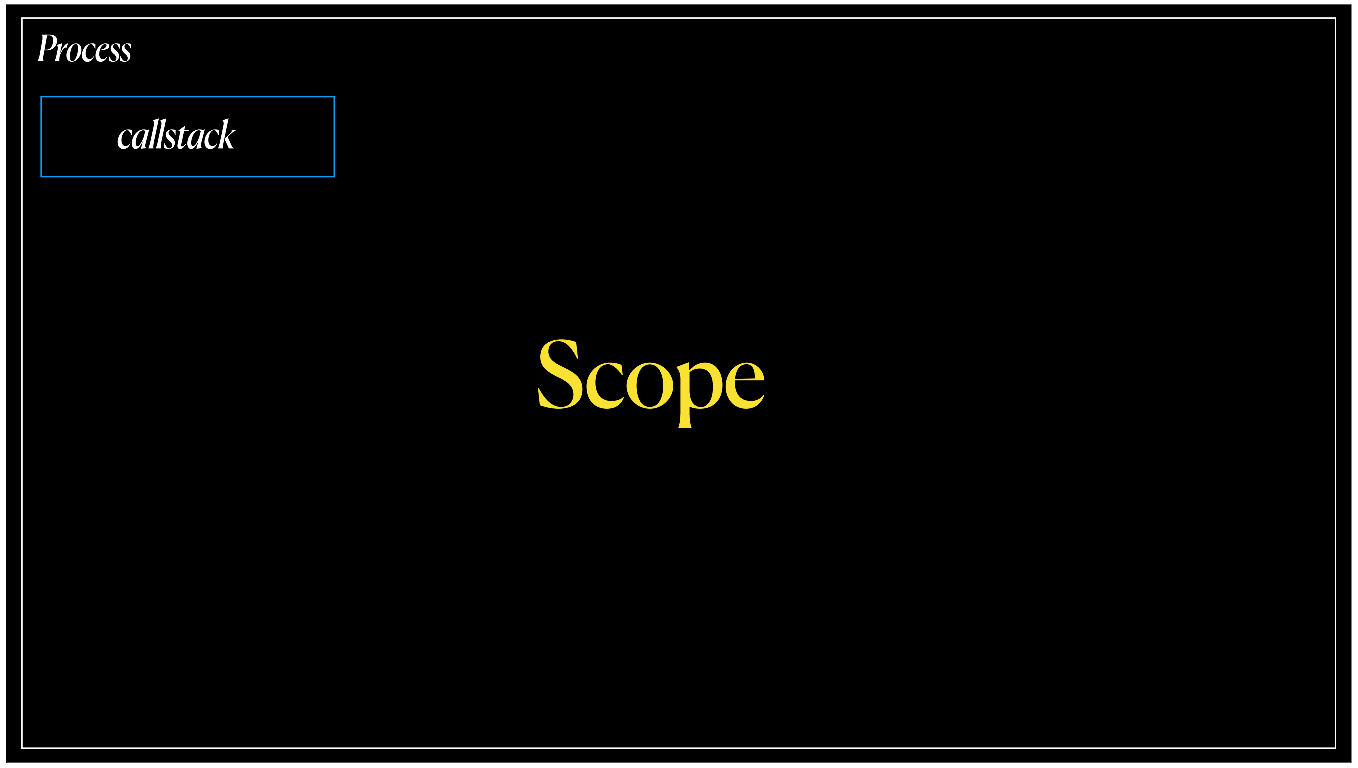
JavaScript *Mental Model*

Process

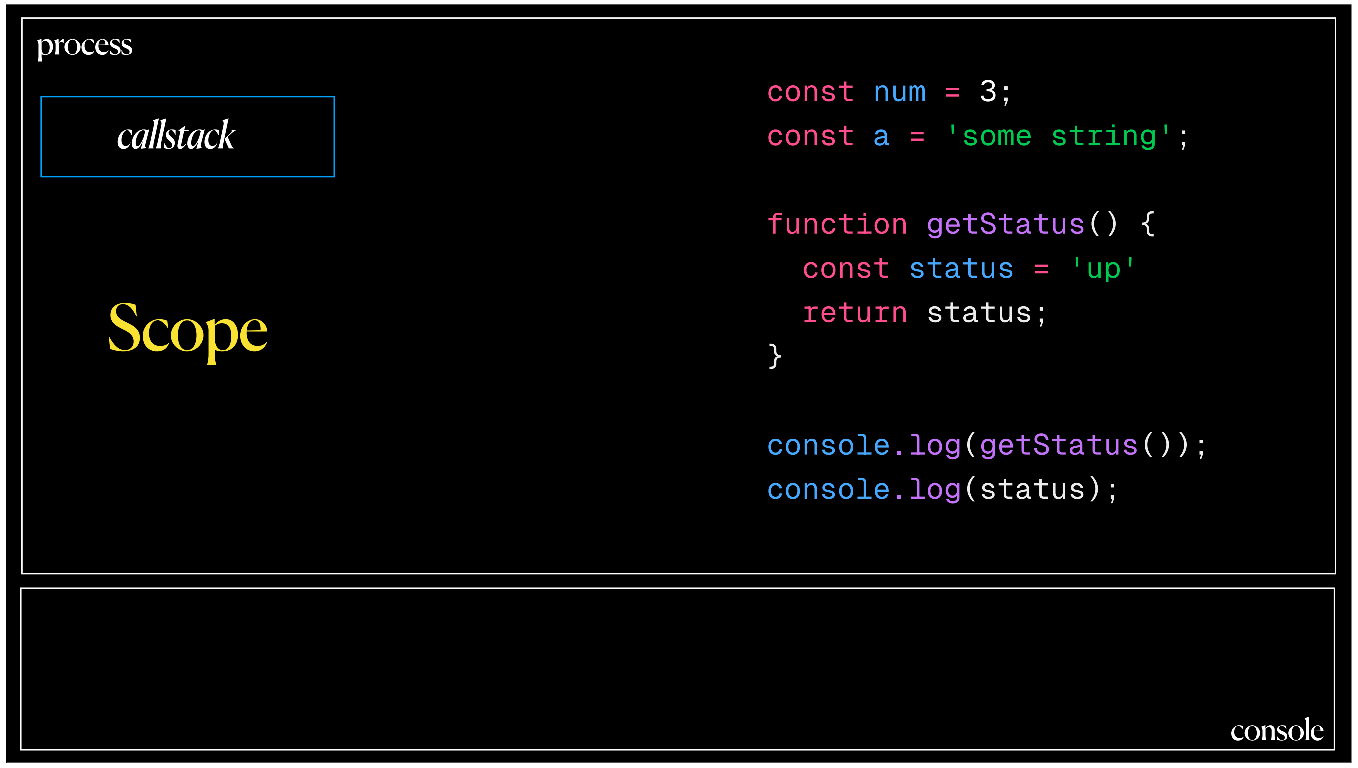
```
→ node index.js
```

what is to be executed has to be at top of stack

that all we need now is to make sure “what is to be executed has to be at top of stack”
>> Now that we know what call stack is; lets move on to scope.

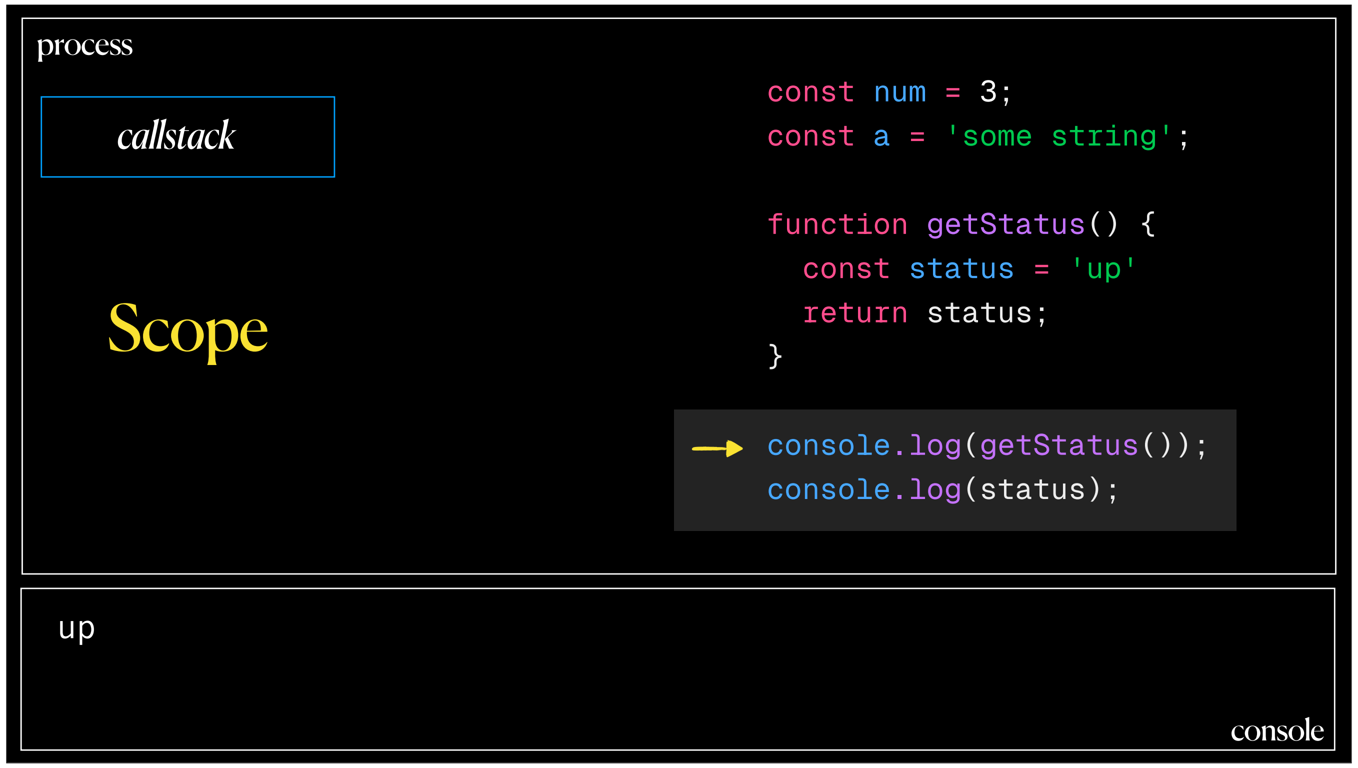


Sope is the memory allocated to a specific or currently executing function or block of code.
>> To get an idea of it, lets look at an example.



```
const num = 3;
const a = 'some string';

function getStatus() {
  const status = 'up'
  return status;
}

console.log(getStatus());
console.log(status);
```

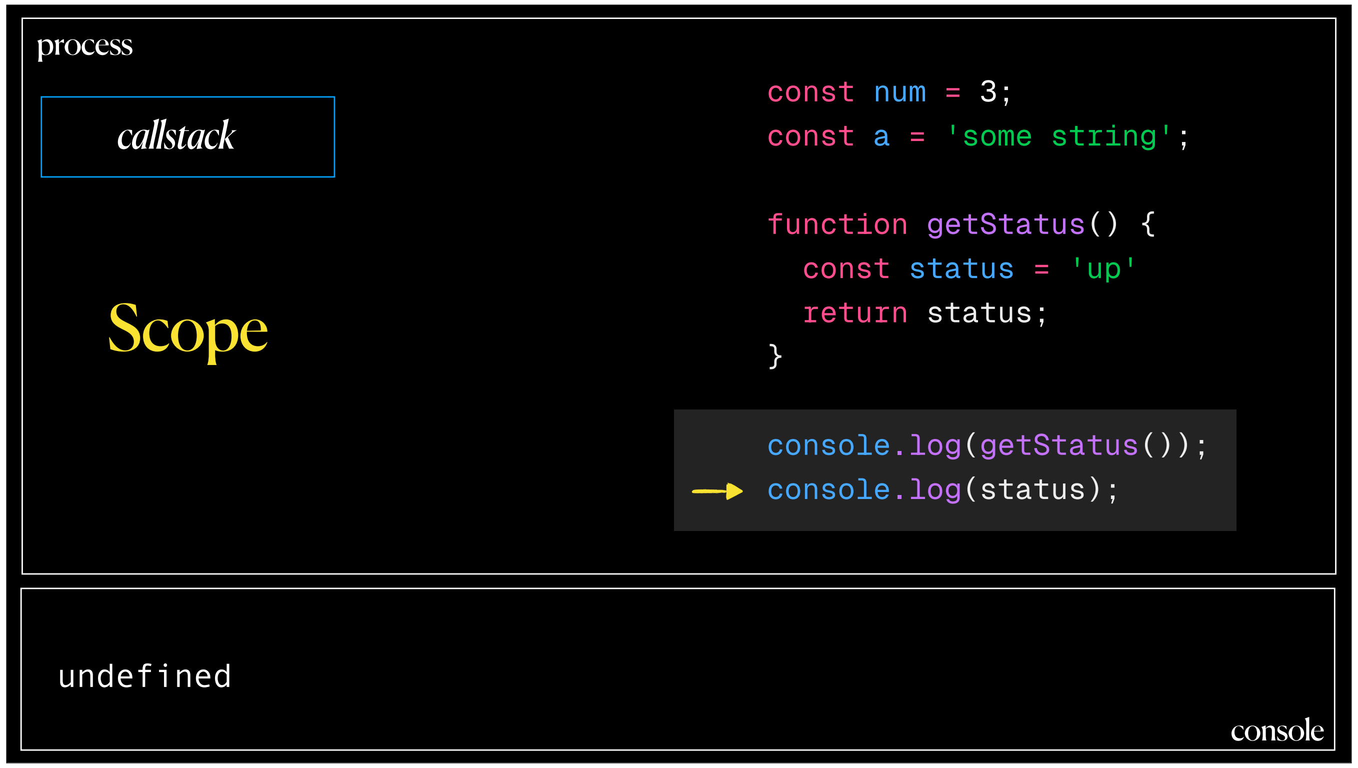
consider this is the code in index.js file we just ran;
When we reach the bottom most line,



```
const num = 3;
const a = 'some string';

function getStatus() {
  const status = 'up'
  return status;
}

console.log(getStatus());
console.log(status);
```

```
up
```

we see in console that the result is ‘up’ as expected.
>> and moving to the second console.

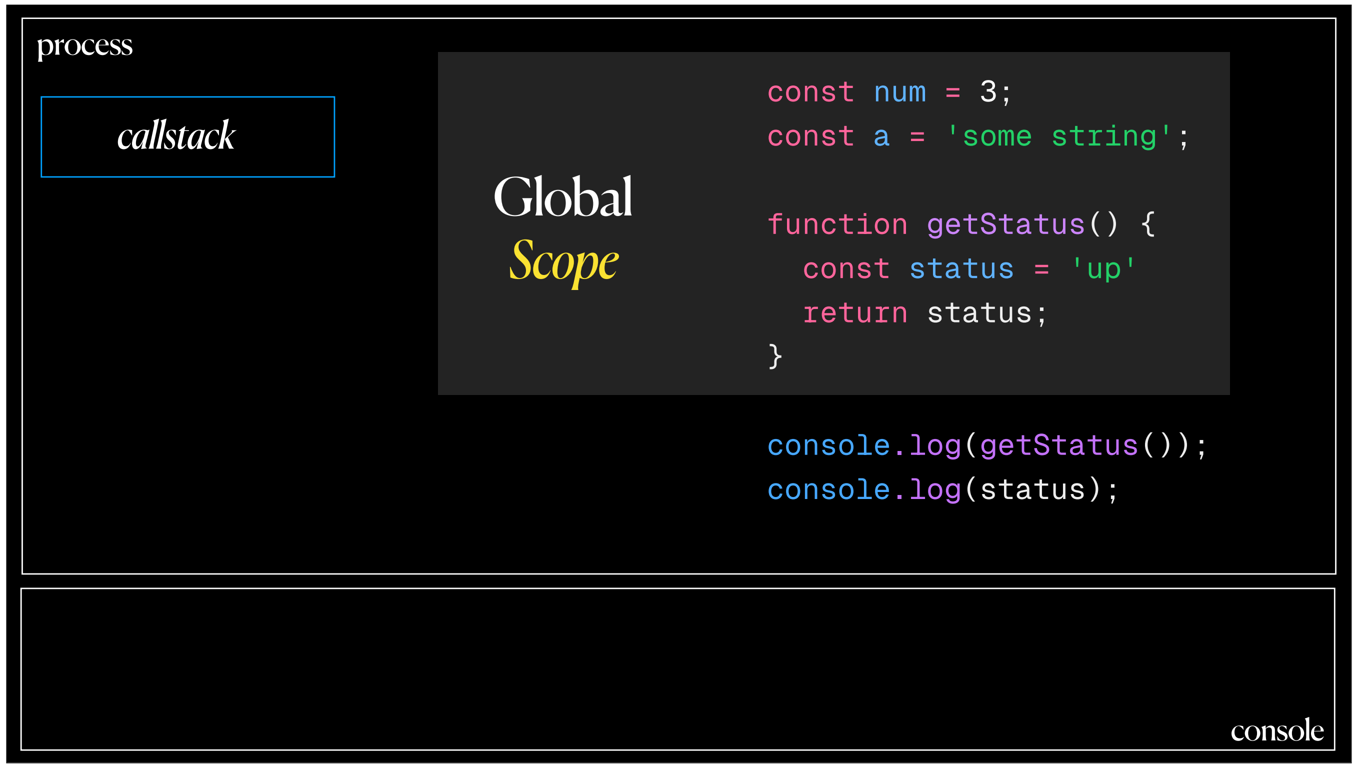


Scope

```
const num = 3;
const a = 'some string';

function getStatus() {
  const status = 'up'
  return status;
}

console.log(getStatus());
console.log(status);
```

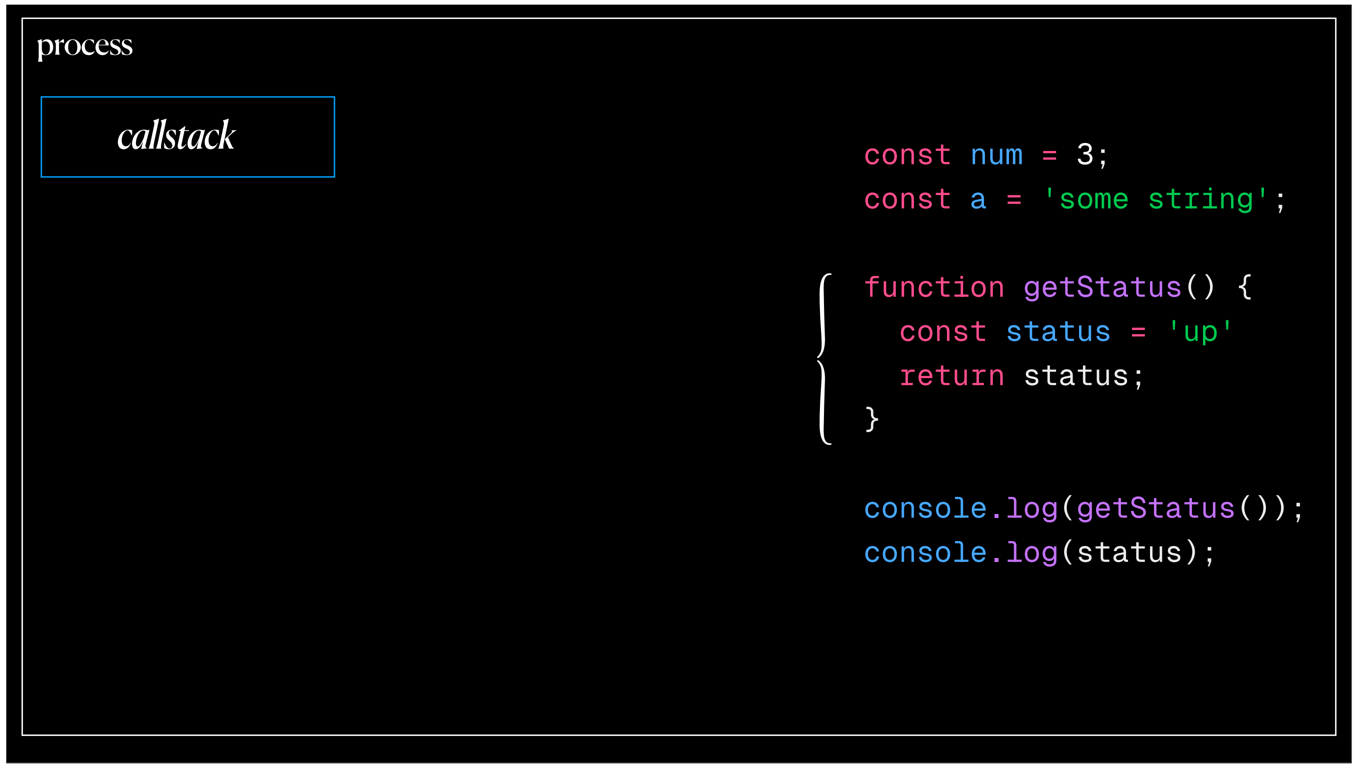
```
undefined
```

We get undefined; Why might be that??
I don't expect you to know, why is this *undefined* as this was some code that we just ran and got undefined (In fact that is some thing we will go in detail);
Rather it should come from intuition that status is just defined inside the function and it should be the one to use it
>> (see easy stuff..)



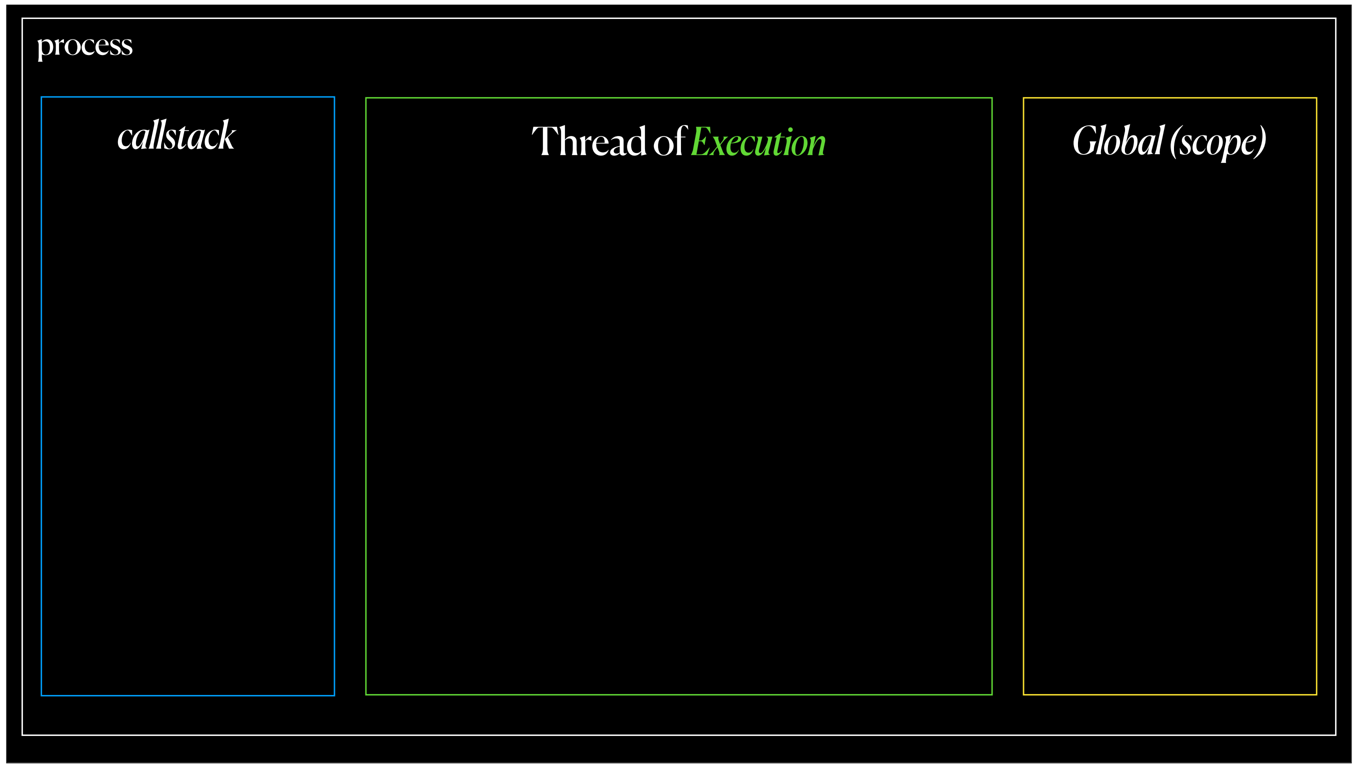
```
const num = 3;
const a = 'some string';

function getStatus() {
  const status = 'up'
  return status;
}

console.log(getStatus());
console.log(status);
```

One type of scope is *Global scope* that any one can access;
>> What can the other scope be?
Right now it is of the function; but if you truly think about this: “what ever will get executed will create its own scope” like we saw in case of function: they try to keep things to themselves;

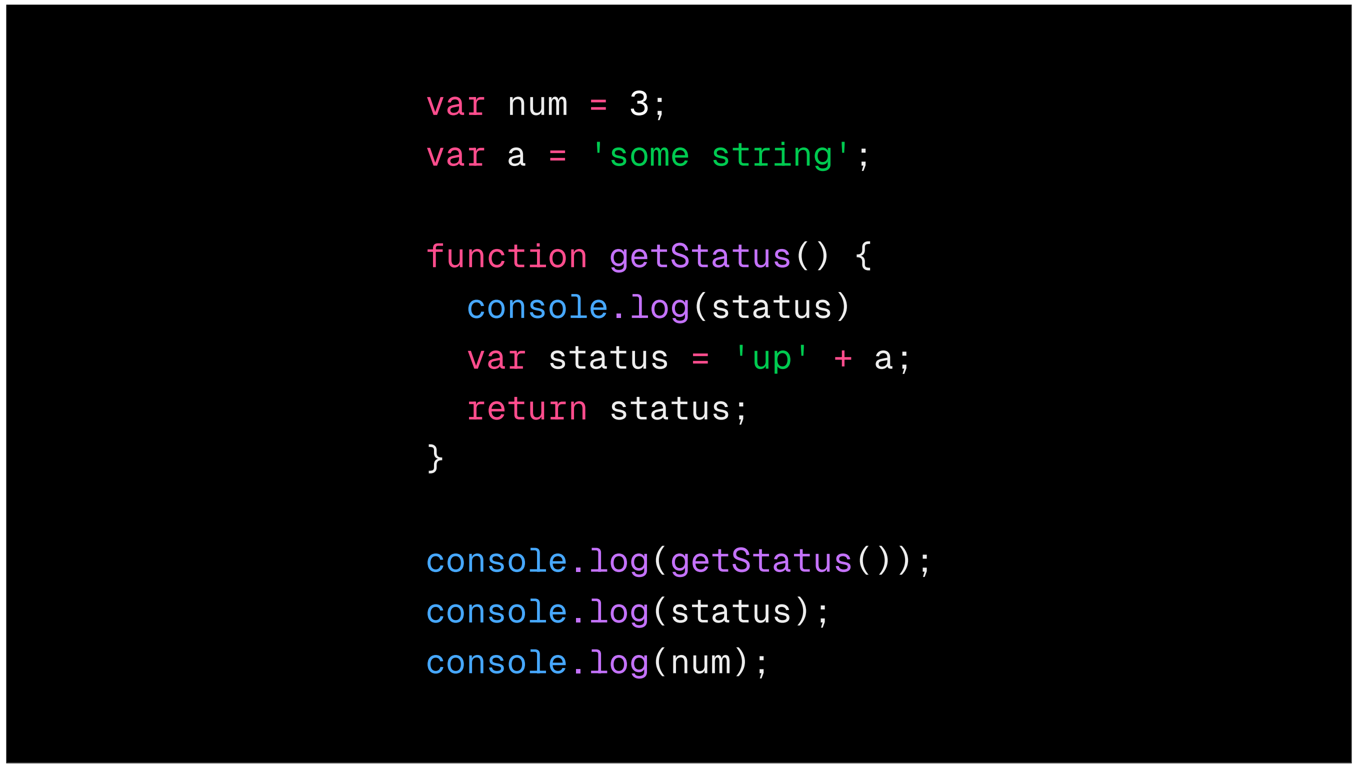


```
const num = 3;
const a = 'some string';

function getStatus() {
  const status = 'up'
  return status;
}

console.log(getStatus());
console.log(status);
```

So what ever this second type of scope is,
for sure needs its own “some amount of memory”[click] and “some set of instructions”[click] and this my friends is what we call as “Thread of execution” OR “Current Execution context”
>> Now lets have a look at

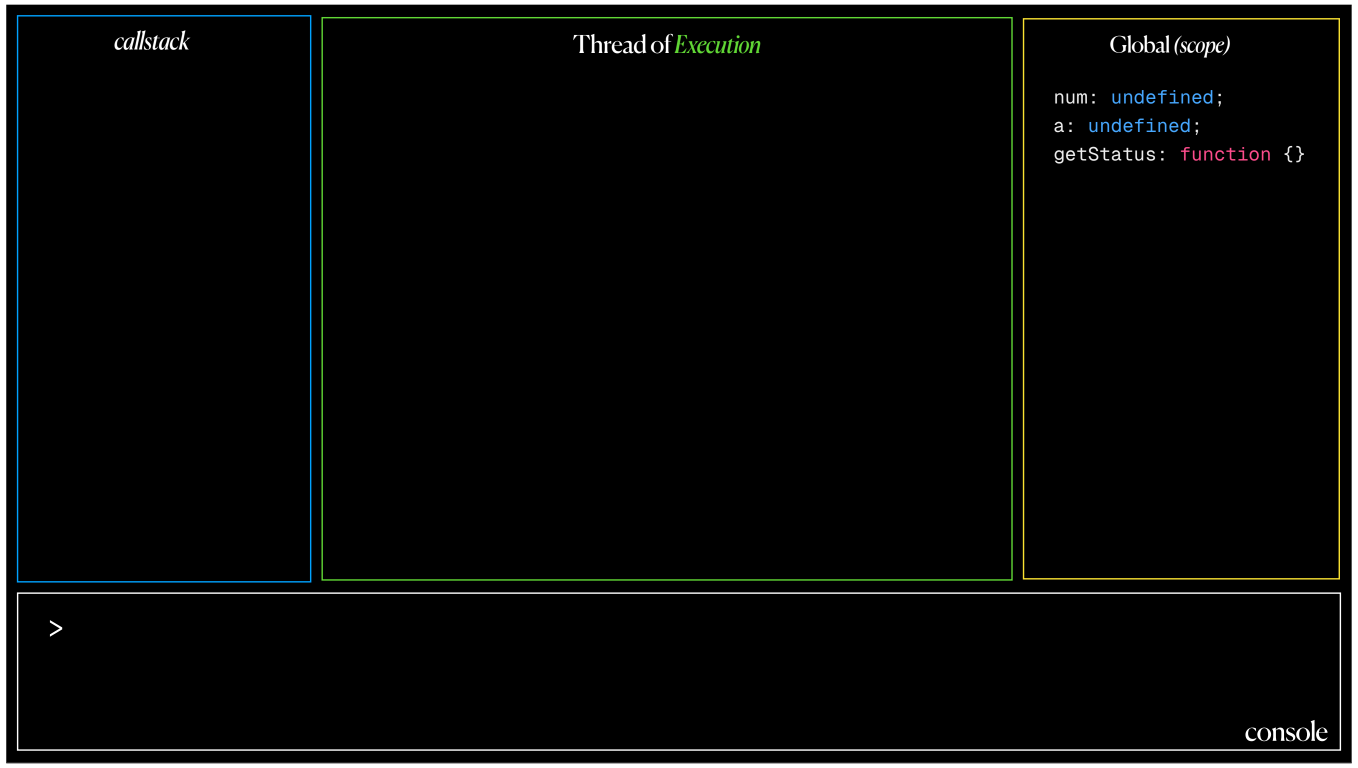


what our current Mental model looks like;
So we have callStack; Thread of execution; Global(scope)
>> Lets take an example (similar to previous one we just looked at) and see how everything works under the hood;


```
var num = 3;
var a = 'some string';

function getStatus() {
  console.log(status)
  var status = 'up' + a;
  return status;
}

console.log(getStatus());
console.log(status);
console.log(num);
```

Now Before we move on, I want you to keep in mind or (take a photograph of) this piece of code; so that you don't get lost along the way.
[WAIT FOR. A BIG]

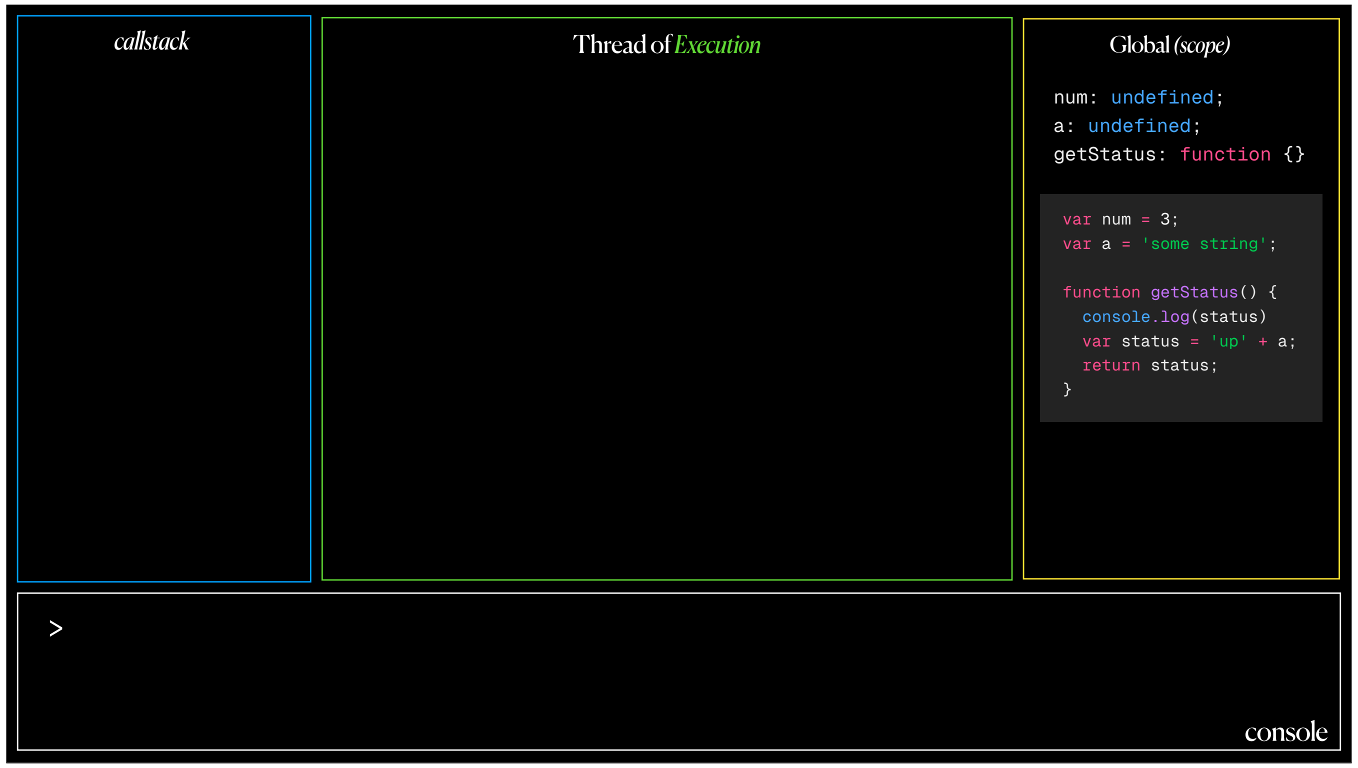
>> Now



```
num: undefined;
a: undefined;
getStatus: function {}
```

```
>
```

It is the default behaviour of javascript to move all declarations to the top of current scope:
Which you all may know as hoisting.
We will shortly go into this why these (num & a) are undefined at first.
>> So after this piece of code gets executed,

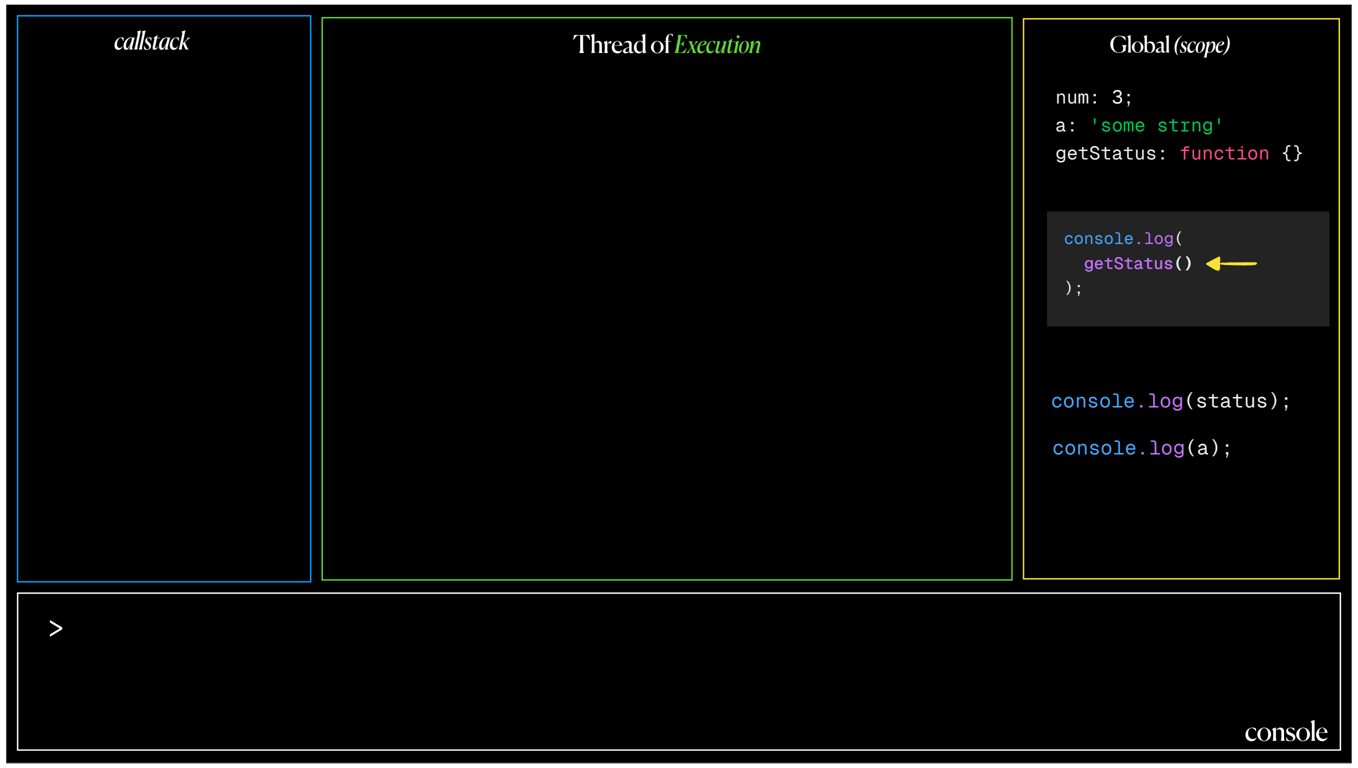


```
num: undefined;
a: undefined;
getStatus: function {}
```

```
var num = 3;
var a = 'some string';

function getStatus() {
  console.log(status)
  var status = 'up' + a;
  return status;
}
```

```
>
```

It is only then the variables gets updated to their rightful values;
>> Now in Javascript; a function

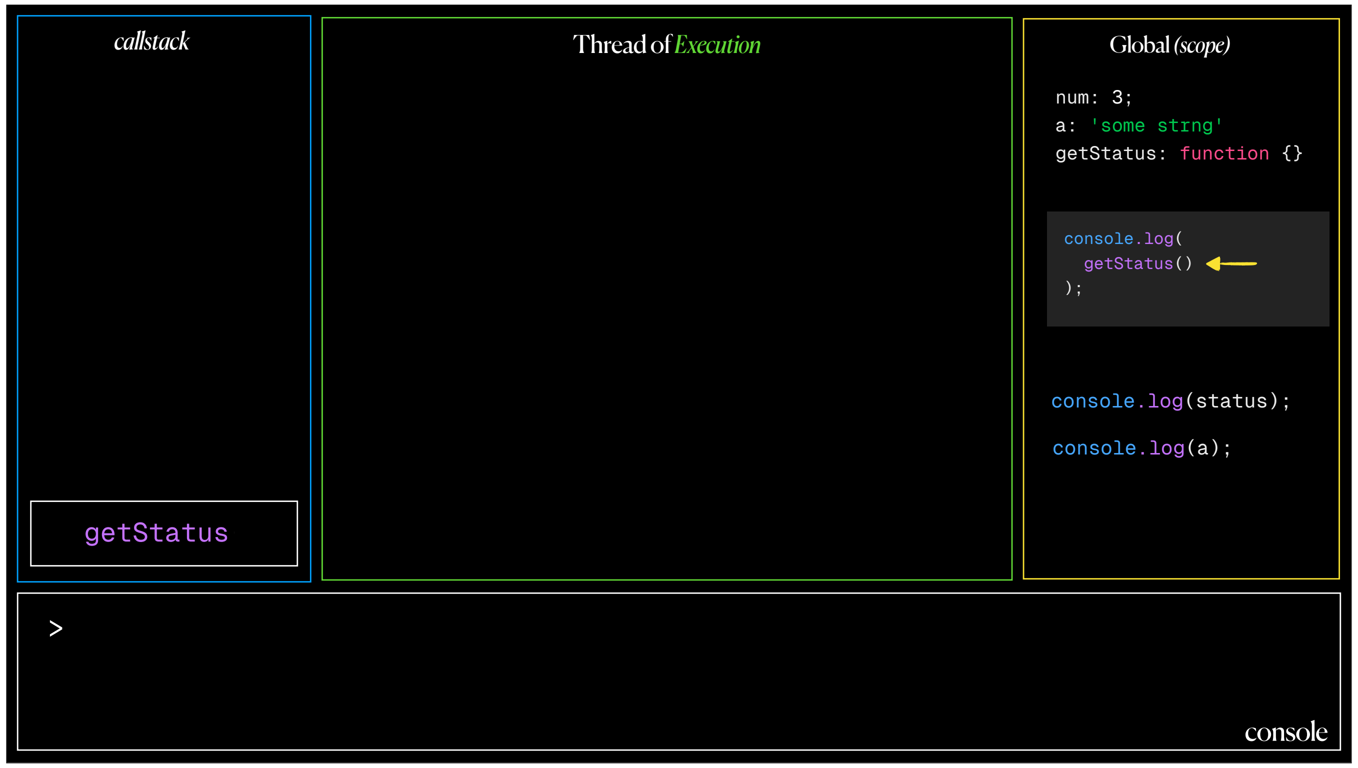


```
num: 3;
a: 'some strng'
getStatus: function {}
```

```
console.log(
  getStatus()
);
```

```
console.log(status);

console.log(a);
```

is actually called when it has these pair of parentheses in front of it.
>> Upon calling this function,

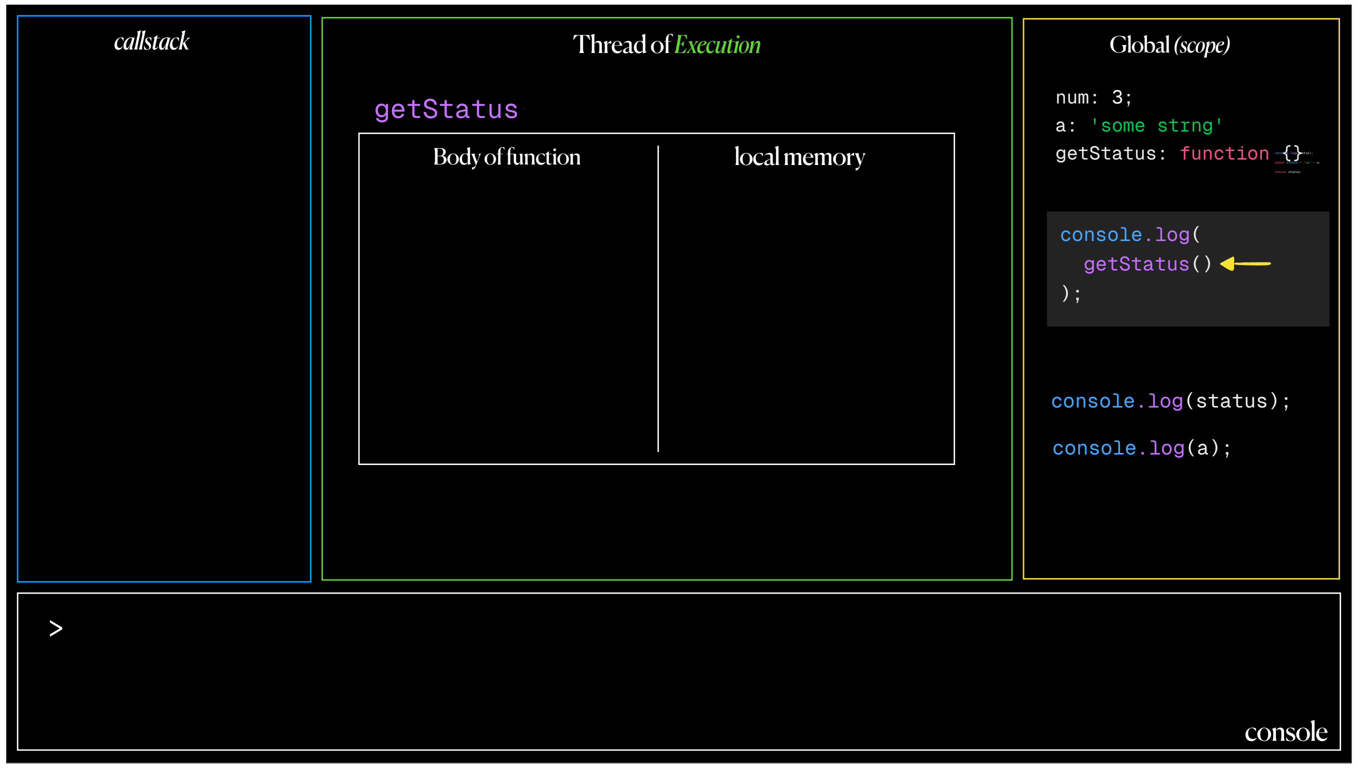


```
num: 3;
a: 'some strng'
getStatus: function {}
```

```
console.log(
  getStatus()
);
```

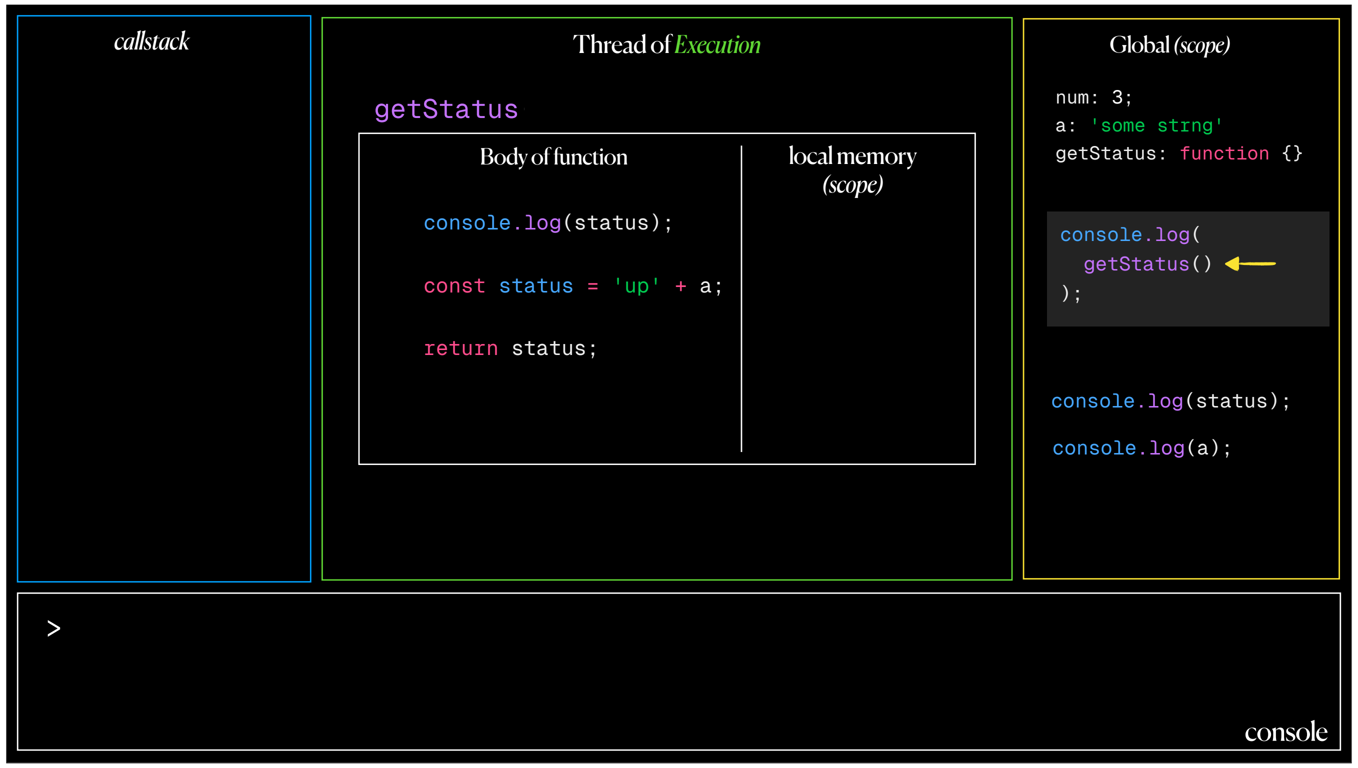
```
console.log(status);

console.log(a);
```

It gets put onto the callStack and remb. what I said about the callstack, whatever is at the top of it, gets executed; provided the thread of execution is empty;
and this is what single threaded behaviour means. It means the execution of JS itself is single threaded. Though there are ways to do concurrent and parallel processing; but that go beyond the scope of this talk.
>> So this getStatus function will now go to the thread of Execution



```
num: 3;
a: 'some strng'
getStatus: function {}

console.log(
  getStatus()
);

console.log(status);

console.log(a);
```

like I said before: the “*thread of execution”* consists of two main things: Local memory and the set of instructions to run.. (body of function here)
>> Now it will grab the body of function

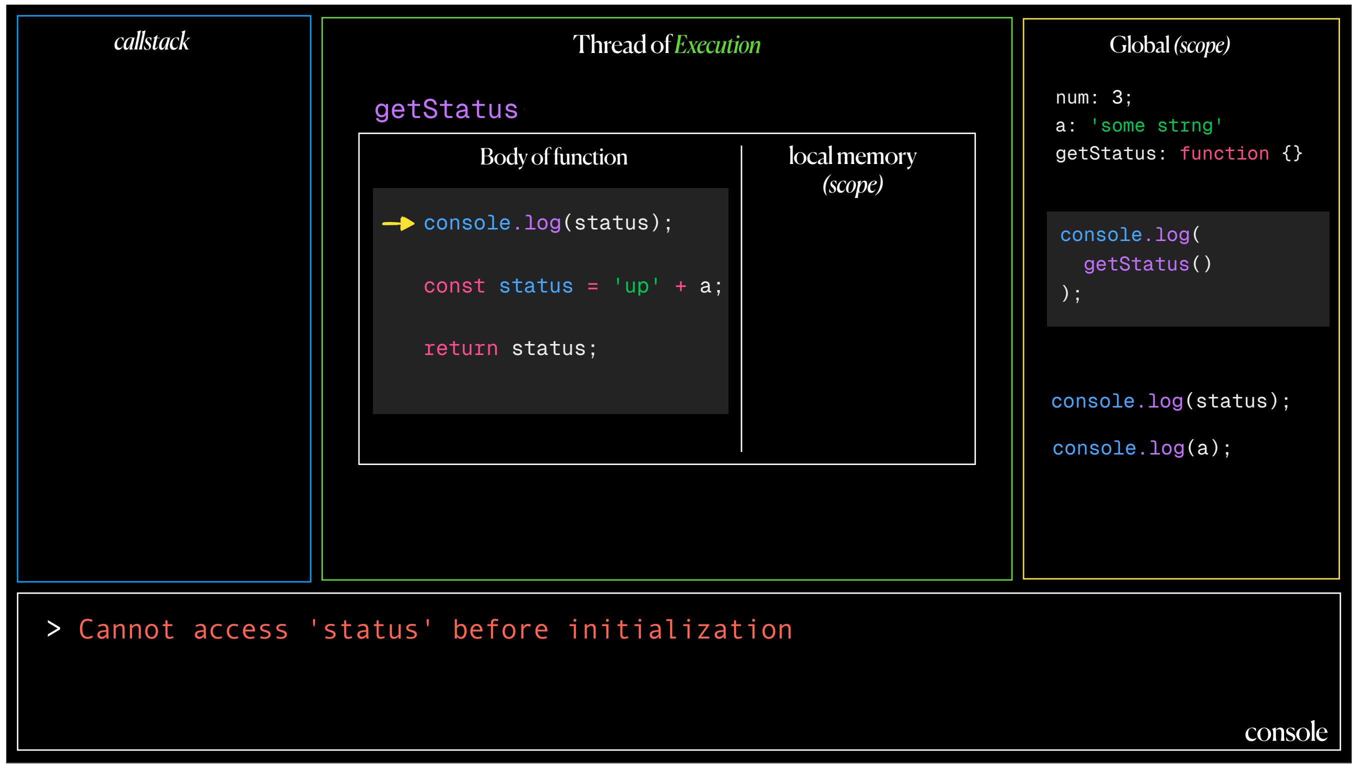


```
num: 3;
a: 'some strng'
getStatus: function {}

console.log(
  getStatus() ←
);

console.log(status);

console.log(a);
```

```
console.log(status);

const status = 'up' + a;

return status;
```

```
>
```

from the globalScope; and as far the hoisting is considered, unlike before, the value of status is not hoisted to undefined; this behaviour will be same if "let" was in place of "const" [I want you to observe that "var" and "const" have some different behaviour here.]
Now Upon the execution of this function; What will be the result of 1st line? …
>> Lets run this.

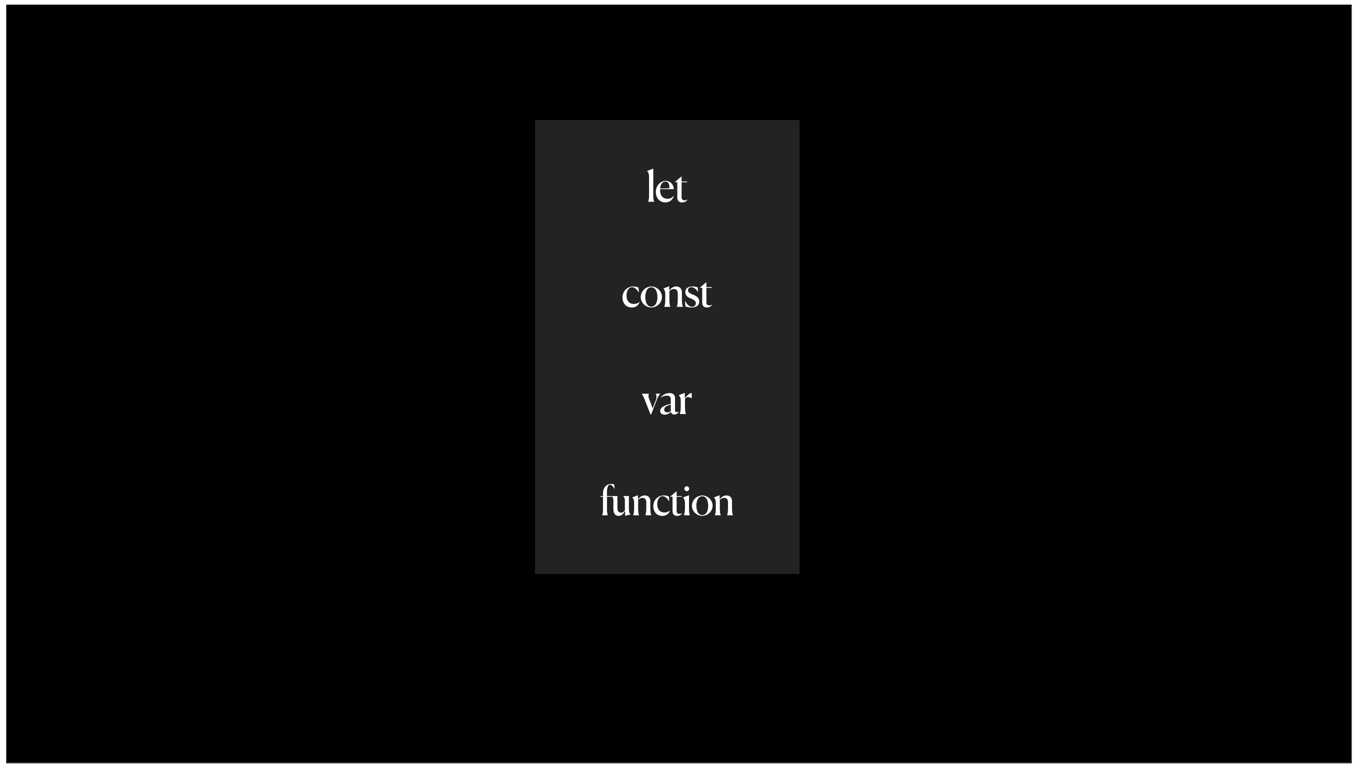


```
console.log(status);

const status = 'up' + a;

return status;
```

```
num: 3;
a: 'some strng'
getStatus: function {}

console.log(
  getStatus()
);


console.log(status);

console.log(a);
```

```
> Cannot access 'status' before initialization
```

It is showing ‘Cannot access ‘status’ before initialization’; but what about hoisting; we were promised that status is hoisted all to the top, so why the “access error” then?;
The first thing that should come from intuition is that “hoisting is not broken”: because at this point, out program knows status is something that is why it knows it is uninitialised (even it is not executed yet.);
Now this happens because hoisting works differently for diff kind of declarations;
>> Lets deviate a bit and talk about hoisting for a moment;



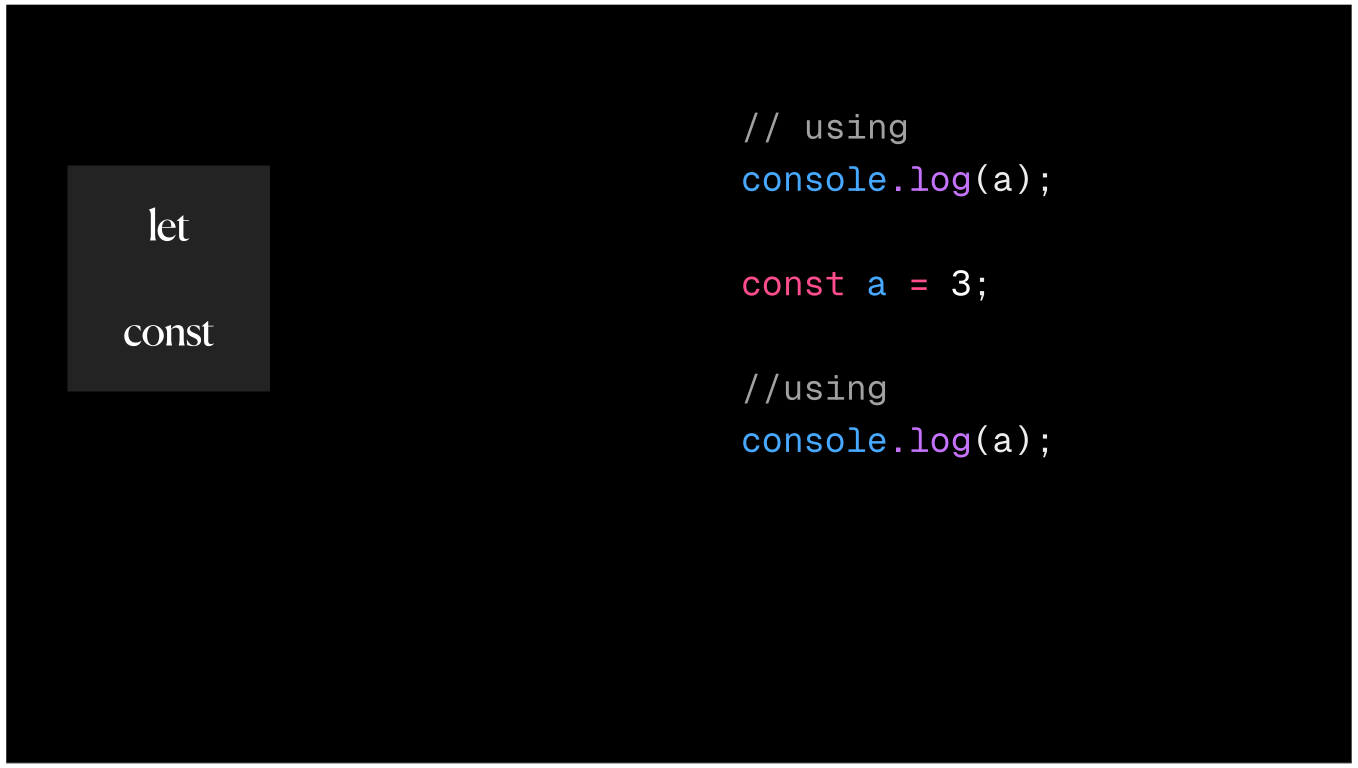
let

const

var

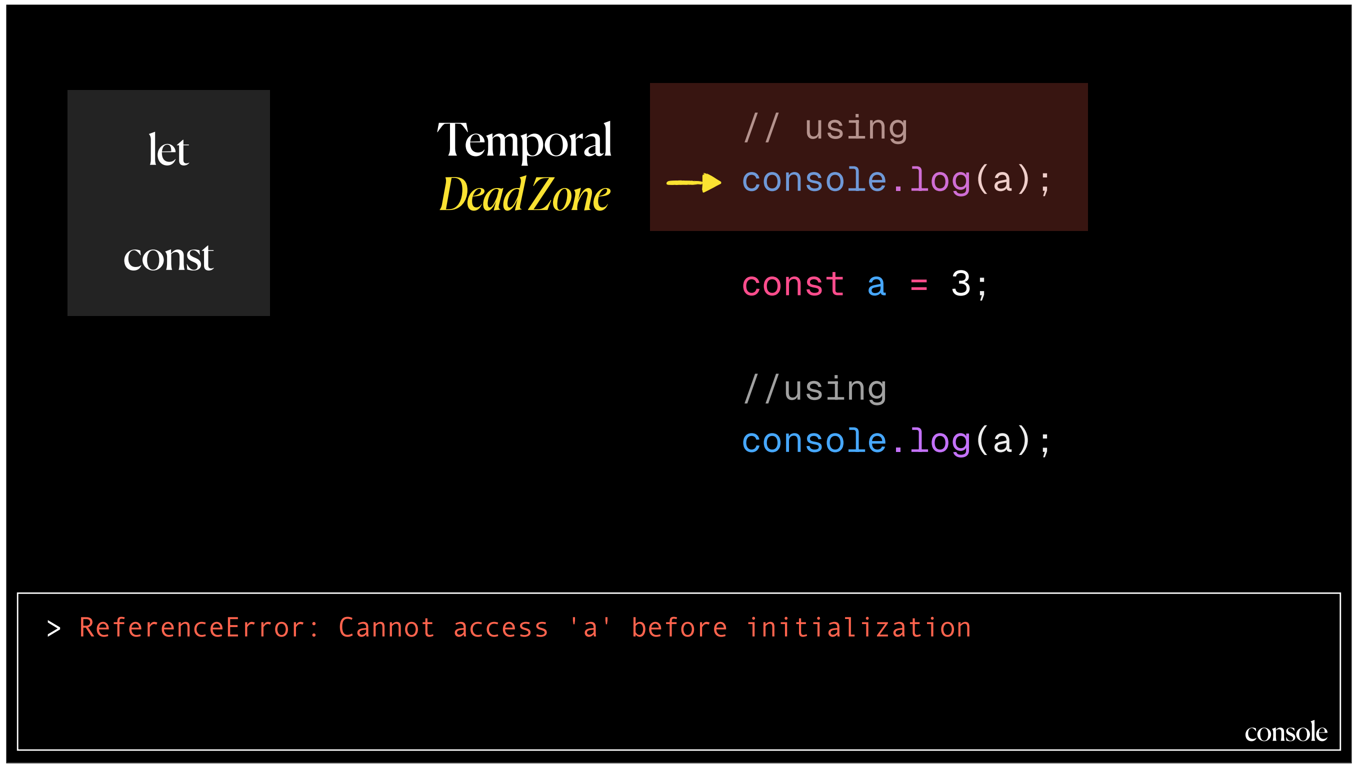
function

Hoisting shows different behaviour for different declarations
Some of which include:
- const declarations; let declarations; var declarations; function declarations;
>> Lets see get into this



```
// using
console.log(a);

const a = 3;

//using
console.log(a);
```

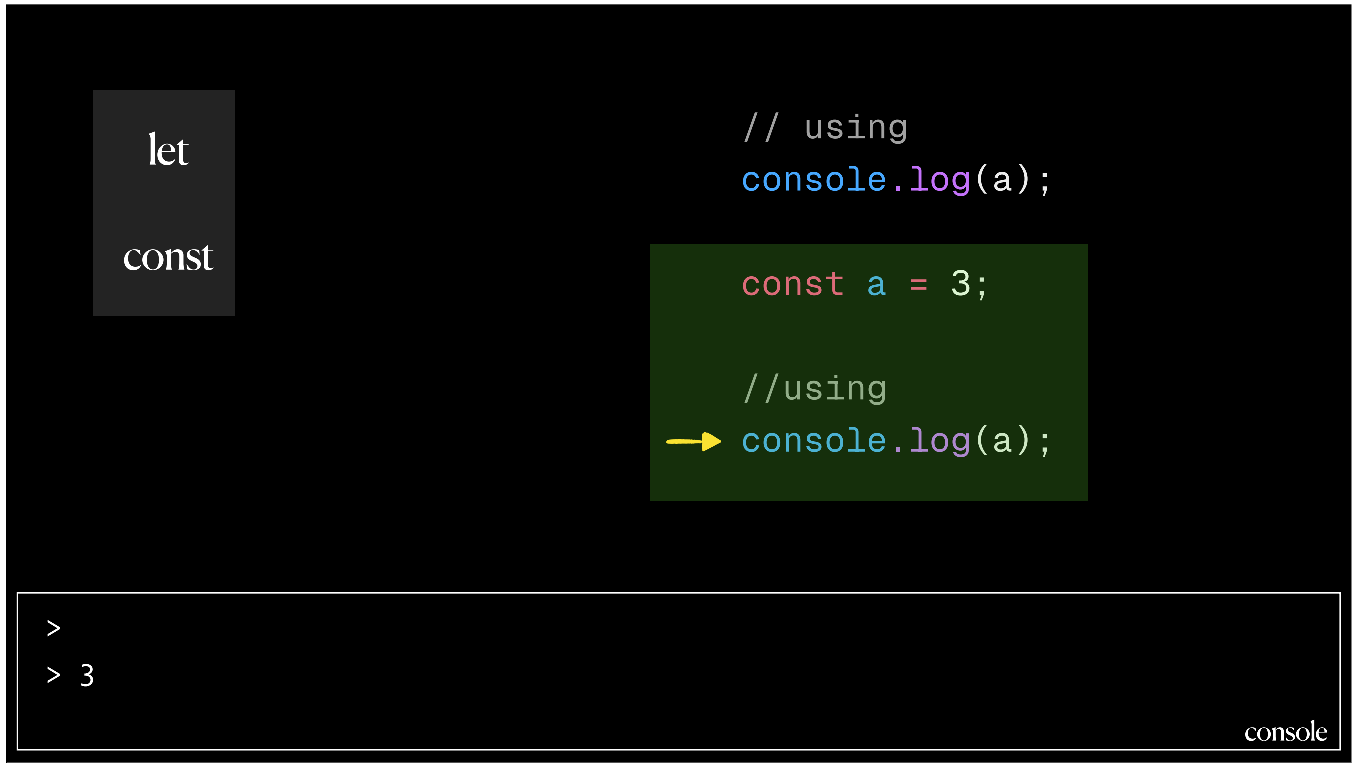
Starting with let and const, they fall into the same category called “Value hoisting” / “lexical hoisting”
>> So in lexical hoisting; the portion of code (OR scope)



```
// using
console.log(a);

const a = 3;

//using
console.log(a);
```

```
> ReferenceError: Cannot access 'a' before initialization
```

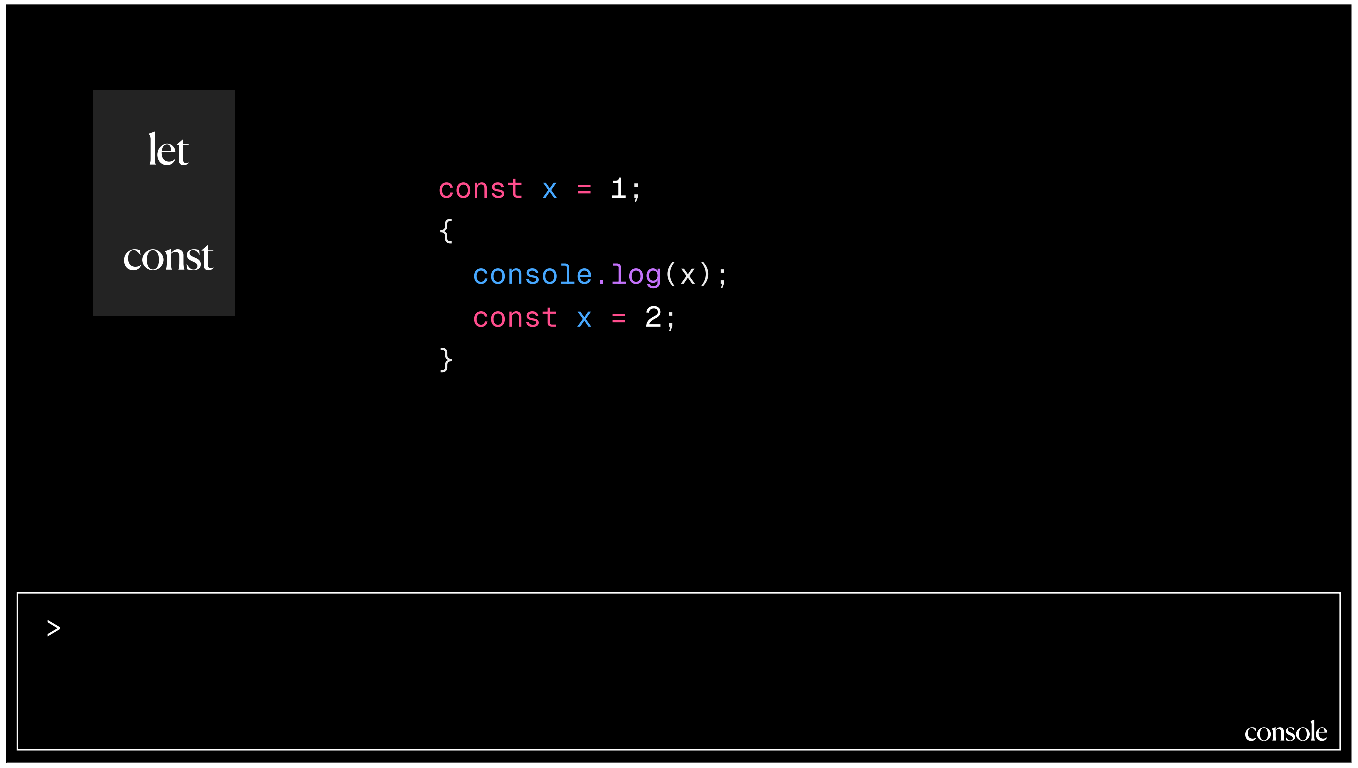
declared before the “execution of declaration”, falls in **TemporalDeadZone;** with respect to the declaration (which here is ‘a’). So in this TDZ, value of this declaration, cannot be accessed; (hence the ReferenceError)
>> Now moving along the execution,



```
// using
console.log(a);

const a = 3;

//using
console.log(a);
```

```
>
> 3
```

you can see that our execution has passed the declaration and when we reach second console.log statement here; we see the result 3 in console
>> Now to test your skills; tell me



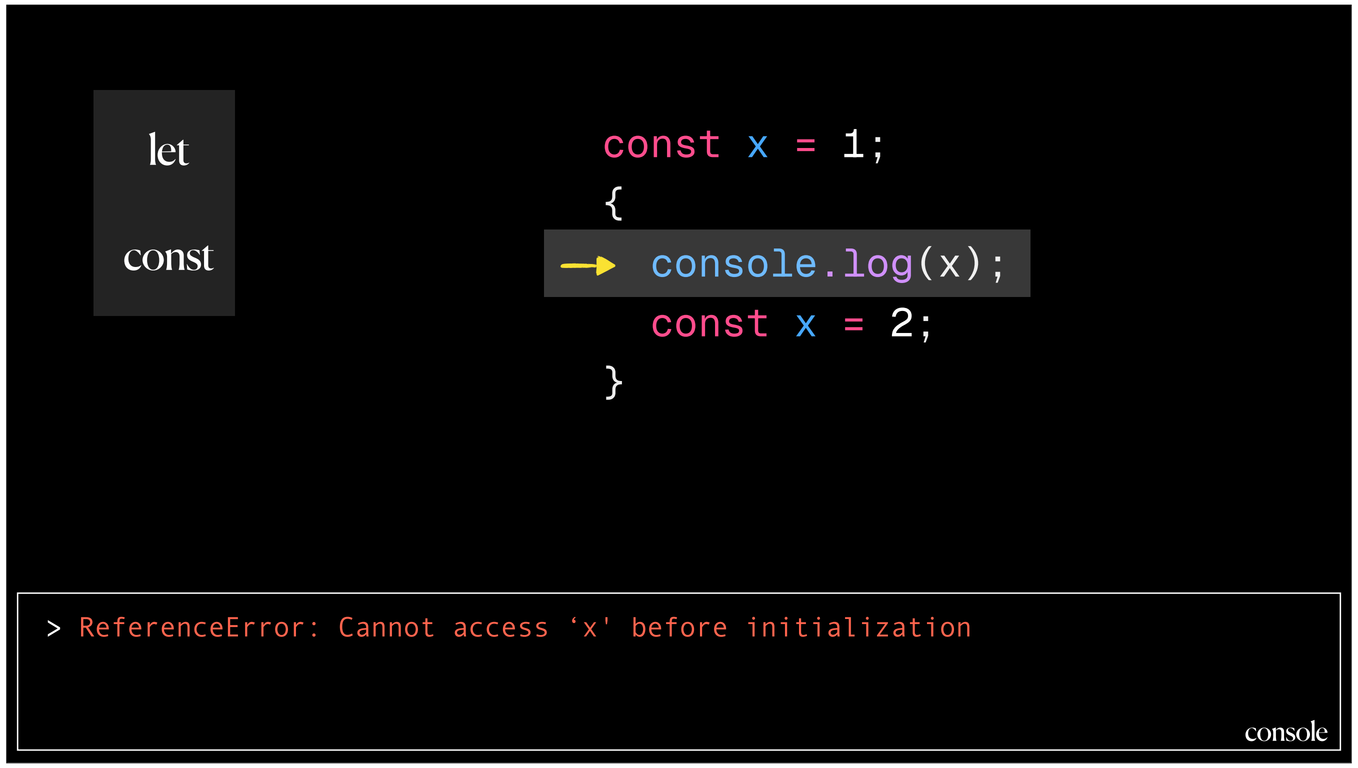
let

const

```
const x = 1;
{
  console.log(x);
  const x = 2;
}
```

```
>
```

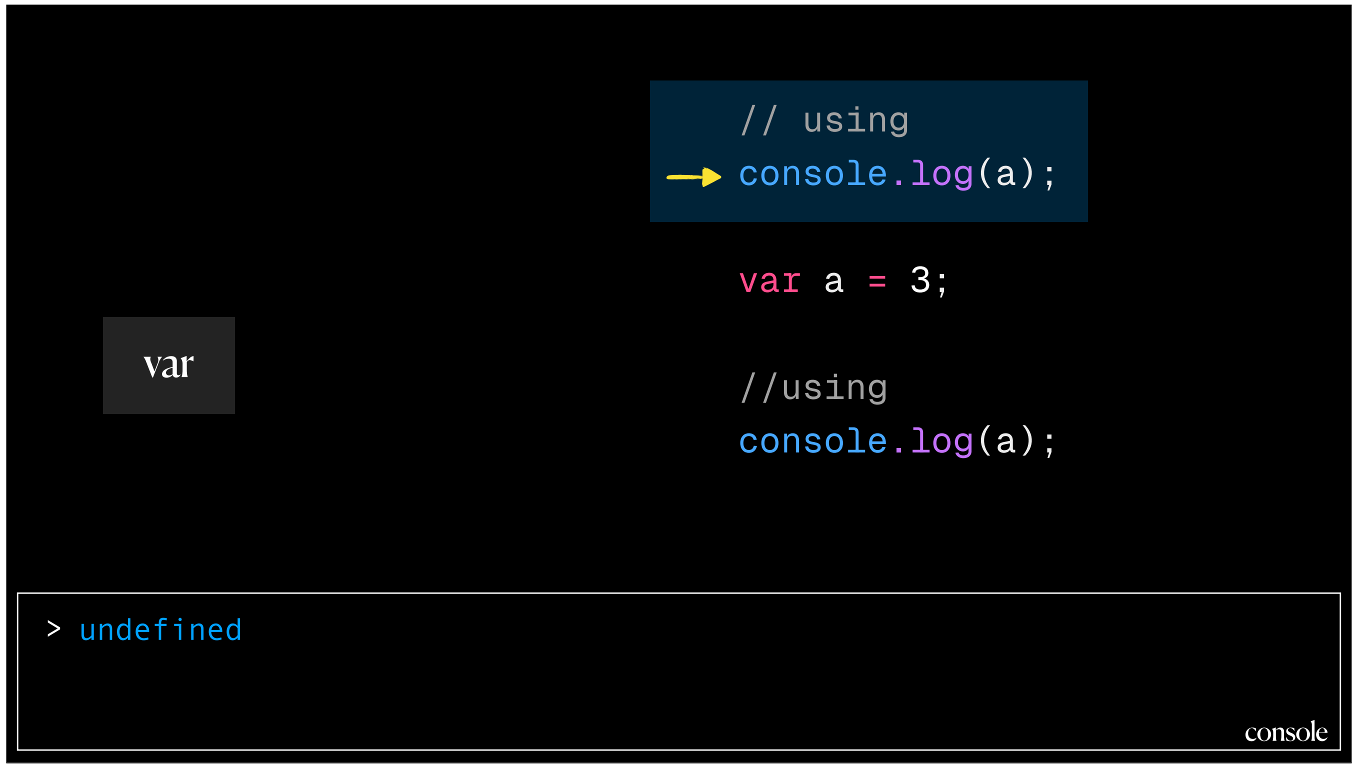
what will get logged down.
1… 2… 3… 4… {show qr code to submit result}
>> So lets see



```
const x = 1;
{
  console.log(x);
  const x = 2;
}
```

```
> ReferenceError: Cannot access 'x' before initialization
```

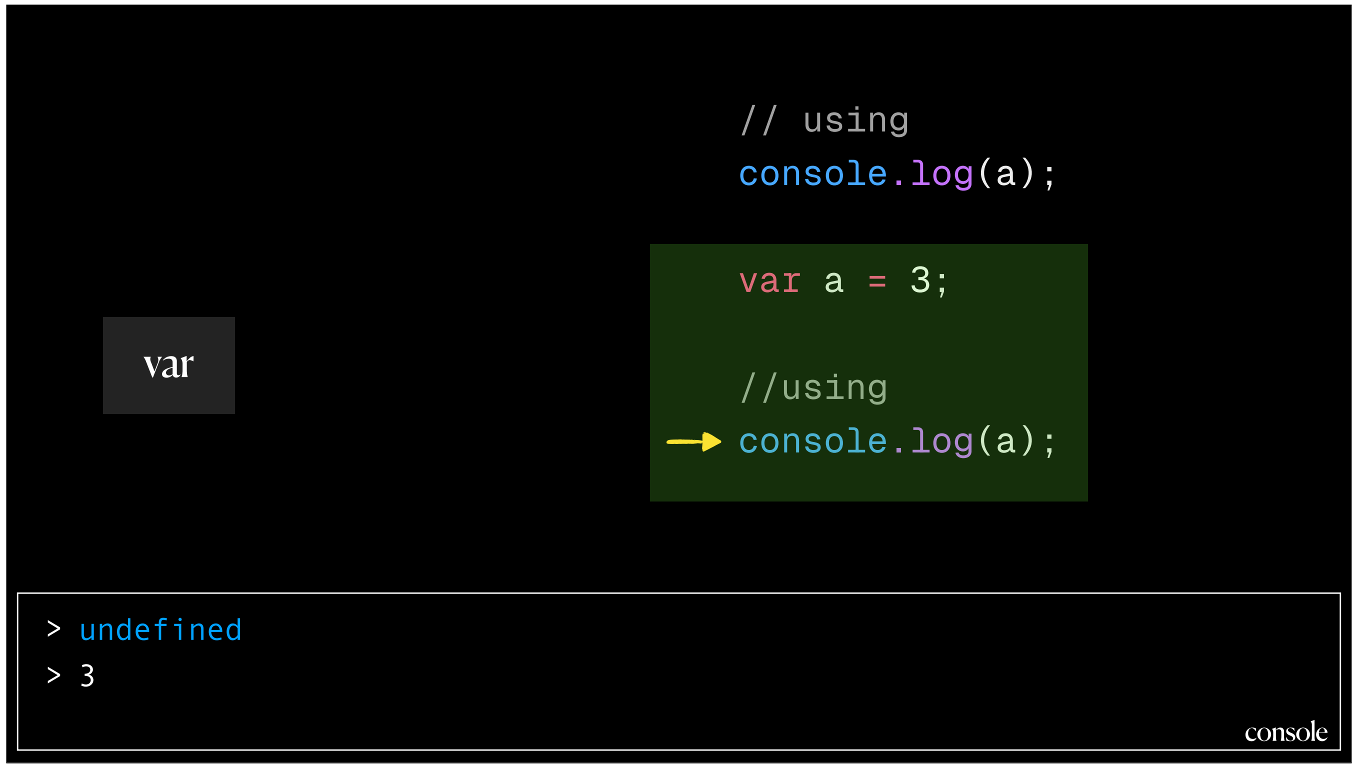
It turns out we neither have 1 nor 2 but a reference error.
This is because in the block, the console.log is right now, “x” cannot be accessed before the “execution of its declaration” remb. It is “lexical hoisting”
>> Now when it comes to var, the portion



```
// using
console.log(a);

var a = 3;

//using
console.log(a);
```

```
> undefined
```

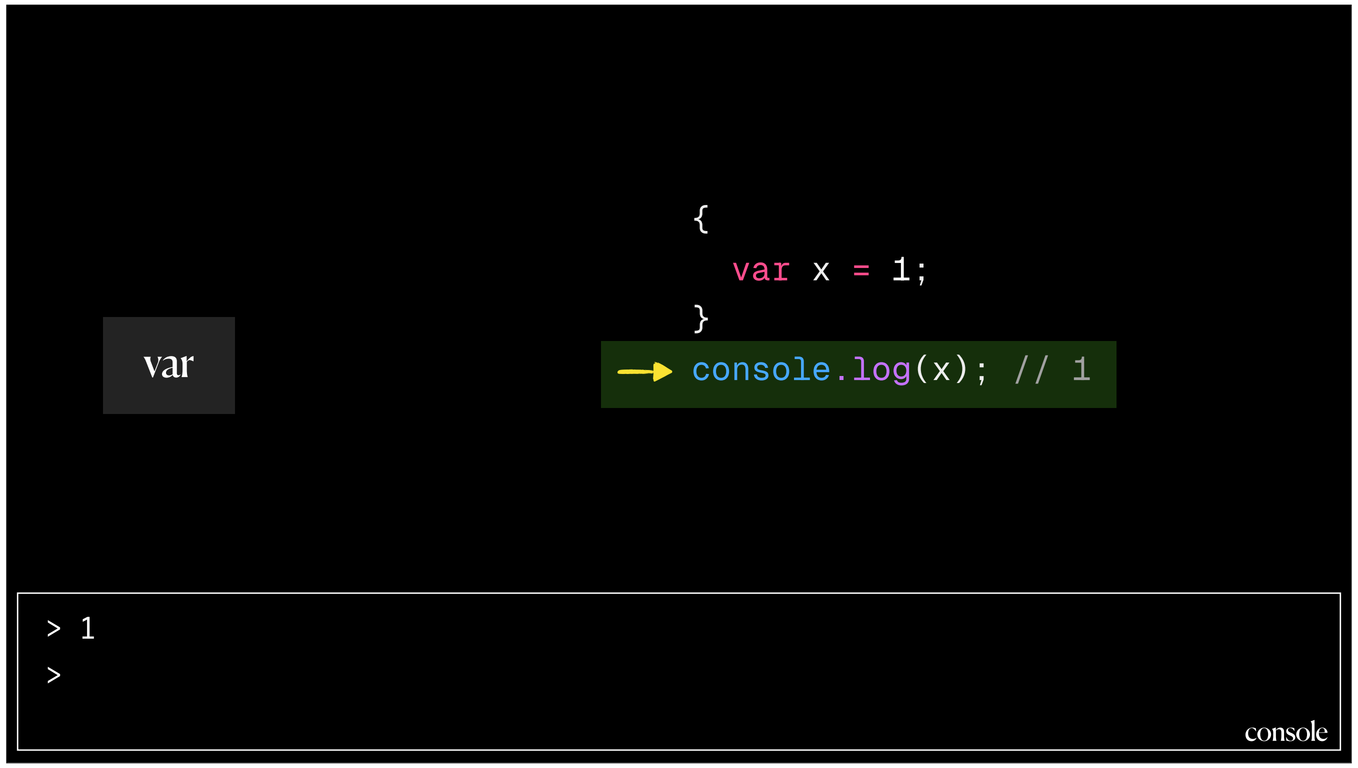
that was TemporalDeadZone for let and const; is the portion where the value of our “var declaration” remains undefined. This is called “declaration hoisting”, Which justifies the log we are seeing in console and why in our code example before, the variables were initially undefined at top.
>> Now moving on with the execution,



```
// using
console.log(a);

var a = 3;

//using
console.log(a);
```

```
> undefined
> 3
```

We get 3 in the console as expected;
Another thing to note here is that var declarations are not scoped to blocks; In other words bock doesn’t prevent the scope of “var” to go up in chain.
>> Which means something like this will totally work


var

```
{
  var x = 1;
}
console.log(x); // 1
```

```
> 1
>
```

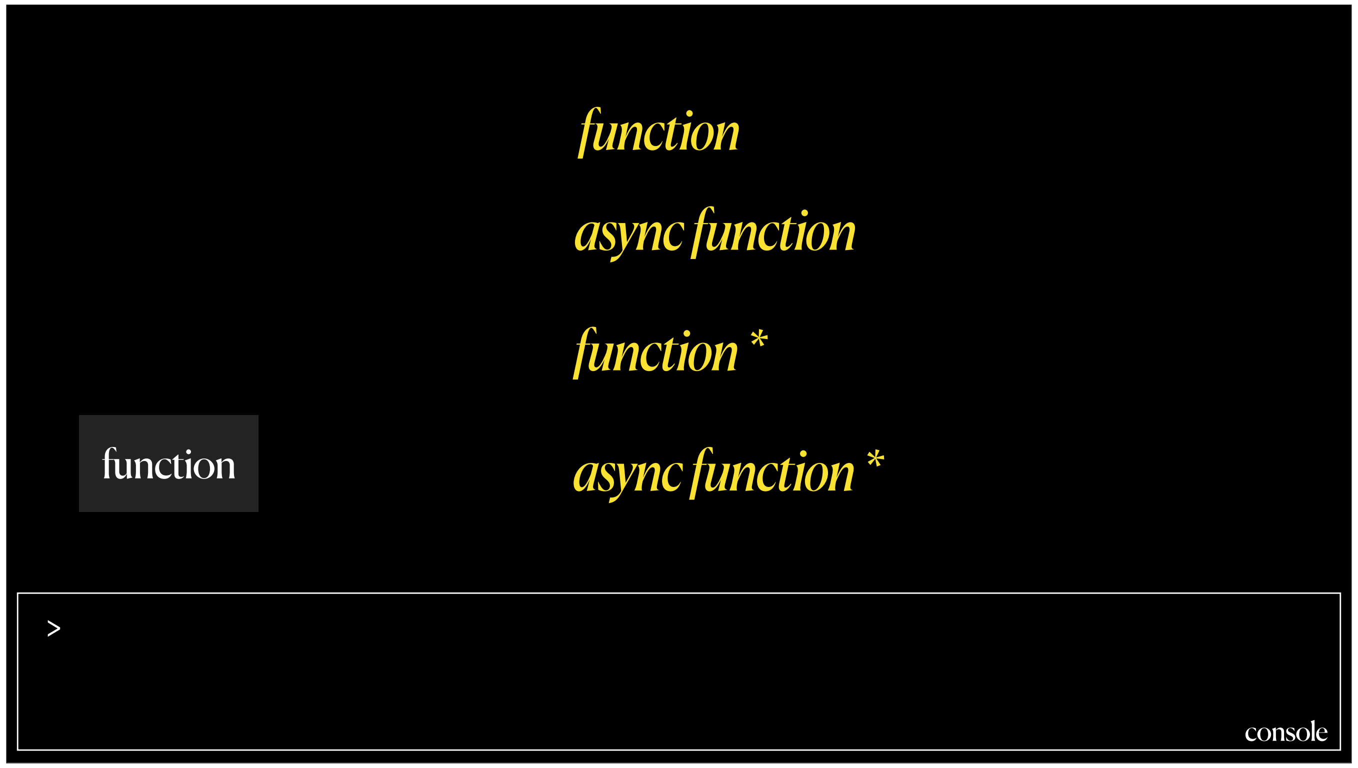
will totally work.
>> When it comes to functions

function

```
>
```

console

Functions are a bit tricky because they can be created in different ways;
Function Expressions (that are often anonymous): can be used as callbacks [click], IIFEs [click], Assigned to variables using var/let/const, anonymous functions[click], named & anonymous function expressions and function declarations;
>> Now specifically in function declarations made using


function

function

async function

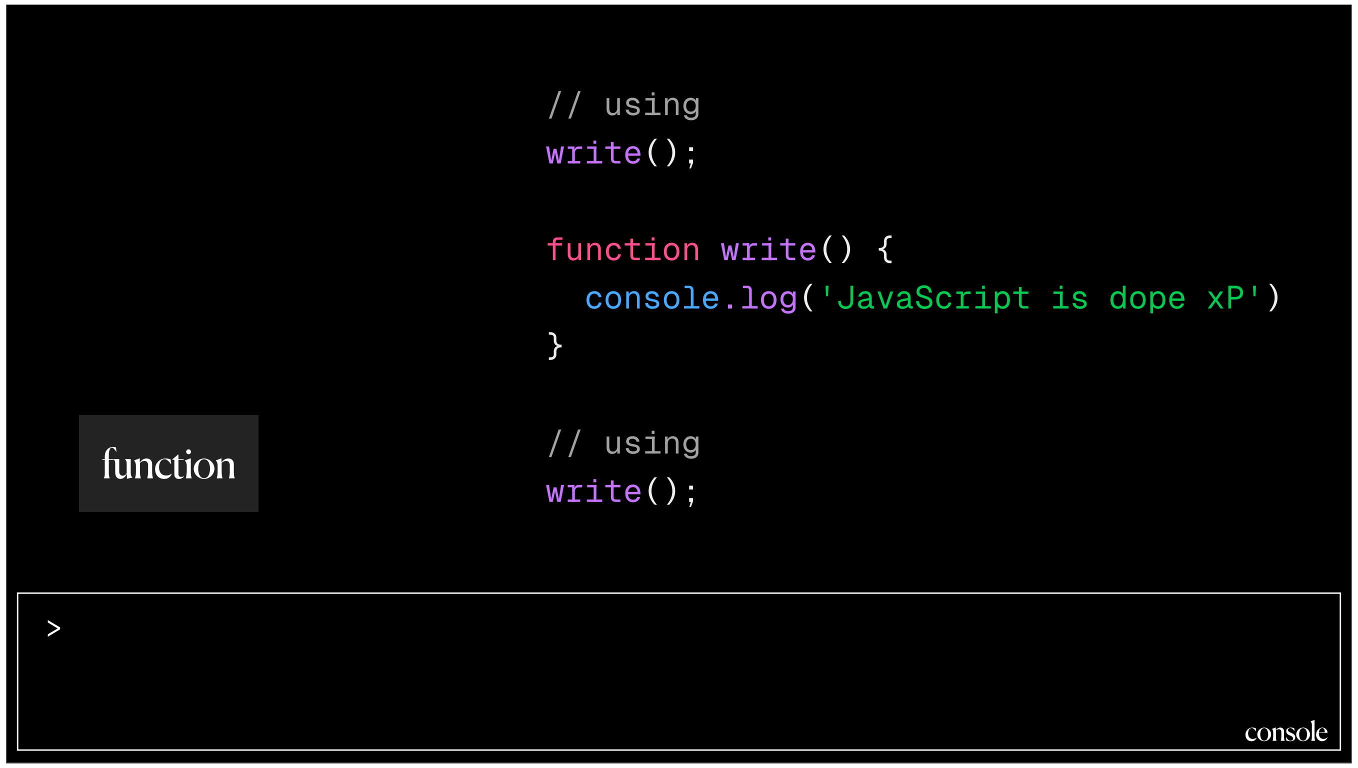
*function **

*async function **

```
>
```

console

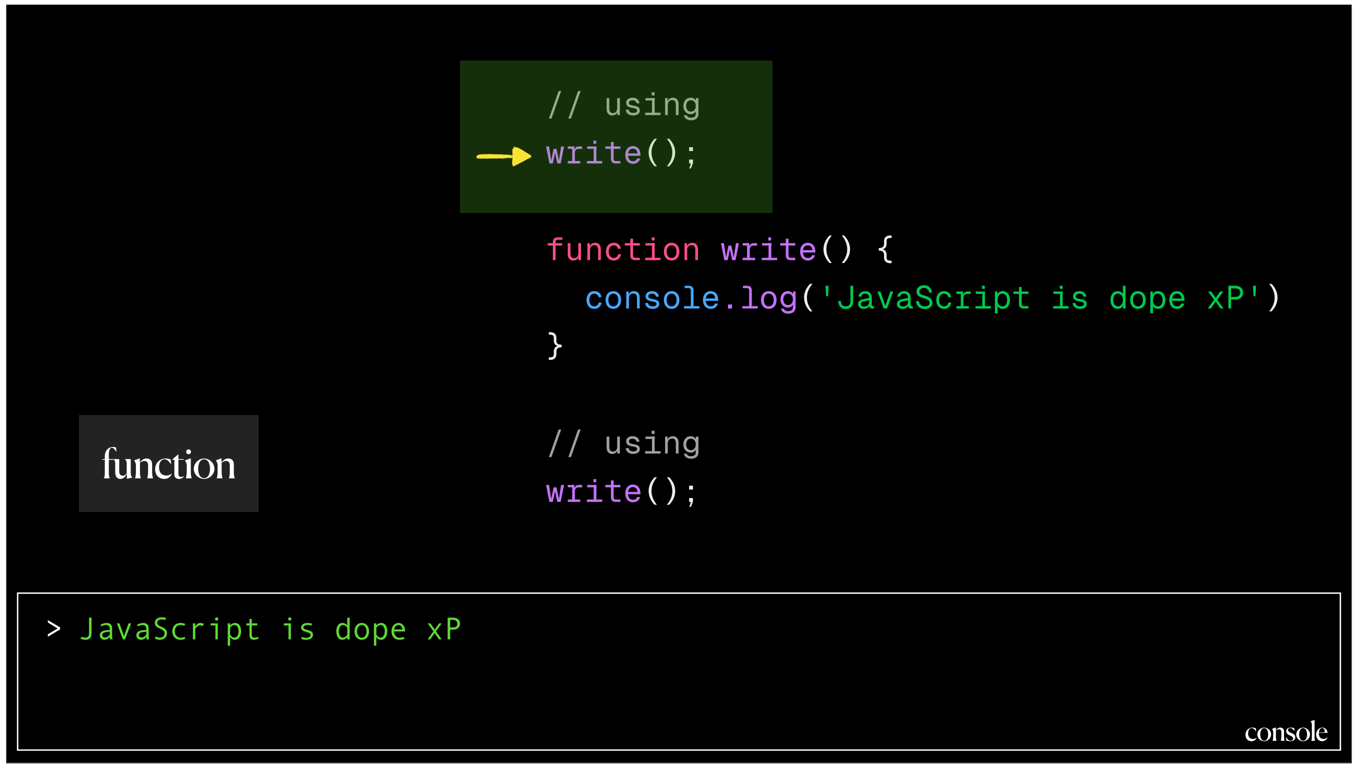
The following 4 notations:
1. Using function keyword;   2. Using async with function keyword.
3. Using generator function signature. 4. Using async with generator function signature.
>> I will demonstrate an example for the 1st function keyword declaration, (same applies to all of them)



```
// using
write();

function write() {
  console.log('JavaScript is dope xP')
}

// using
write();
```

```
>
```

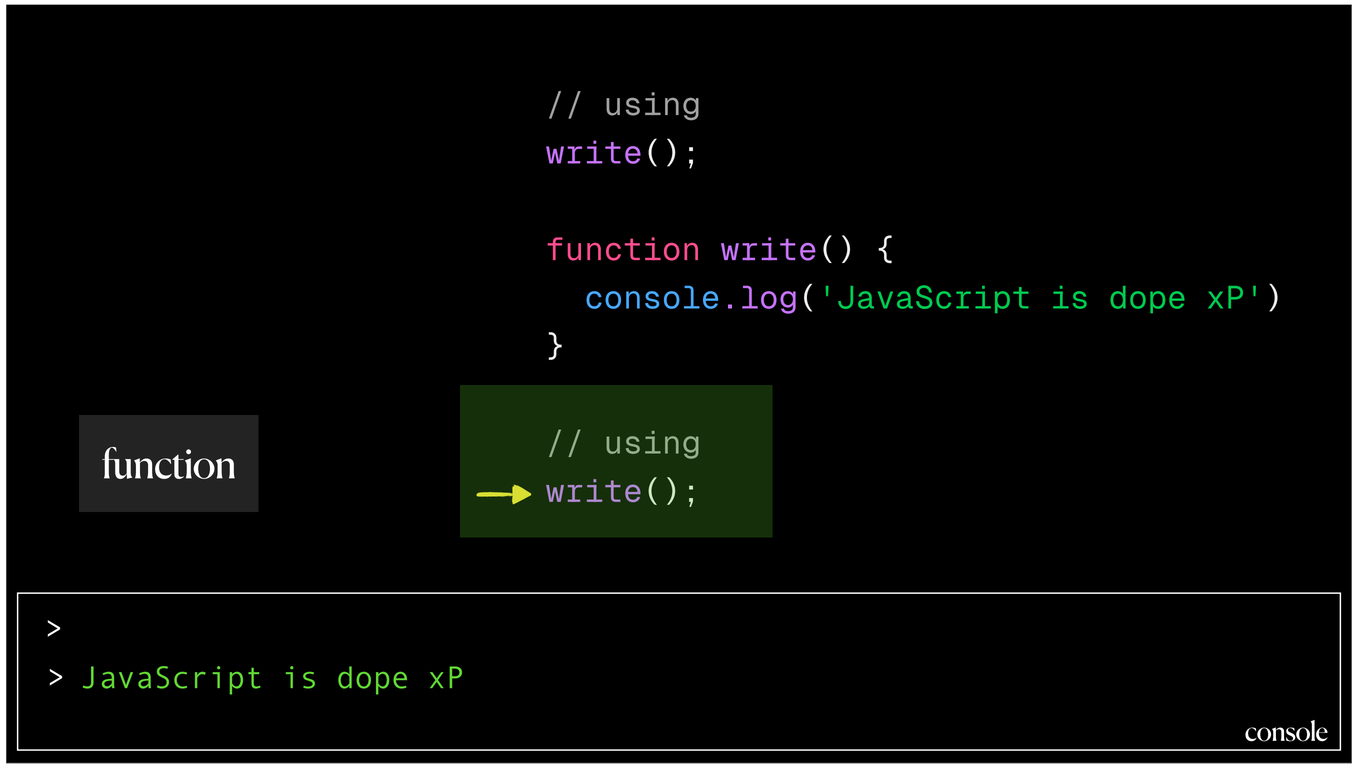
So this is how our function declaration looks like:
we have declared a function with the “function” keyword and called it two times
one Before the execution of its declaration and
One After the execution of its declaration.
>> So lets move on with the execution.



```
// using
write();

function write() {
  console.log('JavaScript is dope xP')
}

// using
write();
```

```
> JavaScript is dope xP
```

As you can see in the console we get the log as expected;
Which Implies even we called it before the “execution of its declaration”,
because of hoisting, we were able to use this function anywhere we want; even somwhere (what would have been a TDZs in case of const and let.)
>> Now moving on with the execution



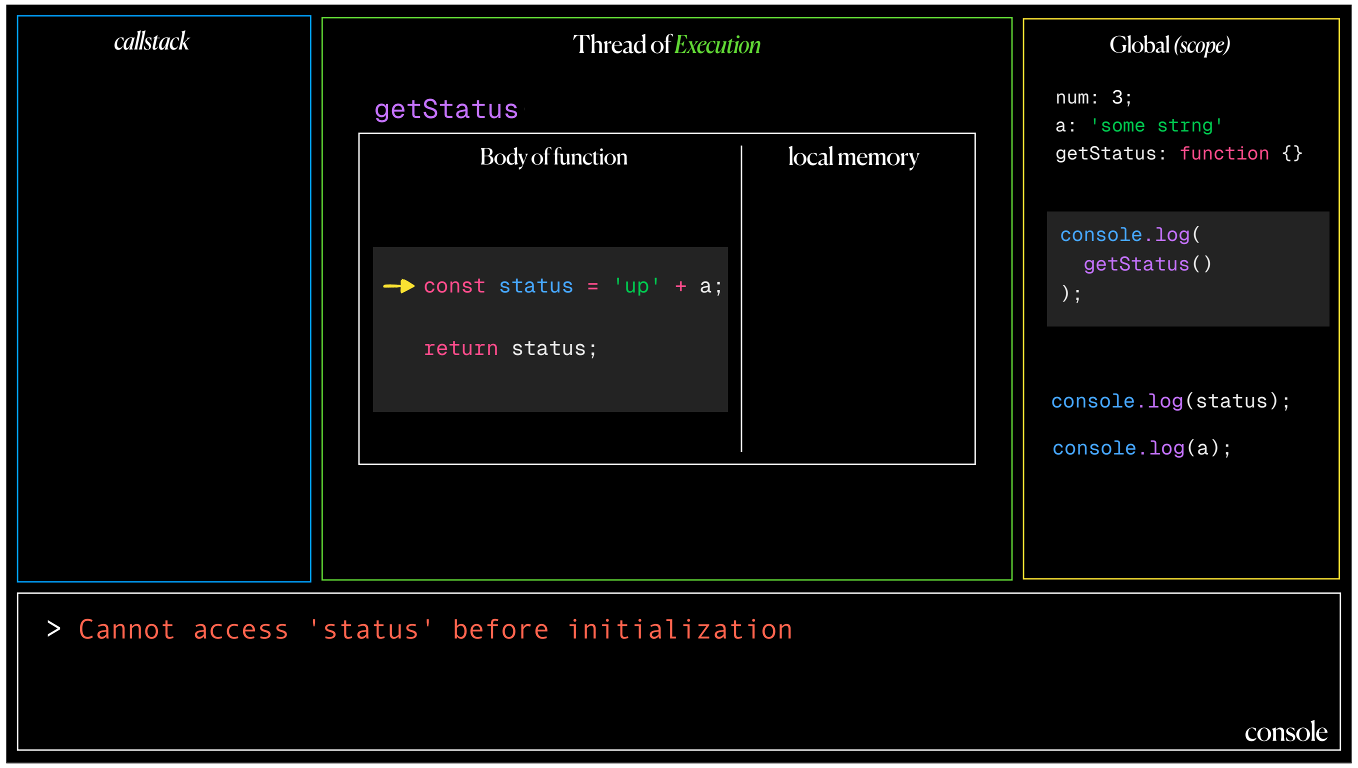
function

```
// using
write();

function write() {
  console.log('JavaScript is dope xP')
}

// using
write();
```

```
>
> JavaScript is dope xP
```

console

The second log is as expected;
Now I hope the concept of hoisting and functions is clear in your heads; So lets go back to
>> the Execution context of our getStatus function.



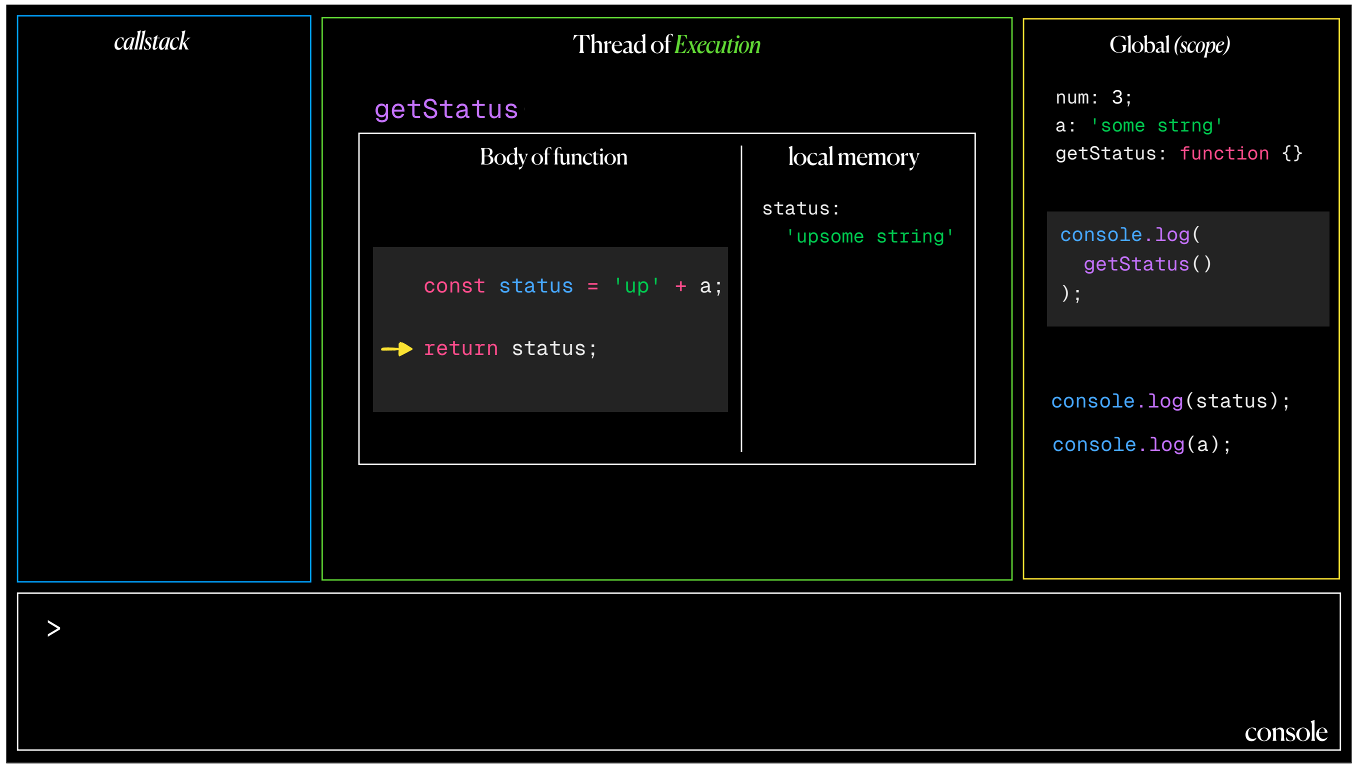
```
const status = 'up' + a;

return status;
```

```
num: 3;
a: 'some strng'
getStatus: function {}
```

```
console.log(
  getStatus()
);


console.log(status);

console.log(a);
```

```
> Cannot access 'status' before initialization
```

Now you know why we were getting this error of initialization (cz we were using status in its TDZ);
Lets just say this line was not in our code, the execution will then continue and will pass the “execution of declaration” of “status" variable;
>> Now you can see that we have



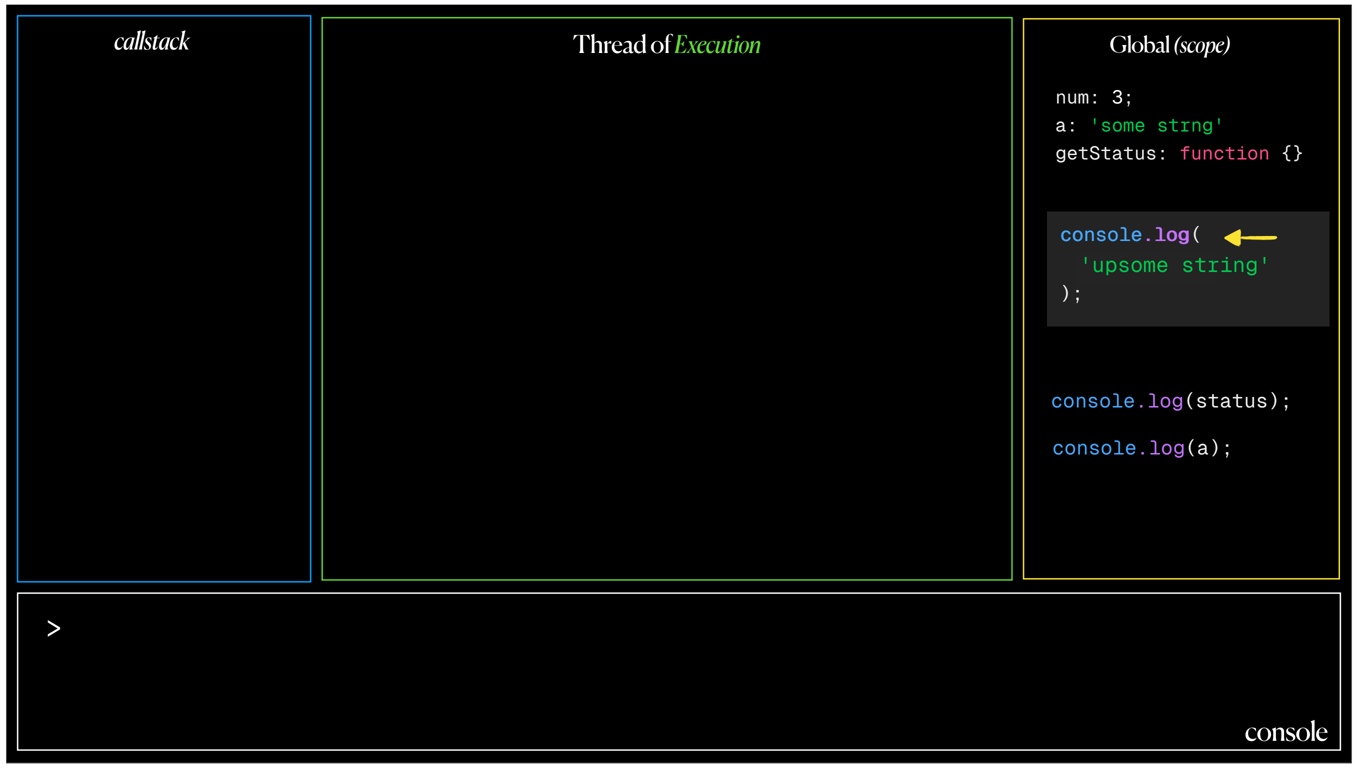
```
getStatus

const status = 'up' + a;

return status;
```

```
status:
  'upsome string'
```

```
num: 3;
a: 'some strng'
getStatus: function {}

console.log(
  getStatus()
);

console.log(status);

console.log(a);
```

```
>
```

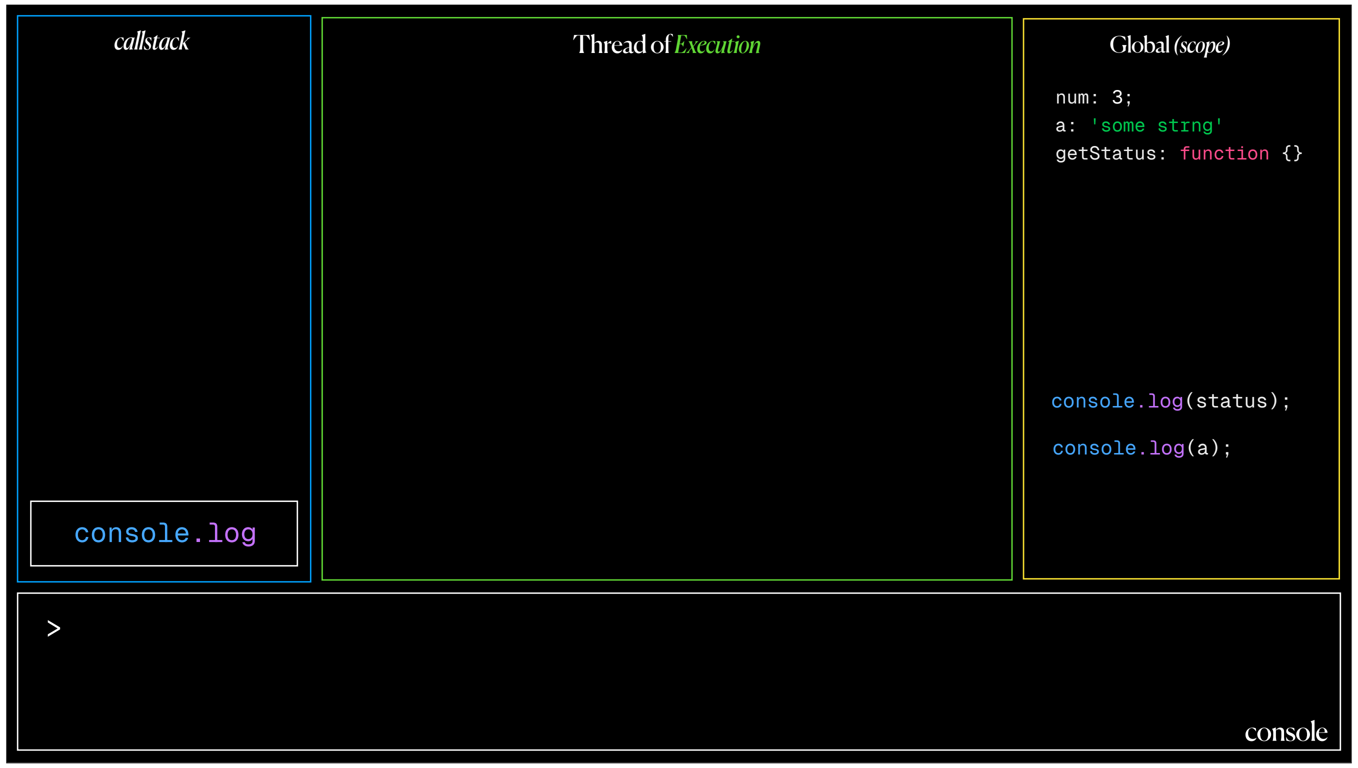
the status variable in our local memory; with the value of ‘upsome string’, the fact that it was able to access the value of ‘a’ from its outermost scope is what is known as “Scope chaining” in JS.
So after hitting that return statement, the getStatus function execution completes.
The value 'up' is returned back to the point where the function was called - in this case, to the console.log() statement.
>> At this point, the execution context of getStatus is destroyed. and the garbage collector of v8 engine will reclaim the unused memory in its GarbageCollection rounds.



```
num: 3;
a: 'some strng'
getStatus: function {}

console.log(
  'upsome string'
);

console.log(status);

console.log(a);
```

```
>
```

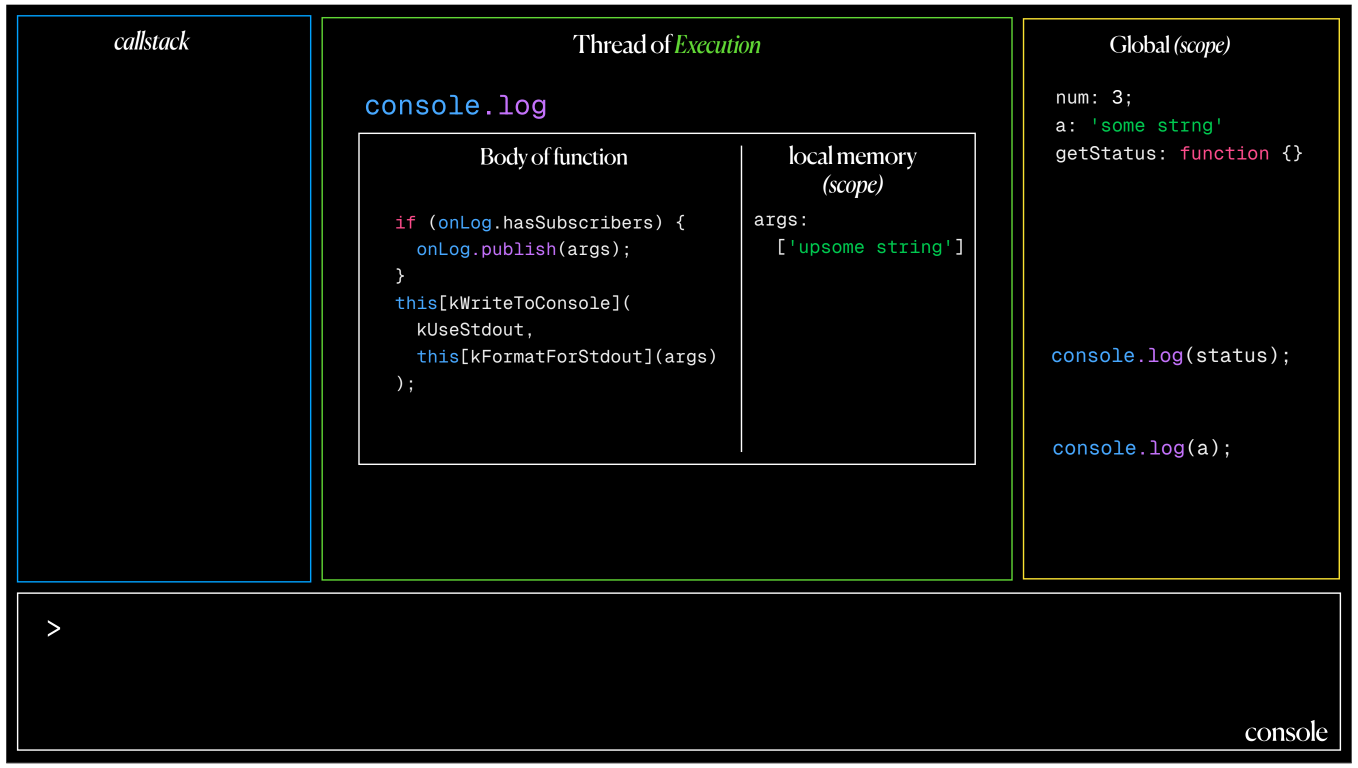
Now our execution is back to the console.log function; Remb. I said that every function that needs to be executed will first go to the callStack, console.log remains no exception
>> After all console.log in itself is



```
num: 3;
a: 'some strng'
getStatus: function {}

console.log(status);

console.log(a);
```

```
console.log
```

a function [which in nodeJS has the default behaviour to stream the data passed to it to the std>output (which here is our console)]
>> The reason I skipped this till now is;



```
console.log
```

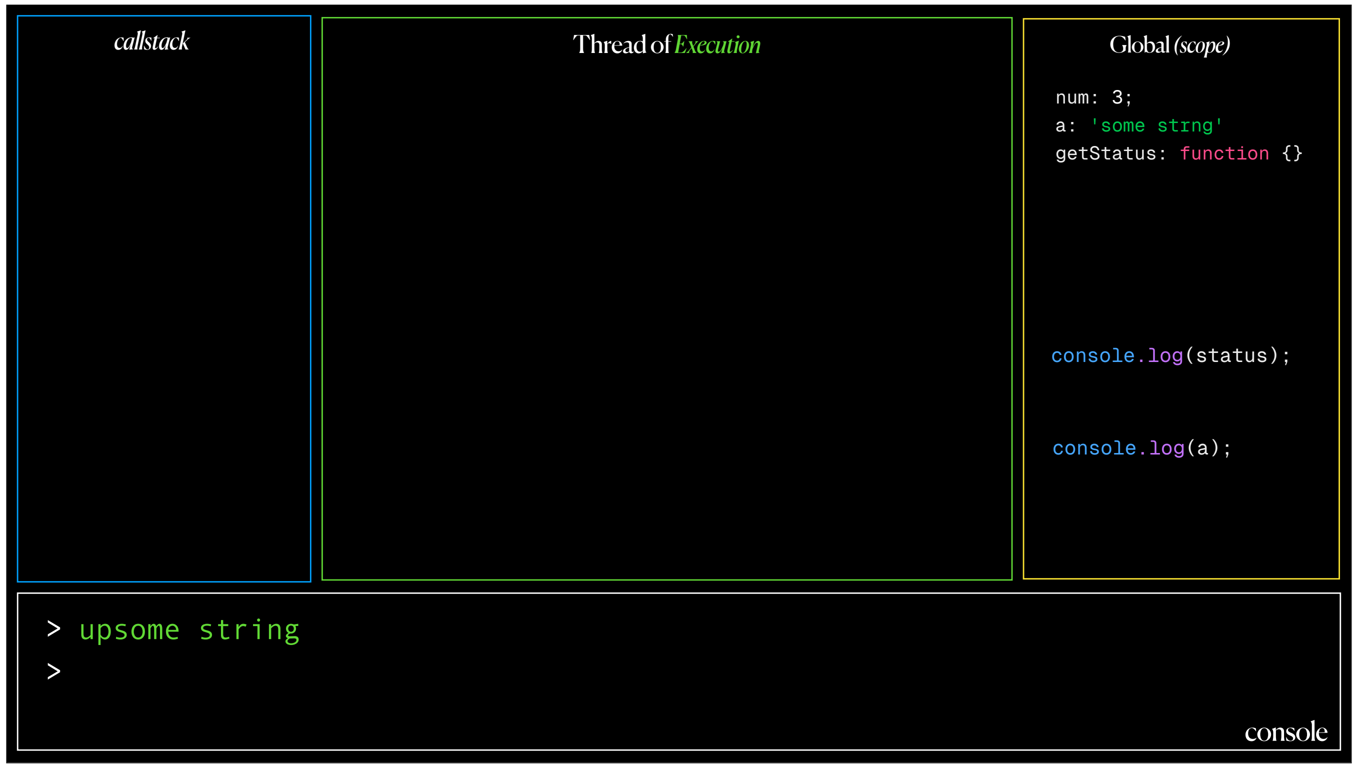
```
if (onLog.hasSubscribers) {
  onLog.publish(args);
}
this[kWriteToConsole](
  kUseStdout,
  this[kFormatForStdout](args)
);
```

```
args:
  ['upsome string']
```

```
num: 3;
a: 'some strng'
getStatus: function {}

console.log(status);

console.log(a);
```

```
>
```

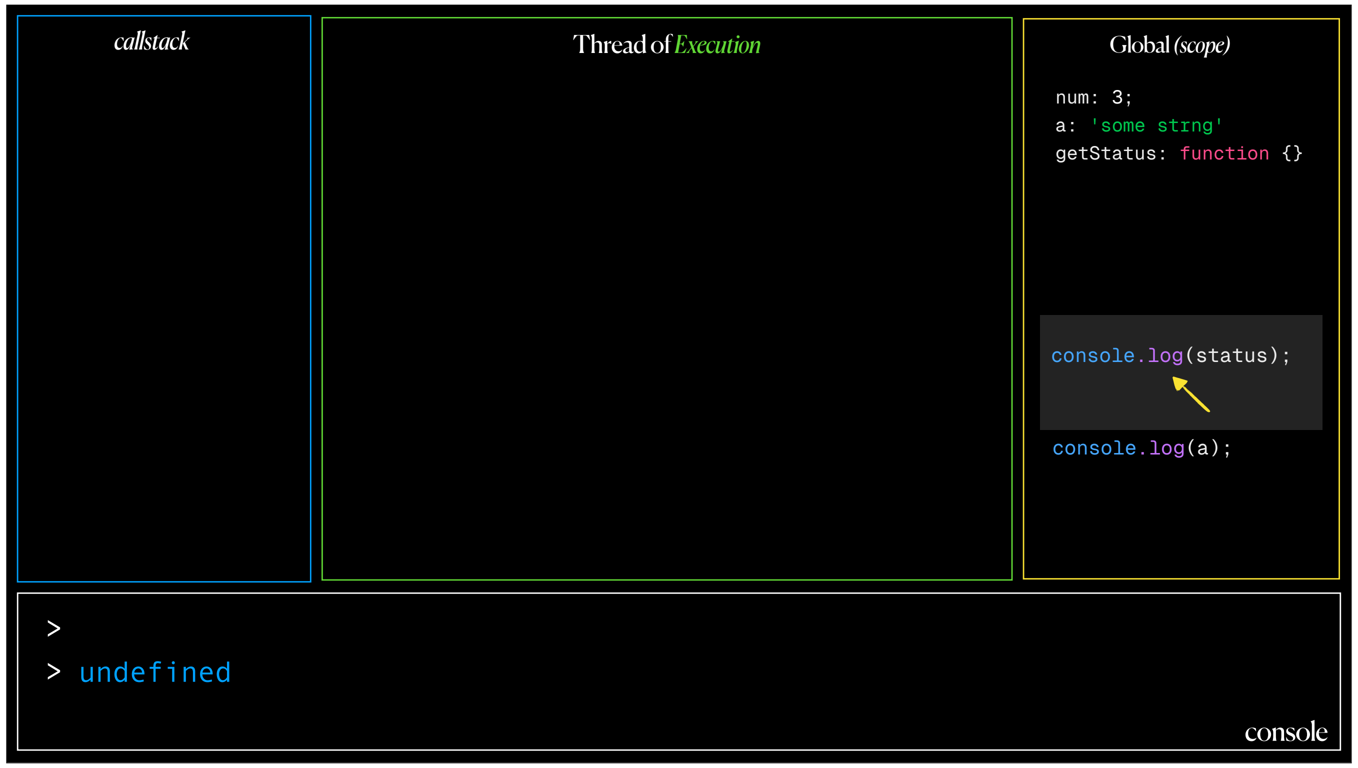
we were not aware till now.
(and in future it comes given that console.log will also go to call stack and have its place in the Thread of execution and only then it will show something on std output).
The function body of console.log is actually straight from the source-code of nodeJS.
>> after this gets executed, (and execution context is destroyed), we see in console ‘up some string’



```
num: 3;
a: 'some strng'
getStatus: function {}

console.log(status);

console.log(a);
```

```
> upsome string
>
```

Now the execution will go on for another console.log in the same way;
>> Which will print

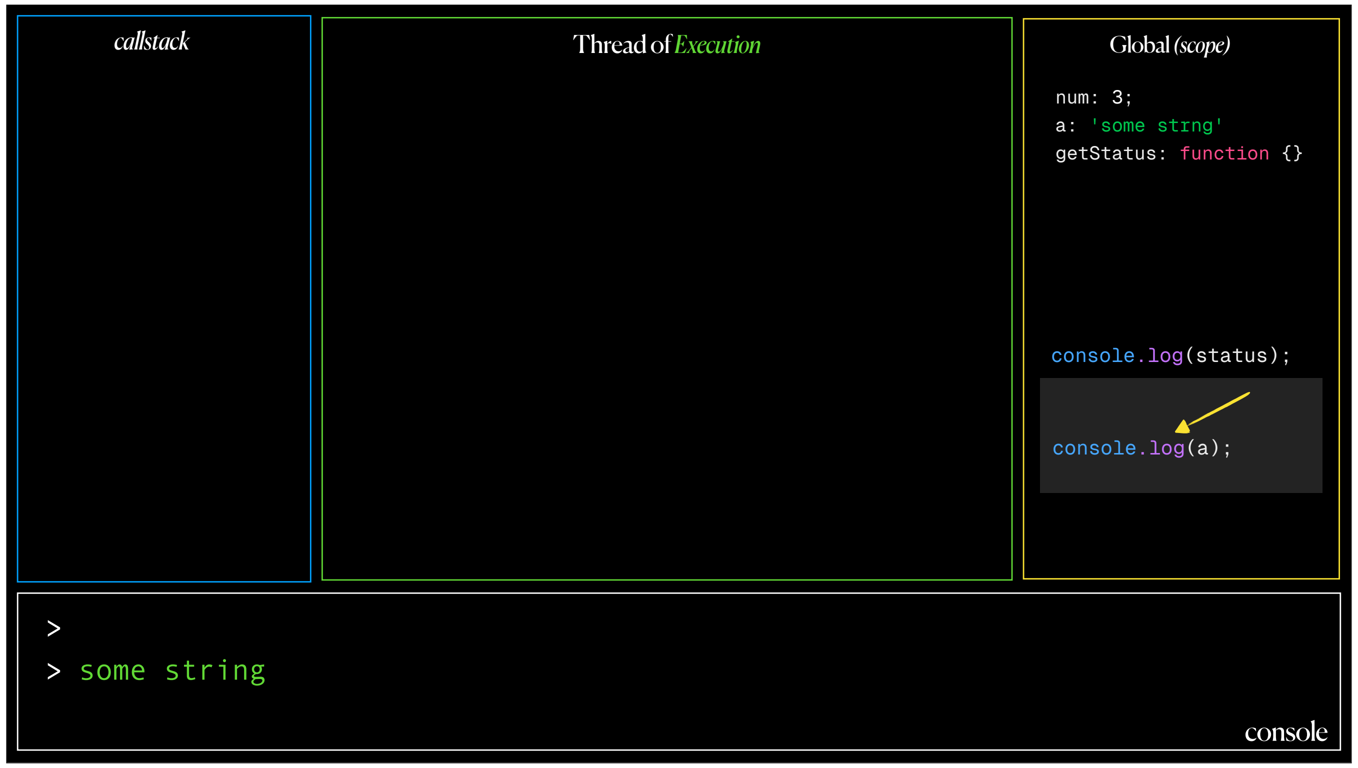


```
num: 3;
a: 'some strng'
getStatus: function {}
```

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

```
>
> undefined
```

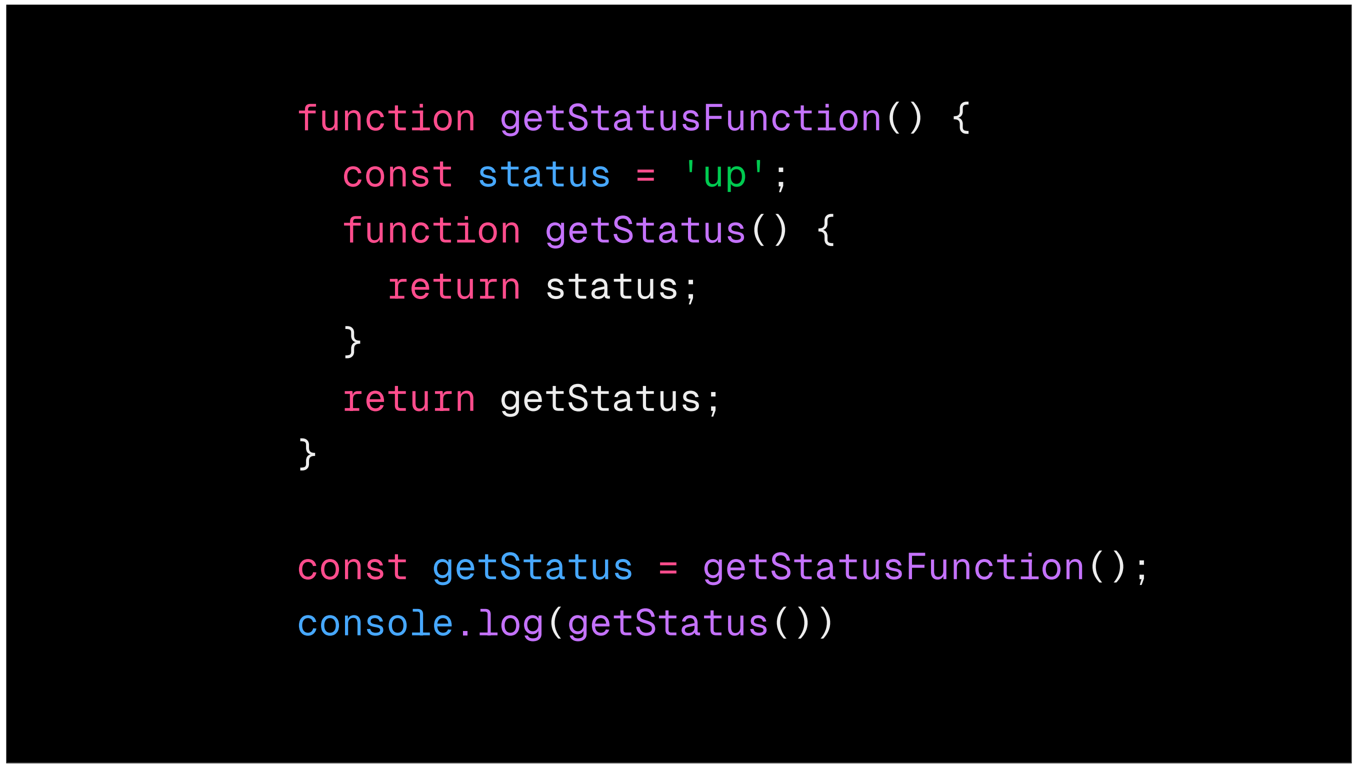
“undefined” because the variable “status” got destroyed along with the execution context of where it was defined and **we have no reference to its value.**
>> at last we will get



```
num: 3;
a: 'some strng'
getStatus: function {}

console.log(status);

console.log(a);
```

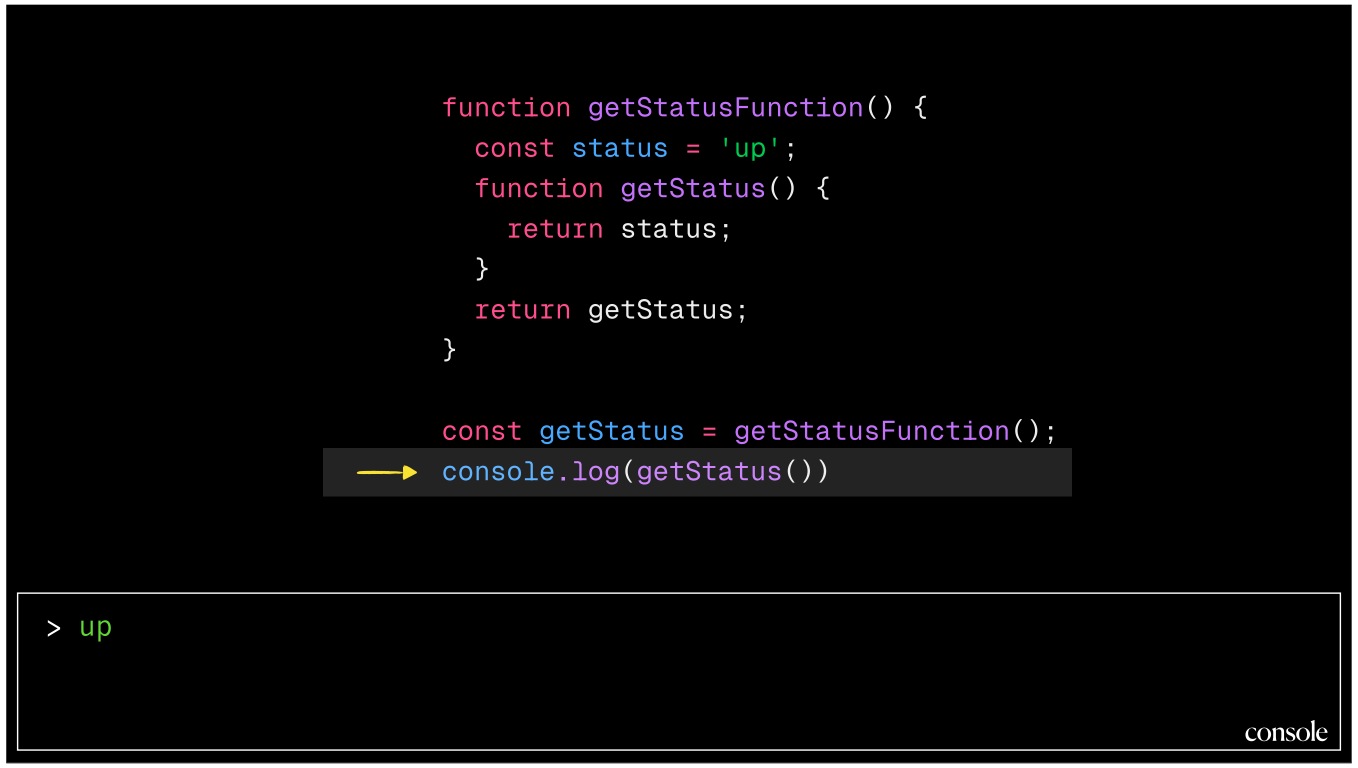
```
>
> some string
```

'some string' as it is in the console because. 'a' is right here in the current scope.



```
function getStatusFunction() {
  const status = 'up';
  function getStatus() {
    return status;
  }
  return getStatus;
}

const getStatus = getStatusFunction();
console.log(getStatus())
```

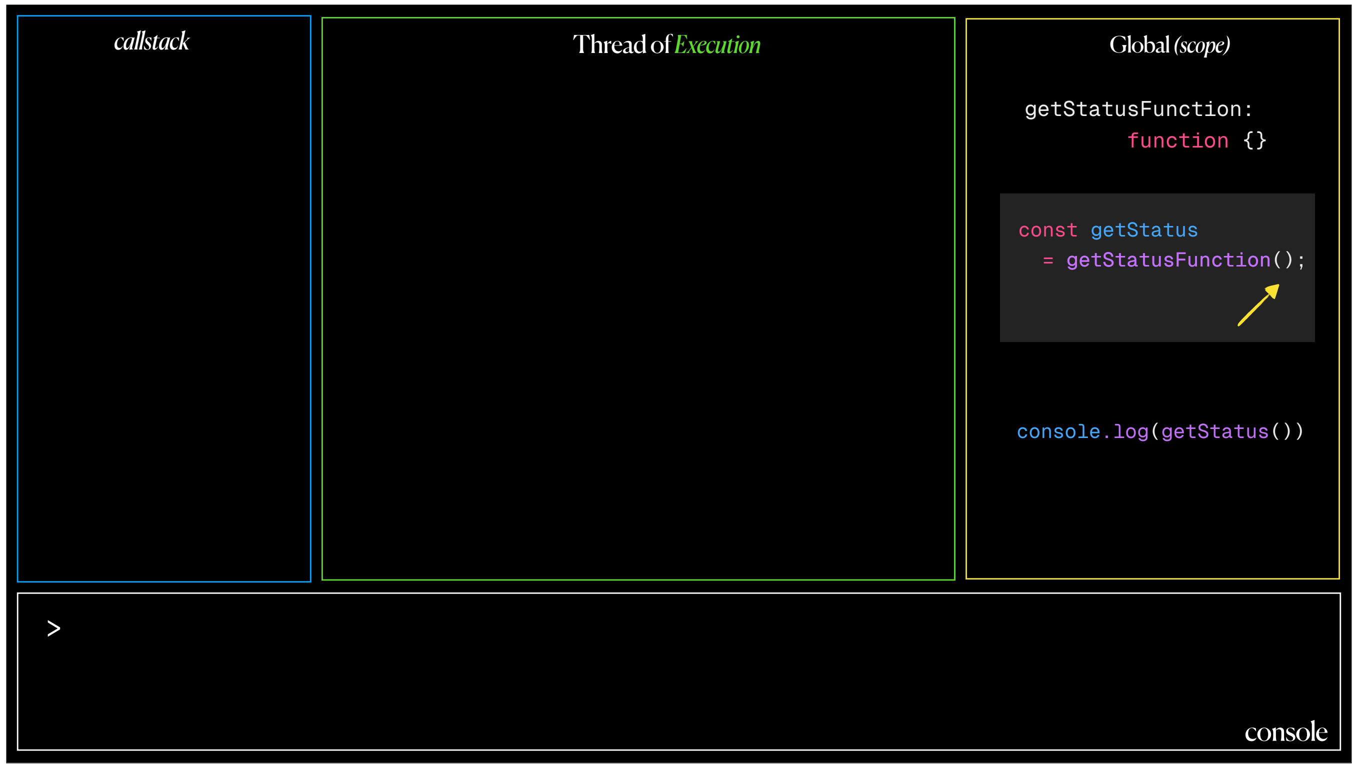
Now that your understanding is clear to some extent;
Lets take a look at another interesting example. (Take a photograph of this one..)
>> Unlike the previous problem, I would like to show you the result first for this one.



```
function getStatusFunction() {
  const status = 'up';
  function getStatus() {
    return status;
  }
  return getStatus;
}

const getStatus = getStatusFunction();
console.log(getStatus())
```

```
> up
```

In here when the execution ends and we reach the last console.log, what we get in console as you can see is ‘up’. Now if you think about what I recently said, the top execution context gets destroyed when we are done with that function; which means when getStatusFunction is called the value of status that was in its scope got destroyed as well; so how in the world: when we called the getStatus function at end; was it able to retrieve the value of status that is no-where to be present; ??
>> That was a lot to digest so lets go step by step

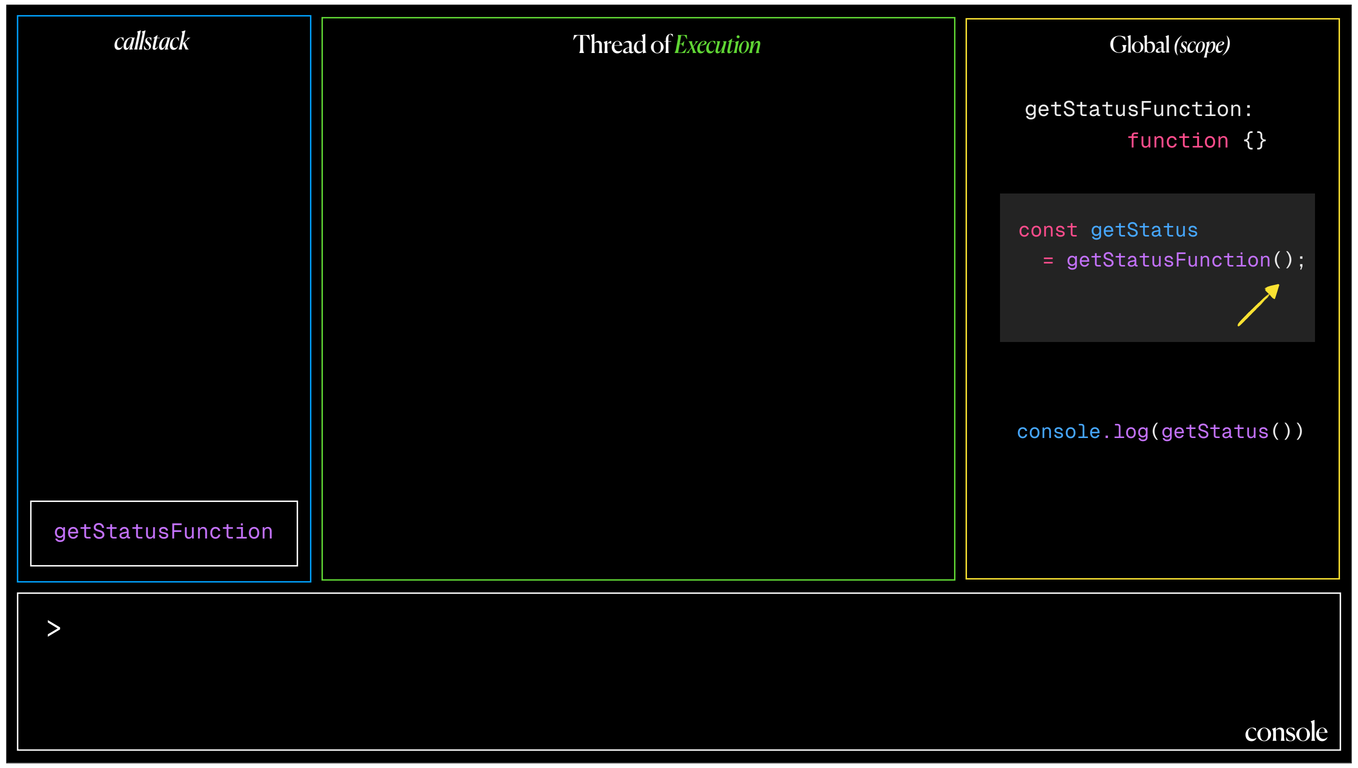


```
getStatusFunction:
        function {}
```

```
const getStatus
  = getStatusFunction();
```

```
console.log(getStatus())
```

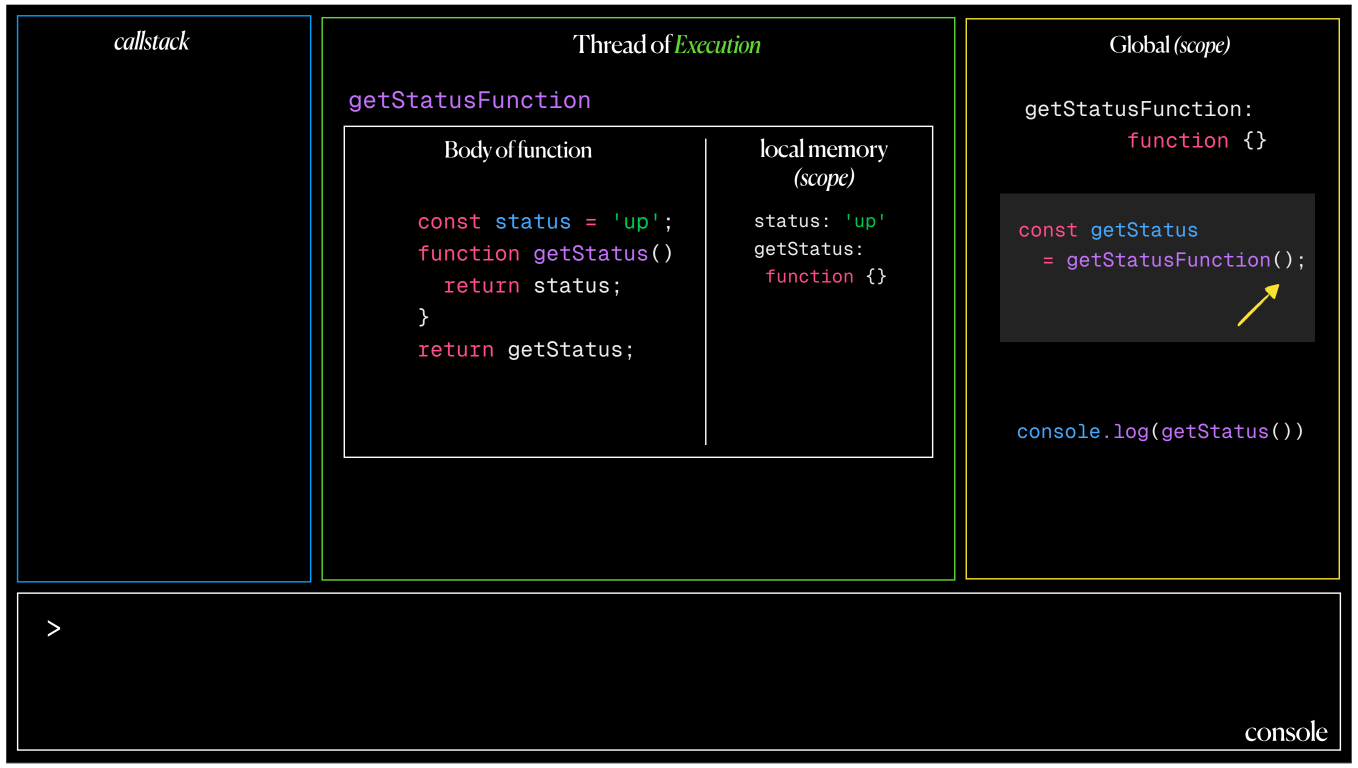
First our function declaration will get hoisted at top and second declaration as we see here is a const one: “lexical hoisting” > which is in TDZ before the “execution of its declaration”.
>> and then this function is called;



```
getStatusFunction:
        function {}

const getStatus
  = getStatusFunction();

console.log(getStatus())
```

```
getStatusFunction
```

It will get added to the callStack;
>> and our callStack will immediately put it



```
getStatusFunction
```

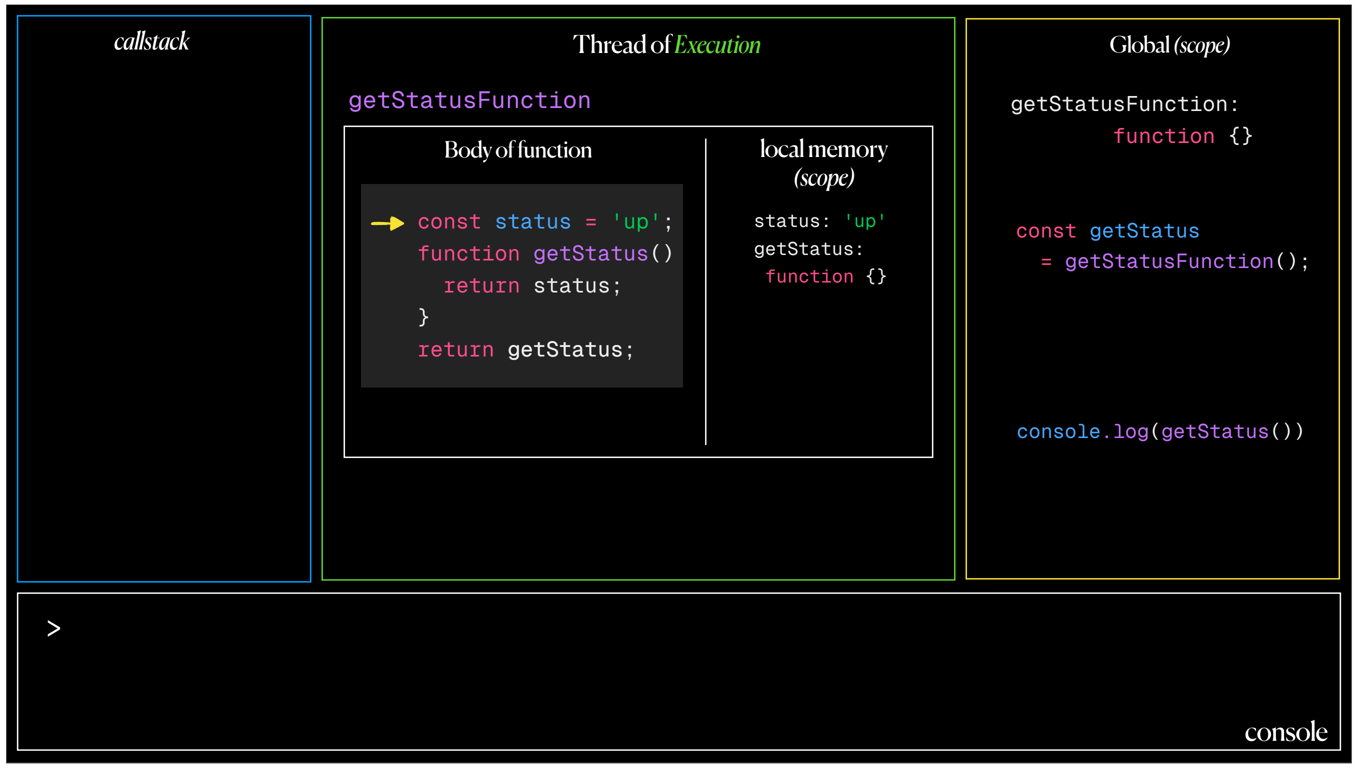
```
const status = 'up';
function getStatus()
  return status;
}
return getStatus;
```

```
status: 'up'
getStatus:
 function {}
```

```
getStatusFunction:
        function {}

const getStatus
  = getStatusFunction();

console.log(getStatus())
```

```
>
```

in the thread of execution (given it is empty).
>> And now the execution begins;



```
getStatusFunction
```

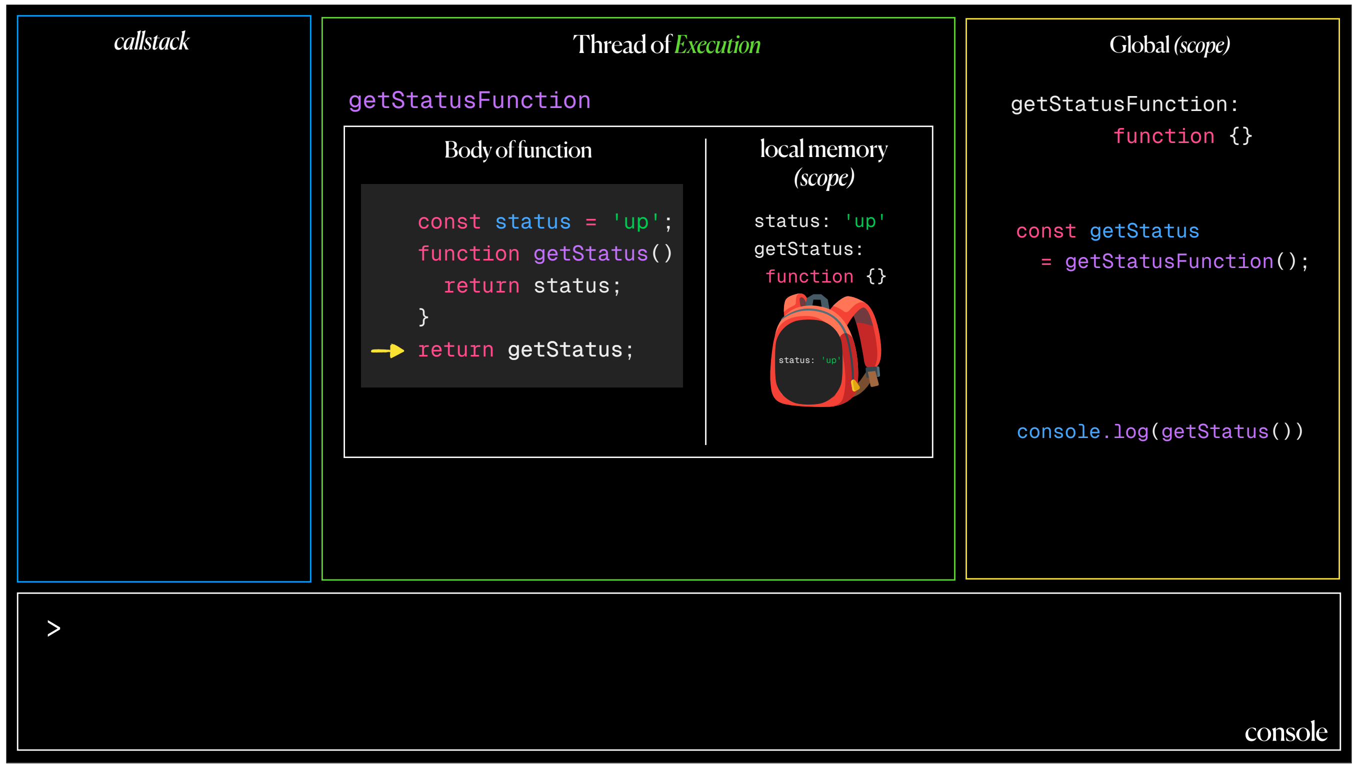
```
const status = 'up';
function getStatus()
  return status;
}
return getStatus;
```

```
status: 'up'
getStatus:
 function {}
```

```
getStatusFunction:
        function {}

const getStatus
  = getStatusFunction();

console.log(getStatus())
```

```
>
```

So everything is fine till now, the status gets initialised and the getStatus function gets hoisted. It is when the execution passes the declaration of the “getStatus” function,
>> a backpack gets created



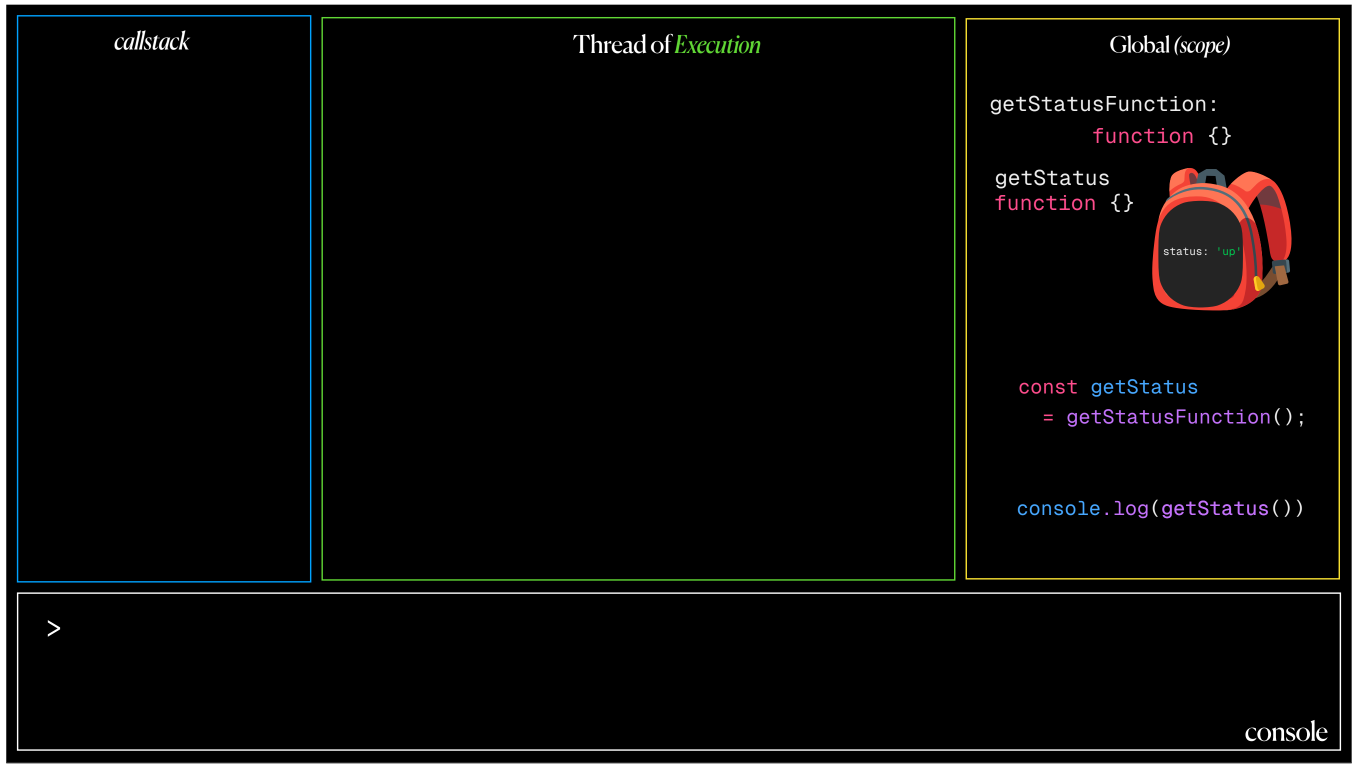
getStatusFunction

```
const status = 'up';
function getStatus()
  return status;
}
return getStatus;
```

```
status: 'up'
getStatus:
 function {}
```

```
getStatusFunction:
        function {}

const getStatus
  = getStatusFunction();

console.log(getStatus())
```

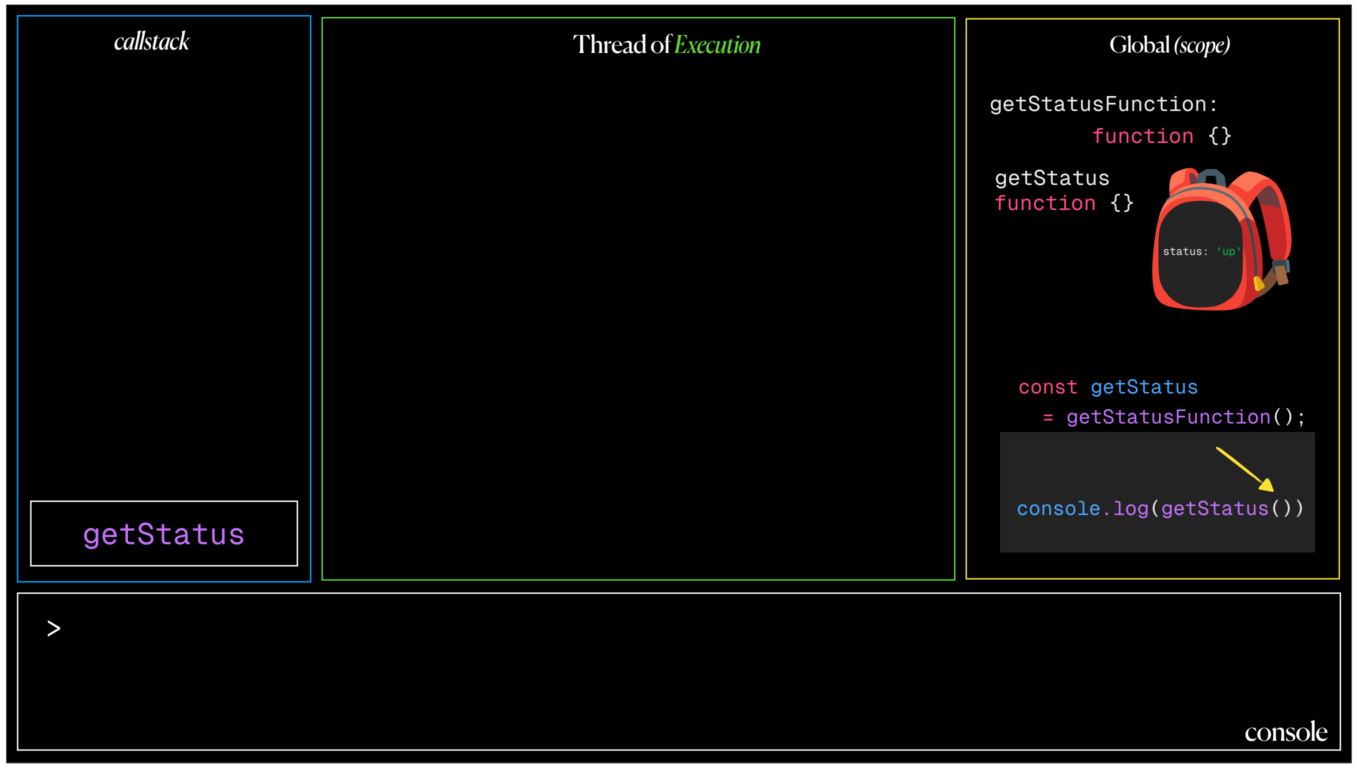
Not because the JS engine executed the code but it does something called "static analysis" to pass into this backpack only the declarations that it is using. Which means the declaration of getStatus function or any other variable (if have been there) will not be passed in the backpack.
The thing to note is that "this backpack was not created when hoisting took place" {parent to child warasat analogy.}
>> And this behaviour



```
getStatusFunction:
        function {}
getStatus
function {}

const getStatus
  = getStatusFunction();

console.log(getStatus())
```

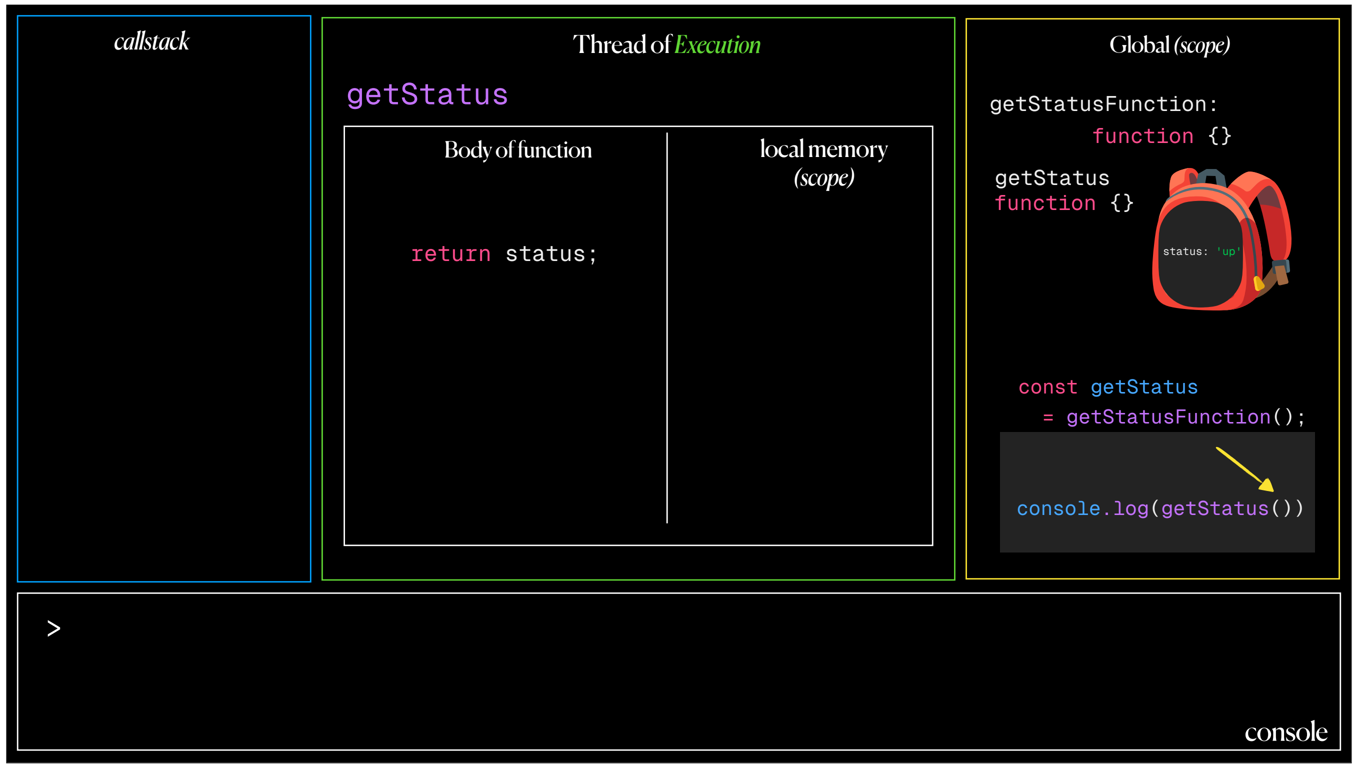
```
>
```

of having a persistent reference to the variables of parent. Is what we call as **Closure** in JS or in other words: “lexical scoping”
>> Moving on with the execution, the get status will now be called and be put in the callStack



```
const getStatus
  = getStatusFunction();
console.log(getStatus())
```

, given that our callstack is empty;
>> It will be move to the thread of execution

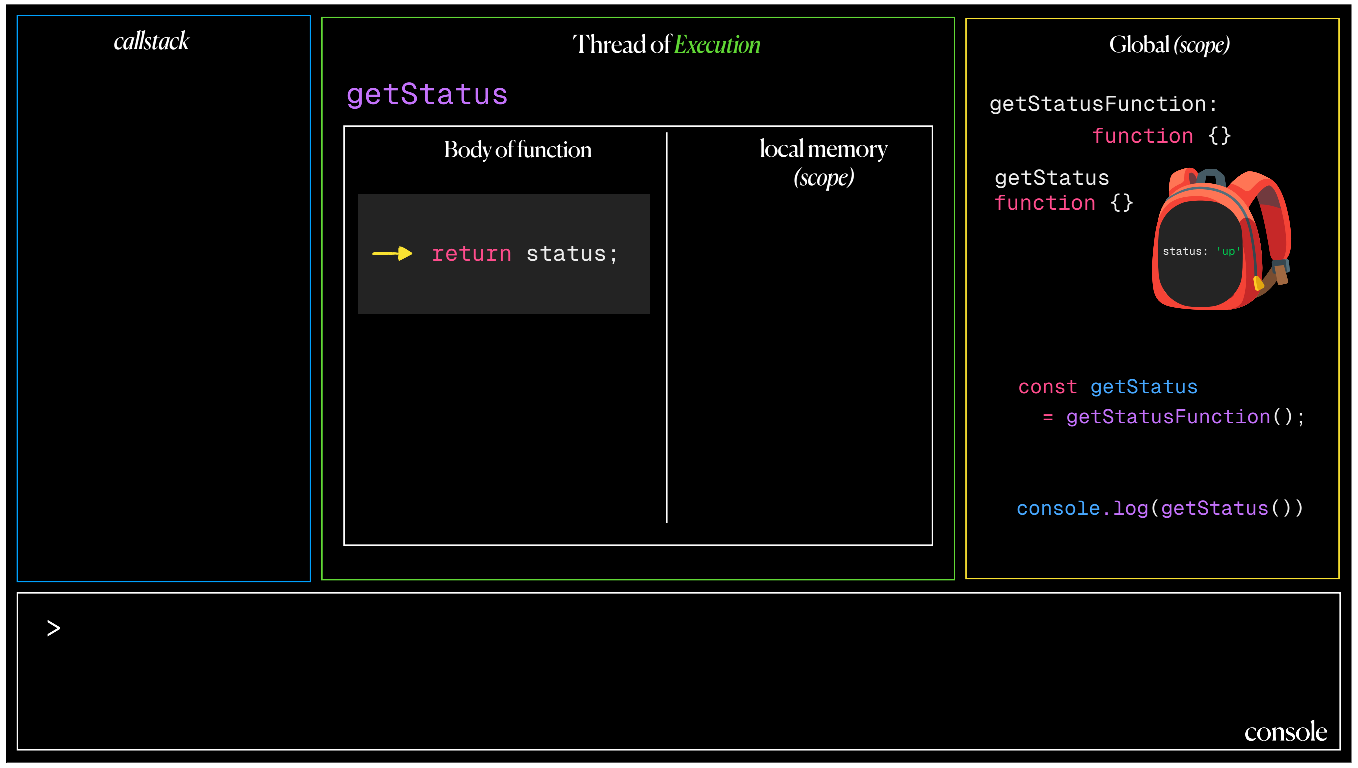


```
return status;
```

```
const getStatus
  = getStatusFunction();
console.log(getStatus())
```

```
>
```

>> where we will now continue with its execution

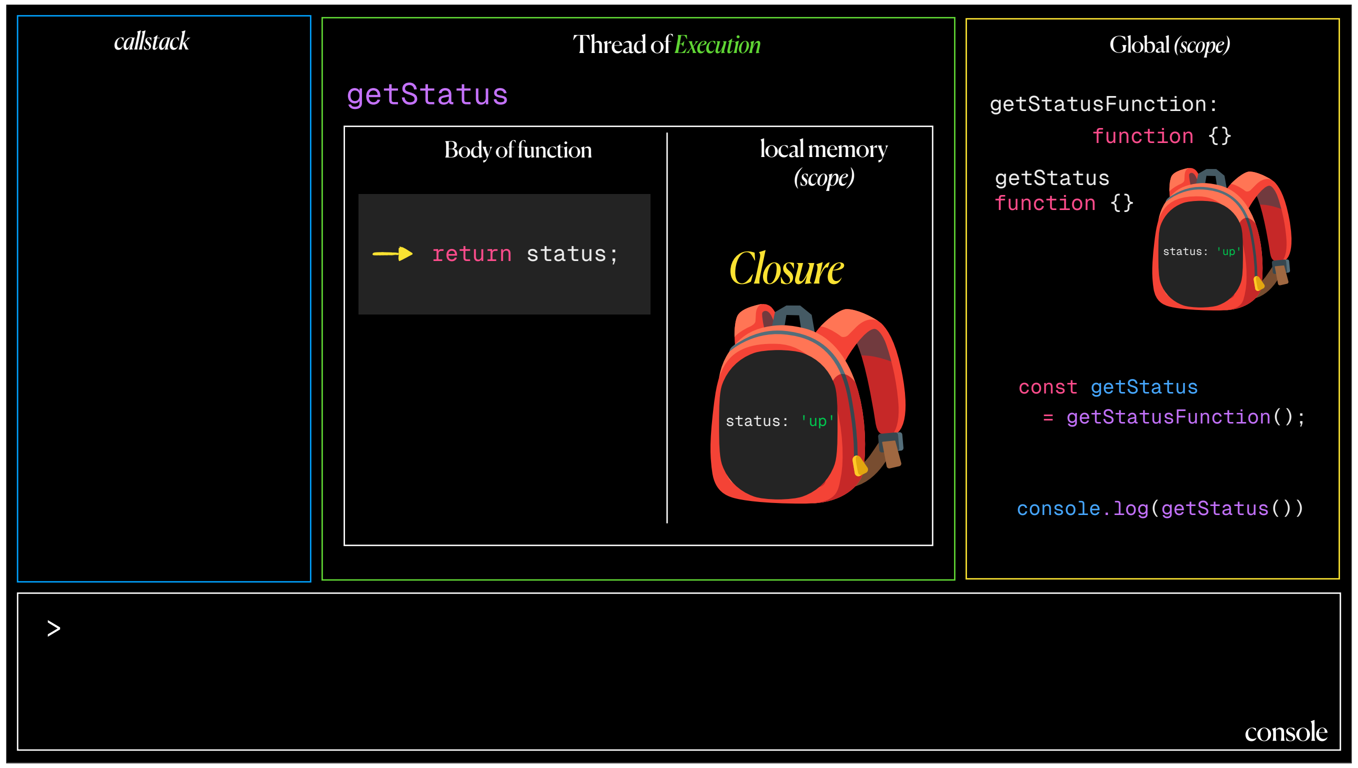


```
return status;
```

```
getStatusFunction:
    function {}
getStatus
function {}

const getStatus
  = getStatusFunction();

console.log(getStatus())
```

```
>
```

As you see the “getStatus function” returns the “status” variable and it was declared in the execution context of getStatusFunction which was just destroyed;
So here comes **Closure** to rescue and says: “Here is something left by your parents; when they were dying cz they thought you might need it one day.” So parents in JS take good care of their children.
>> and now from its lexical surrounding, it obtains this variable.

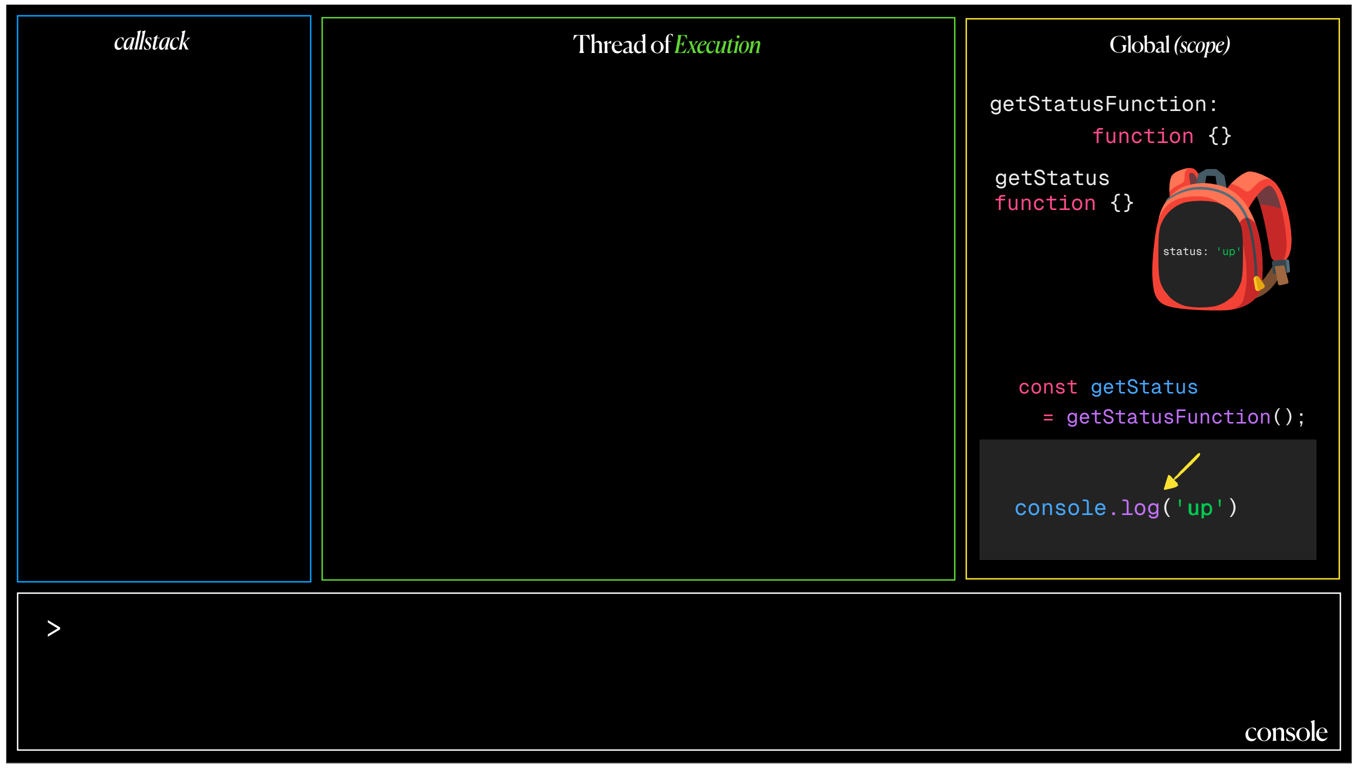


```
return status;
```

```
const getStatus
  = getStatusFunction();

console.log(getStatus())
```

```
>
```

and that is how “lexical scoping” works: in which the parser resolves the variable names when functions are nested.
>> and with that the execution of this function call comes to an end and the value ‘up’ gets returned

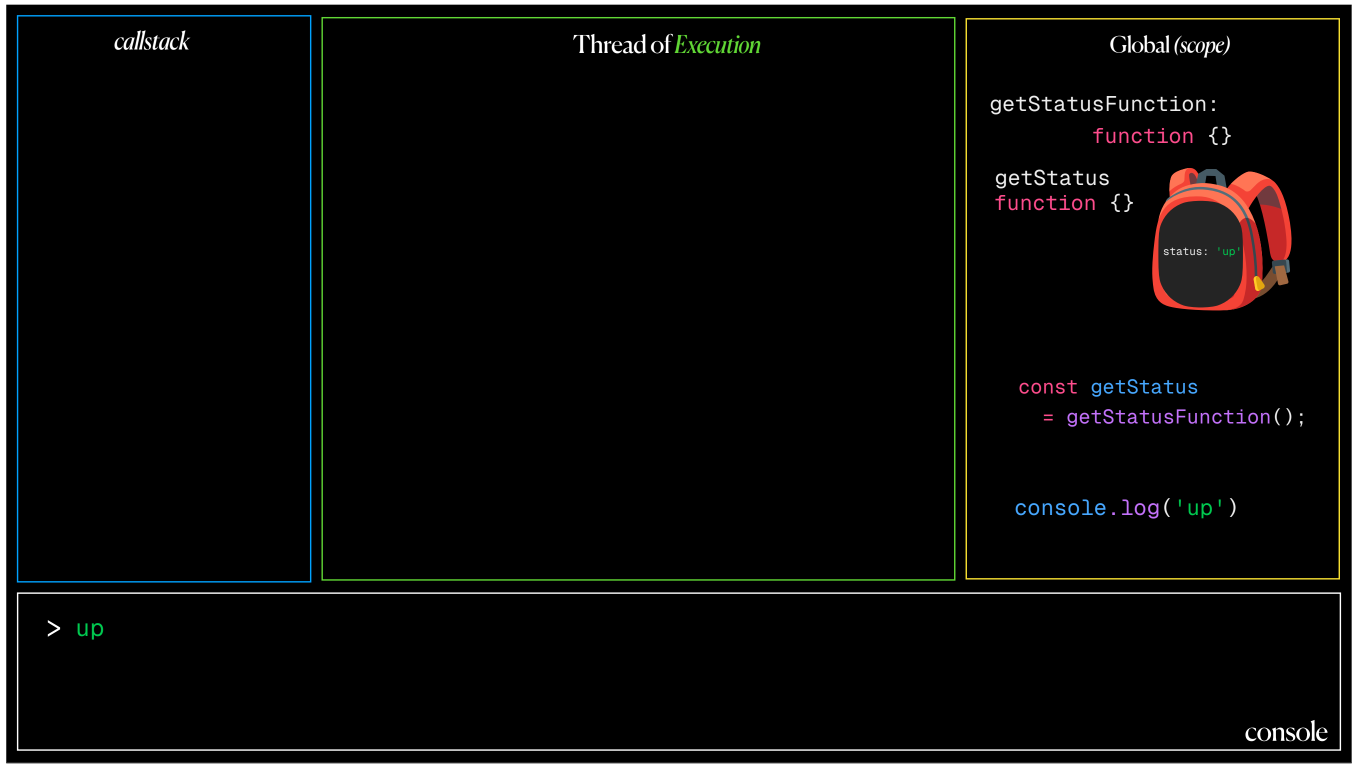


```
const getStatus
   = getStatusFunction();
```

```
console.log('up')
```

as an argument to the console.log function

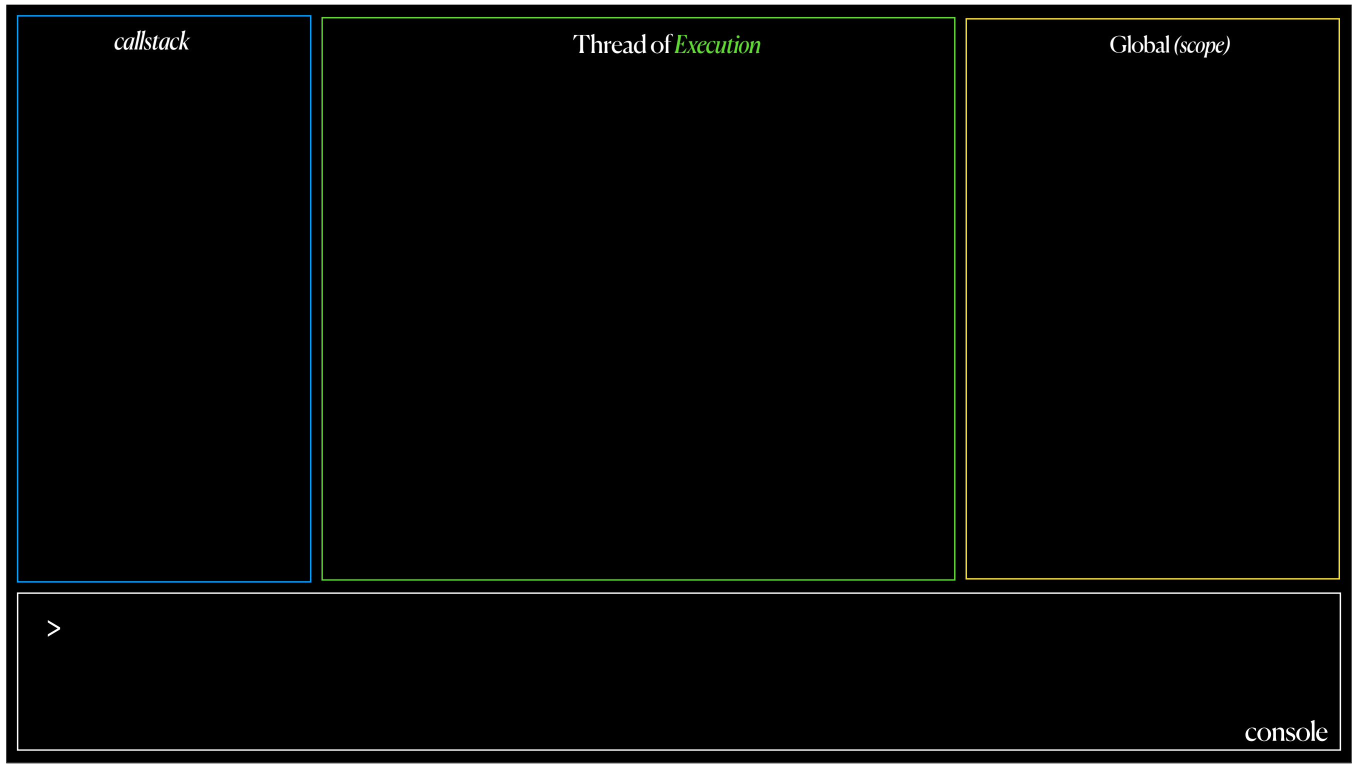
And the log function will again go to callStack and get executed;
>> which will result in ‘up’ getting logged to the console



```
getStatusFunction:
        function {}
getStatus
function {}
status: 'up'

const getStatus
  = getStatusFunction();

console.log('up')
```

```
> up
```

getting logged to the console
>> And our process / program will end



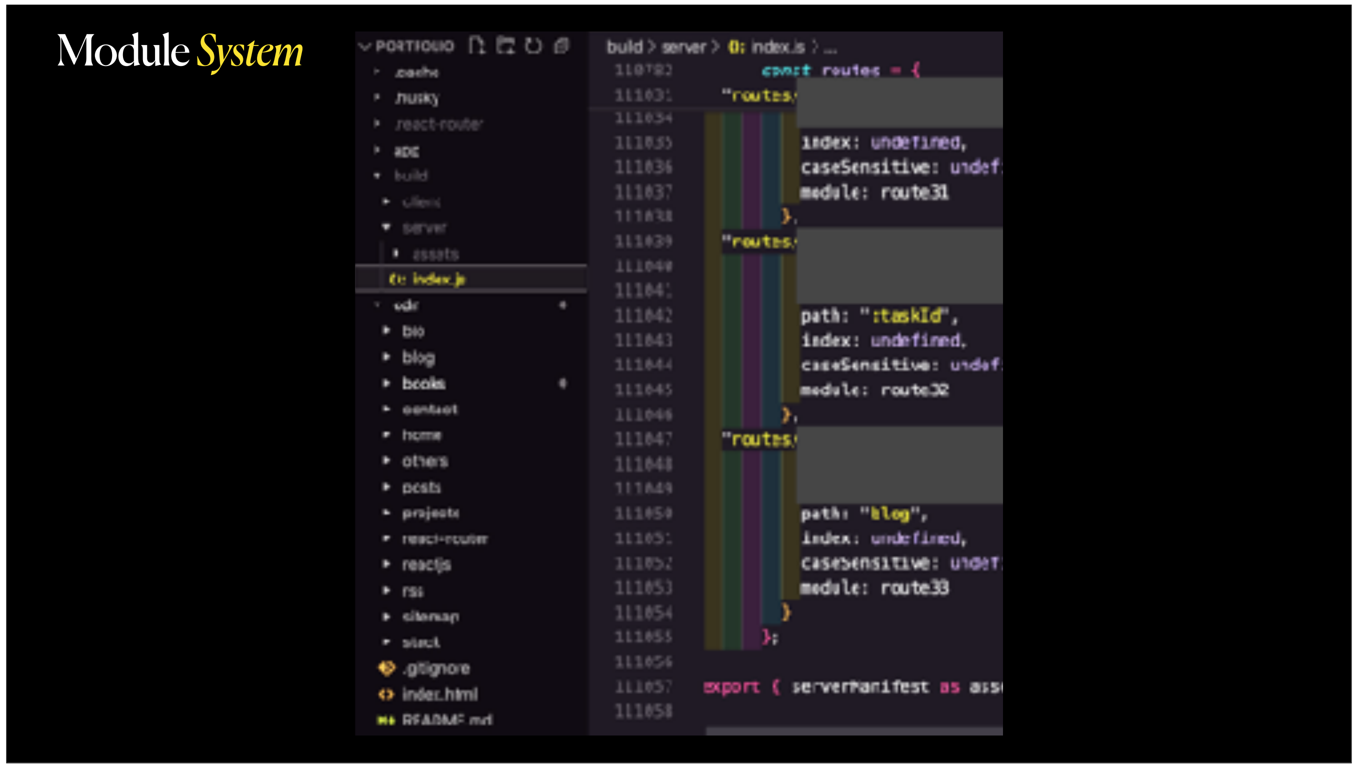
and at this time the OS itself will reclaim the memory that was allocated with this process;
I hope you now have a decent understanding of how JS works.
>> Now I would like to call Amaan here to discuss with us how modules work in JS.

Module *System*

We will look into >*why modules exist*; >how do they differ; > which ones to use.
Lets leave module systems for a while and think about the work we do.
So what is that we do… anyone………
>> I guess rats to rescue

Write *Programs*

So at the end of the day, we write programs to make systems, api-clients, applications, and in case of JS, even operating systems.
>> The thing about programs

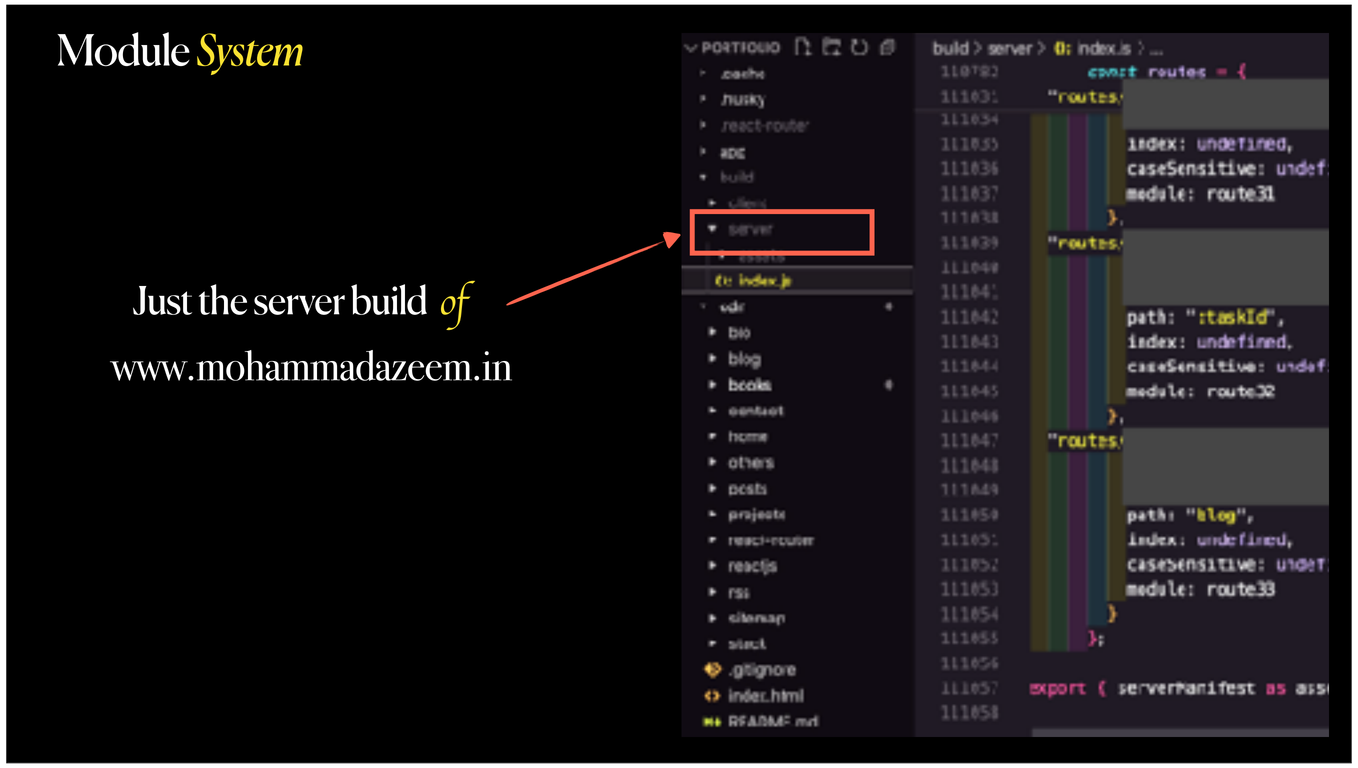


Module *System*

```
PORTFOLIO
  .cache
  .husky
  .react-router
  app
  build
    client
    server
      assets
      index.js
  cdn
    bio
    blog
    books
    contact
    home
    others
    posts
    projects
    react-router
    reactjs
    rss
    sitemap
    stack
  .gitignore
  index.html
  README.md
```

```
build > server > index.js > ...
110782        const routes = {
111031    "routes/
111034
111035            index: undefined,
111036            caseSensitive: undef
111037            module: route31
111038          },
111039    "routes/
111040
111041
111042            path: ":taskId",
111043            index: undefined,
111044            caseSensitive: undef
111045            module: route32
111046          },
111047    "routes/
111048
111049
111050            path: "blog",
111051            index: undefined,
111052            caseSensitive: undef
111053            module: route33
111054          }
111055        };
111056
111057  export { serverManifest as ass
111058
```

is that they can get pretty big in no time, for example
>> lets consider the source file of my personal site: mohammadazeem.in



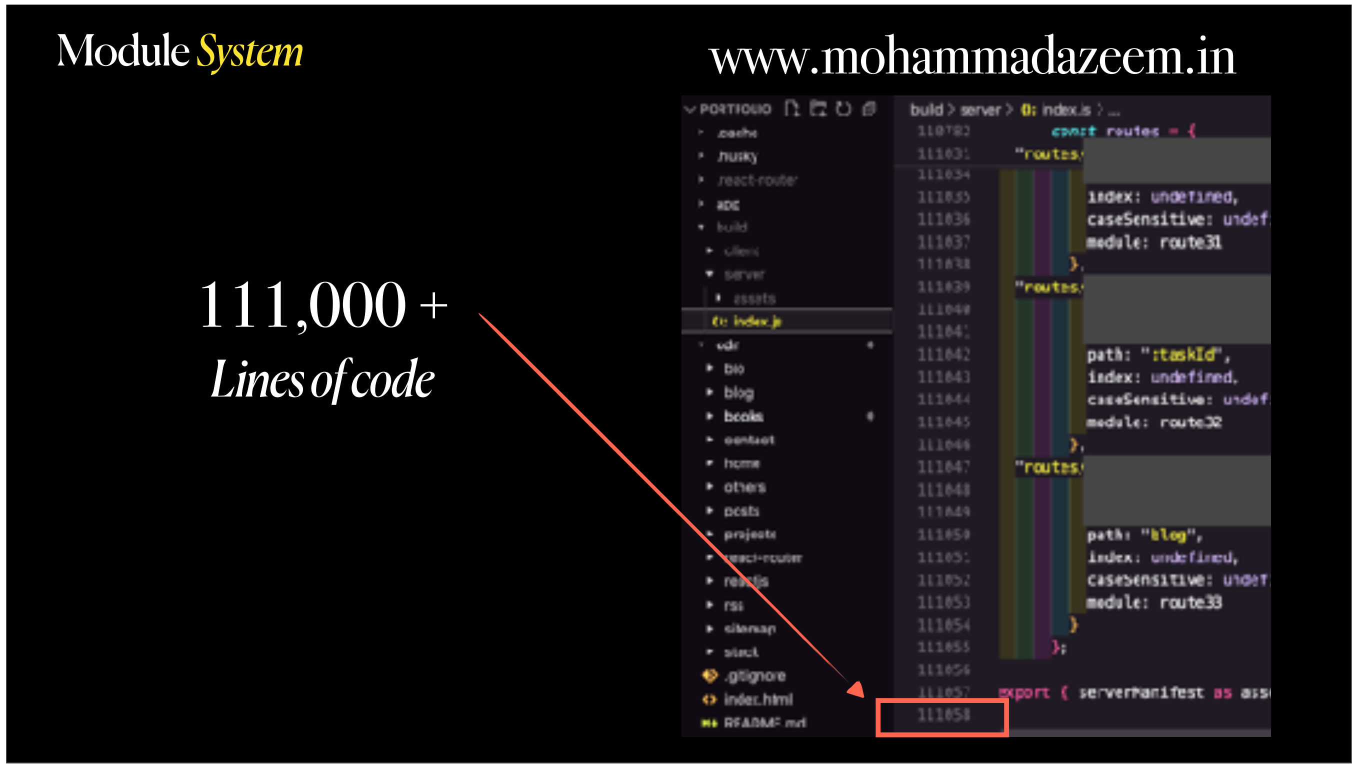
Module *System*

Just the server build *of*

www.mohammadazeem.in

```
          const routes = {
    "routes/[redacted]
        index: undefined,
        caseSensitive: undef
        module: route31
      },
    "routes/[redacted]
        path: ":taskId",
        index: undefined,
        caseSensitive: undef
        module: route32
      },
    "routes/[redacted]
        path: "blog",
        index: undefined,
        caseSensitive: undef
        module: route33
      }
    };

export { serverManifest as ass
```

This is just the server side build we are talking about; not the client side bundles;
>> Now as you can see that



Module *System*

www.mohammadazeem.in

111,000 +

Lines of code

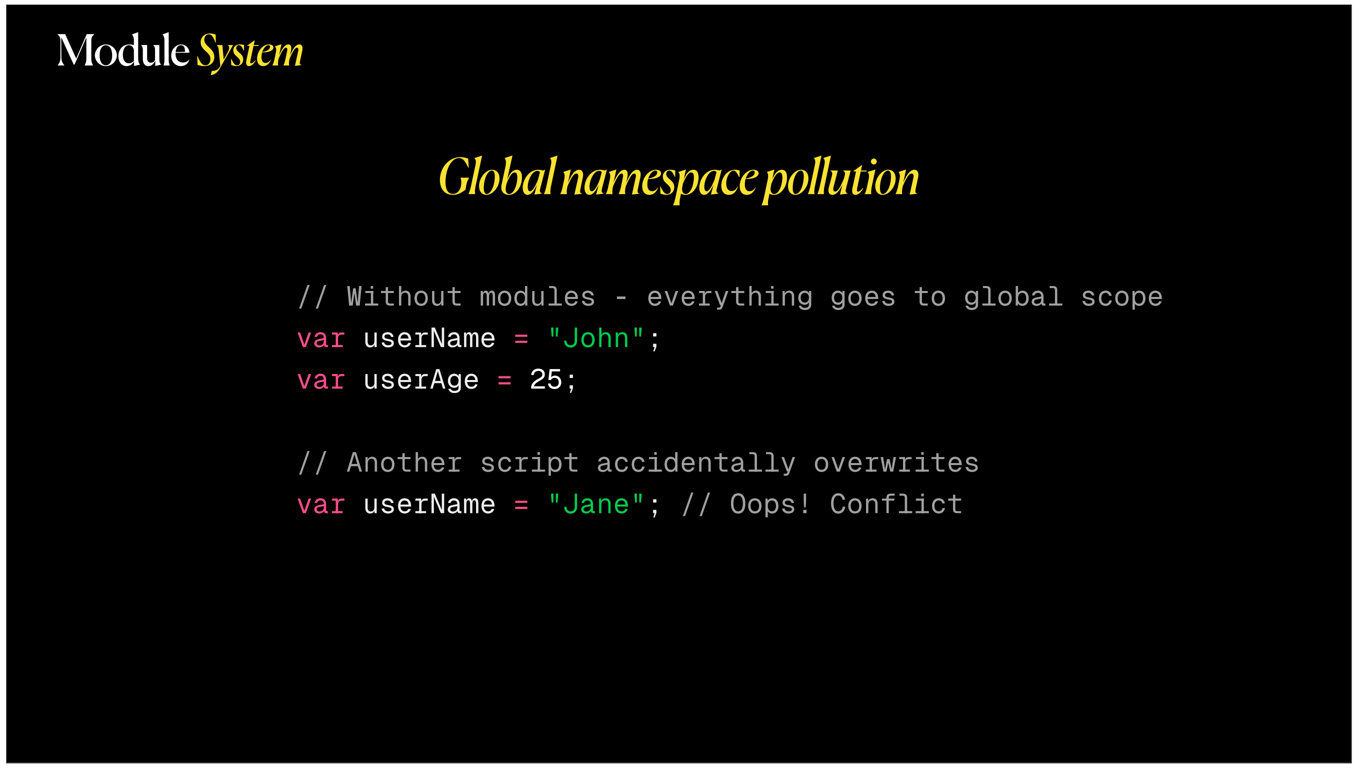
```
const routes = {
  "routes/
      index: undefined,
      caseSensitive: undef
      module: route31
    },
  "routes/
      path: ":taskId",
      index: undefined,
      caseSensitive: undef
      module: route32
    },
  "routes/
      path: "blog",
      index: undefined,
      caseSensitive: undef
      module: route33
    }
};

export { serverManifest as ass
```

the server file here itself exceeds 111k (one hundred thousand) LinesOfCode. That is a LOT.
And believe me when I say it;
>> it is a nightmare to maintain a file size of even 2k (two thousand) LOCs.

Module *System*

Development nightmare 👻

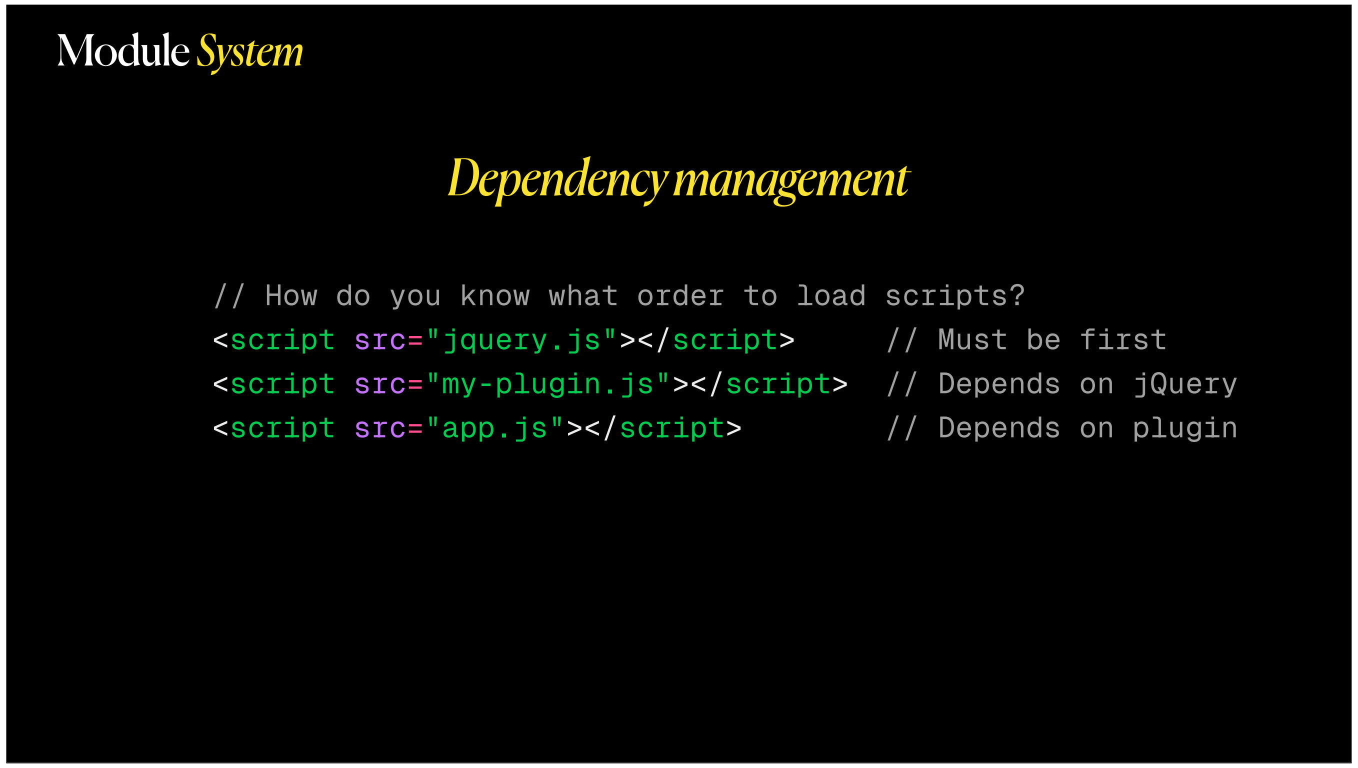
Now that adds the “Development nightmare” as one of the reason we can’t write everything in single file. While it is possible that we can split files even without a module system. But that does nothing but pile up the reasons to have a module system.
>> Such as


Module *System*

Global namespace pollution

```
// Without modules - everything goes to global scope
var userName = "John";
var userAge = 25;

// Another script accidentally overwrites
var userName = "Jane"; // Oops! Conflict
```

Global namespace pollution;
Where one has to deal with lots of overhead to remb: what gets overridden and what not: which is other way of saying “not-scalable”
>> And one that y’all doing web-development, might have encountered;

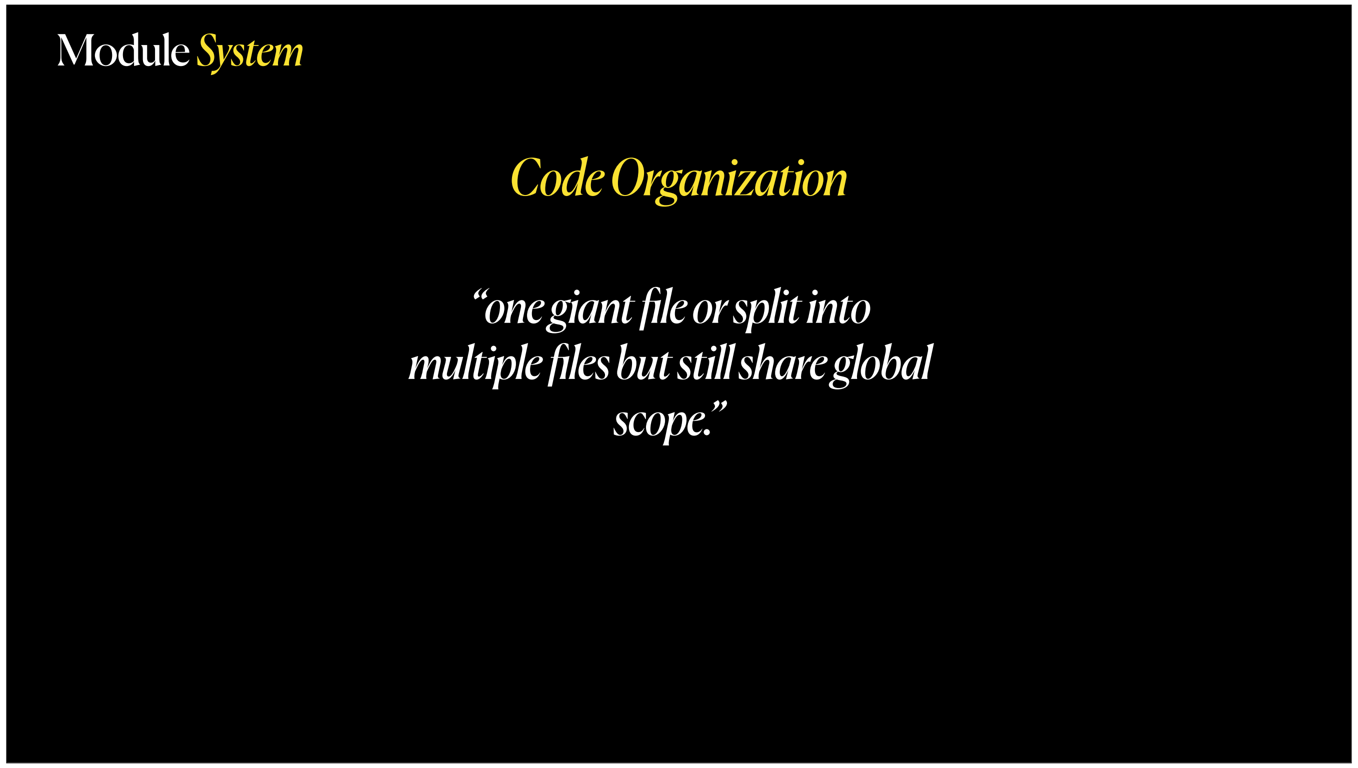


Module *System*

Dependency management

```
// How do you know what order to load scripts?
<script src="jquery.js"></script>     // Must be first
<script src="my-plugin.js"></script>  // Depends on jQuery
<script src="app.js"></script>        // Depends on plugin
```

The Dependency management.
Where in this case for example; if you forgot to import jquery, everything will break;
even if app.js below uses no functionality that depends on jquery.
Even though using script tags are module systems of their own but they too bring other problems >> such as



Module *System*

Code Organization

“one giant file or split into multiple files but still share global scope.”

Code organization cz Without modules, you either put everything in one giant file or split into multiple files but still share global scope.
>> There are other

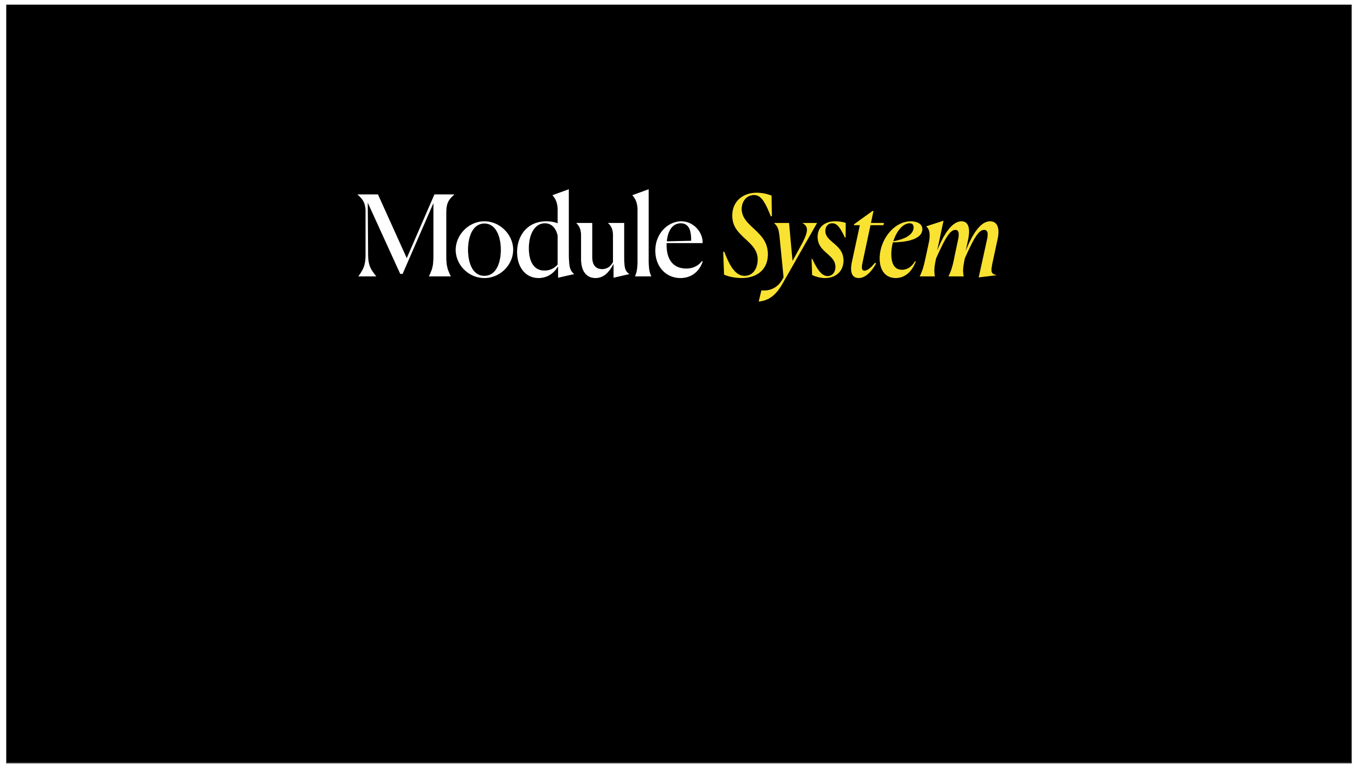
Code Organization

Dependency management

Global namespace pollution

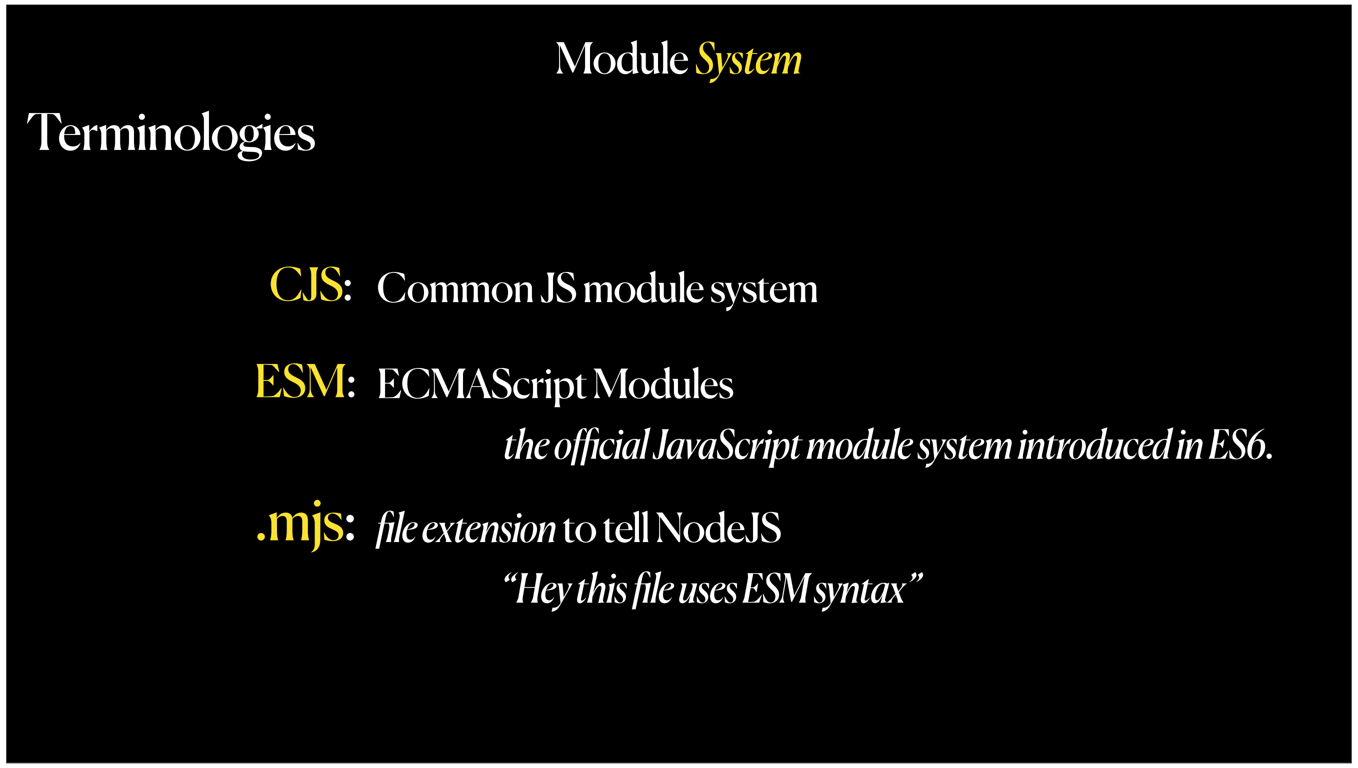
Development nightmare 👻

problems as well that pile up and
>> scream that a “Module system is needed”



Module *System*

So lets talk about module system.
>> Before diving into this topic



Module *System*

Terminologies

CJS: Common JS module system

ESM: ECMAScript Modules

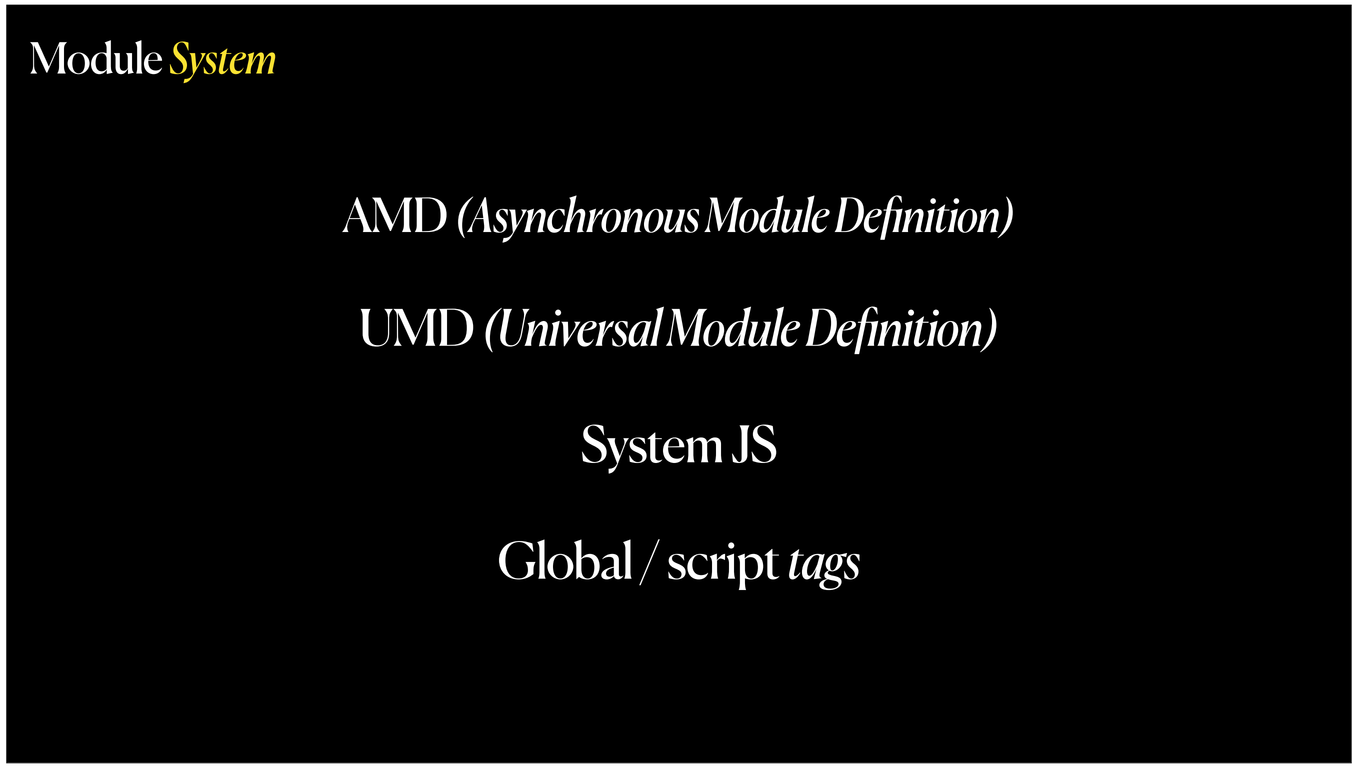
the official JavaScript module system introduced in ES6.

.mjs: *file extension* to tell NodeJS

"Hey this file uses ESM syntax"

there are some terminologies, I want you to keep in your mind:
"None of them you need to know and > it is totally ok"
- CJS: stands for CommonJS module system
- ESM: is short form of ECMAScript Modules: which is the official JS module system introduced in ES6 (6th version of the ECMAScript language specification; in the year 2015)
- .mjs: it is a file extension that we use to tell NodeJS that hey this file uses ESM syntax.

>> Now like we saw



Module *System*

AMD *(Asynchronous Module Definition)*

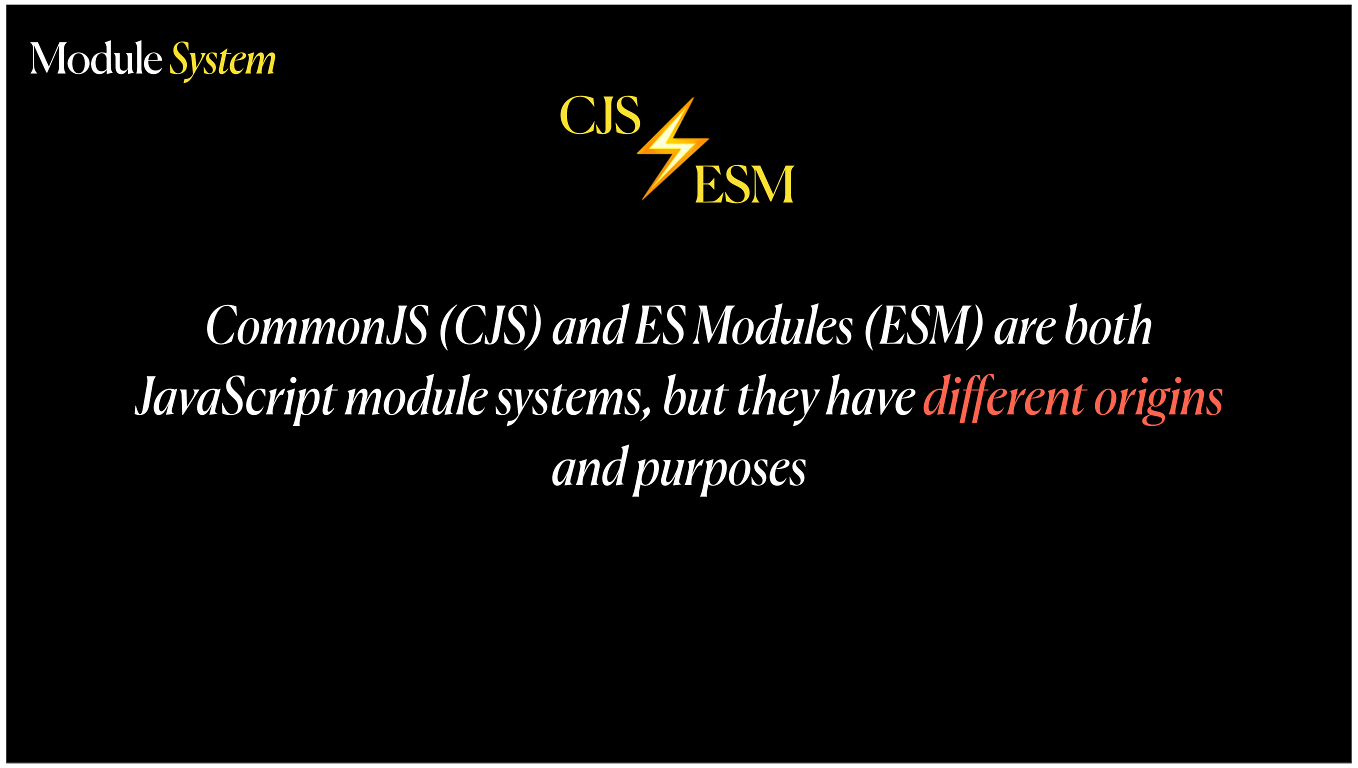
UMD *(Universal Module Definition)*

System JS

Global / script *tags*

The need to have a module system is not some recent problem, in JS community, it has been there since we started making programs. So several attempts were made to achieve a module system;
Some of them worth knowing are:
- [click] AMD which stands for Asynchronous module definition
- [click] UMD (Universal Module Definition)
- [click] SystemJS
- [click] Global / script tags “which we just saw. *..mob*”

Now this is the legacy stuff we generally not care as of today (especially when doing development)
>> What we do care about is these two guys

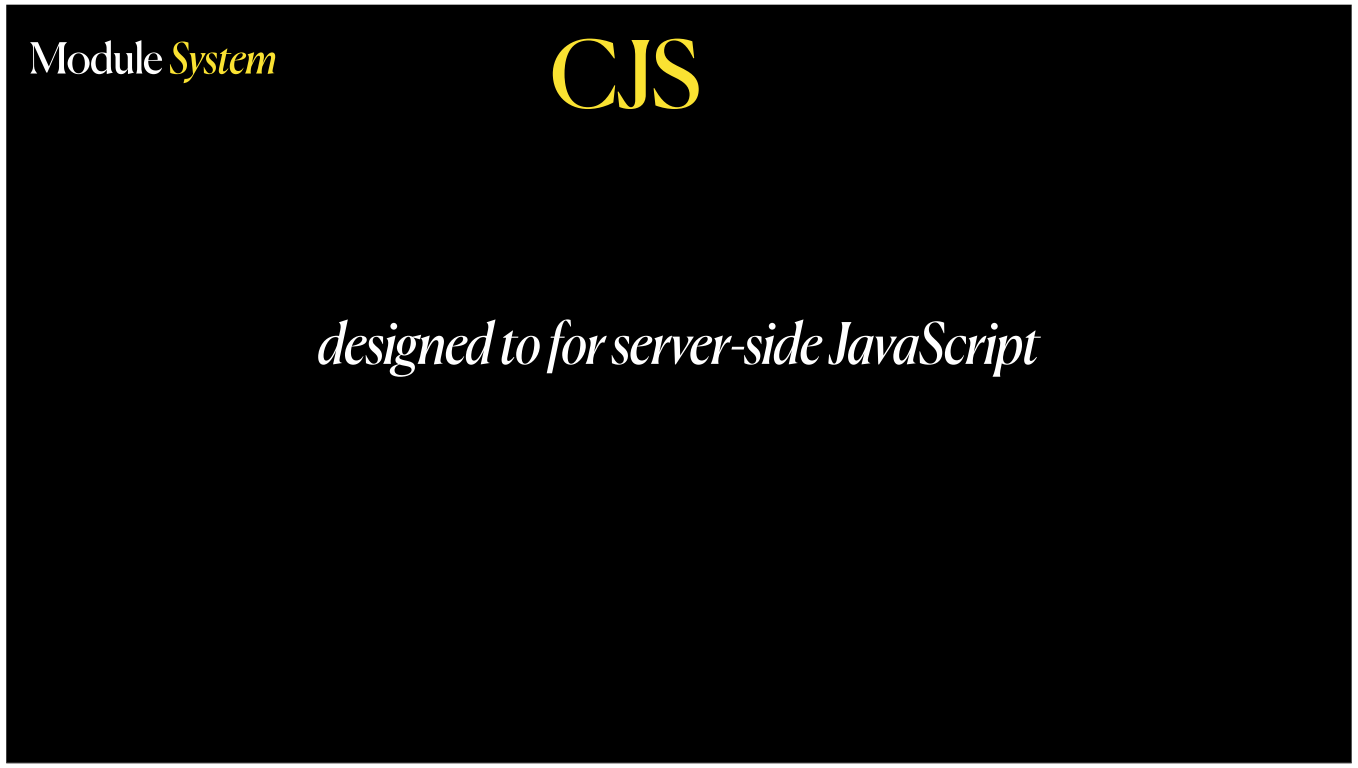


Module *System*

CJS ⚡ ESM

CommonJS (CJS) and ES Modules (ESM) are both JavaScript module systems, but they have different origins and purposes

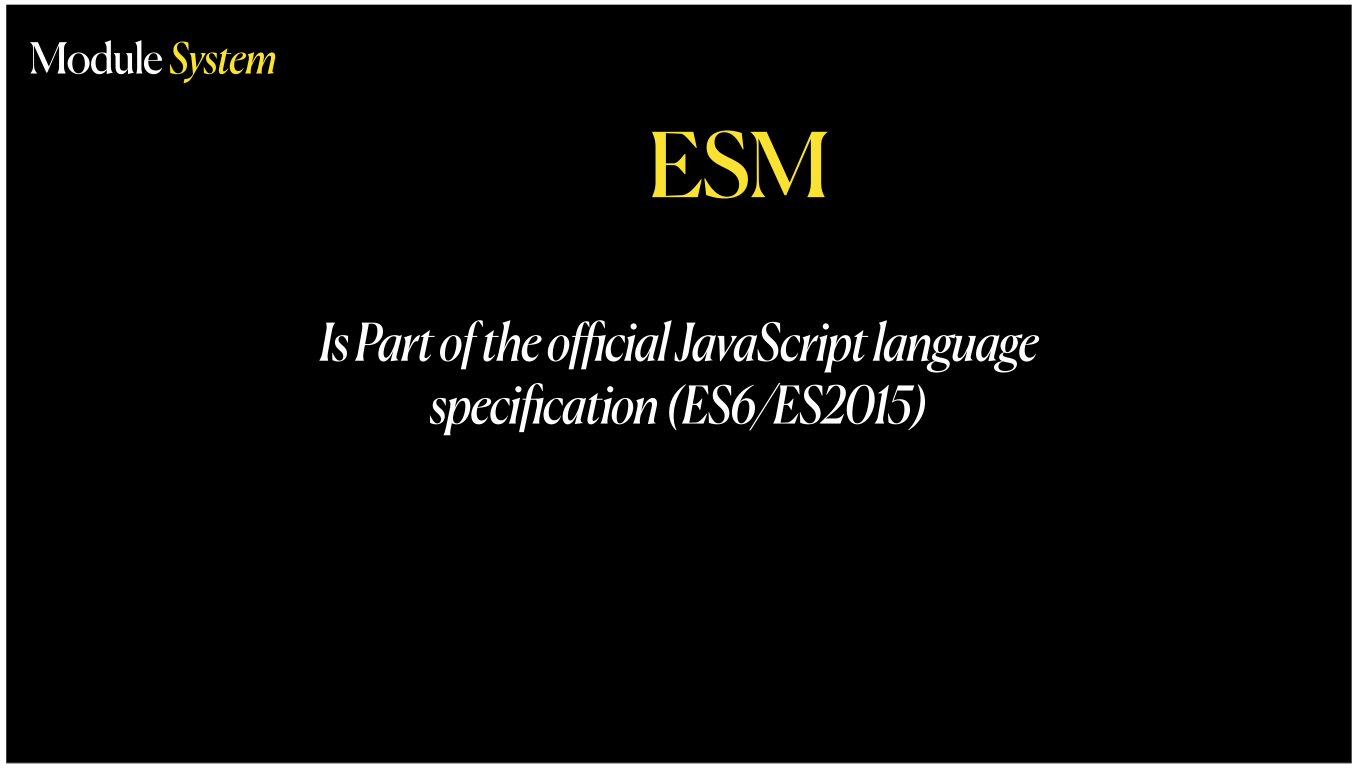
CommonJS and ESM
So they both first of all are JavaScript module systems as we know them. What makes them different is there origins and purpose they were built for and of course the implementation.
>> Each JS runtime has a module system.


Module *System*

CJS

designed to for server-side JavaScript

CJS was designed for server-side JavaScript (Node.js)
>> where as



Module *System*

ESM

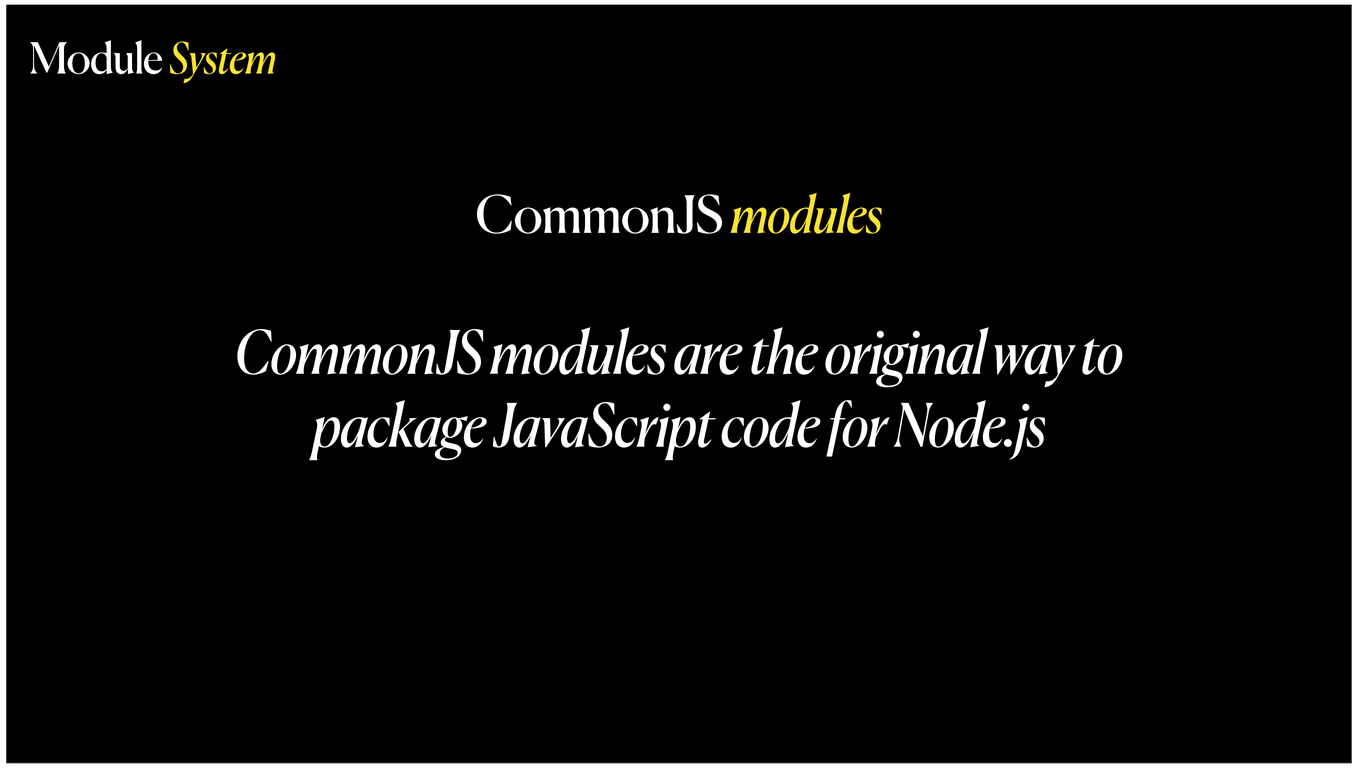
Is Part of the official JavaScript language specification (ES6/ES2015)

ES Modules (ESM): are Part of the official JavaScript language specification (ES6/ES2015)
So ESM is a native JavaScript language feature, while CommonJS is more of a standard that became widely adopted, especially in & because of Node.js

Module *System*

CJS ⚡ ESM

Node.js now supports both systems, and you can even use them together with some configuration
Now that we have a little Idea of what problem Module systems solves and why they exist.
Lets move on to see what differs CJS from ESM and how they are implemented.
We will first talk about CJS and then move on to ESM.
>> So lets go.

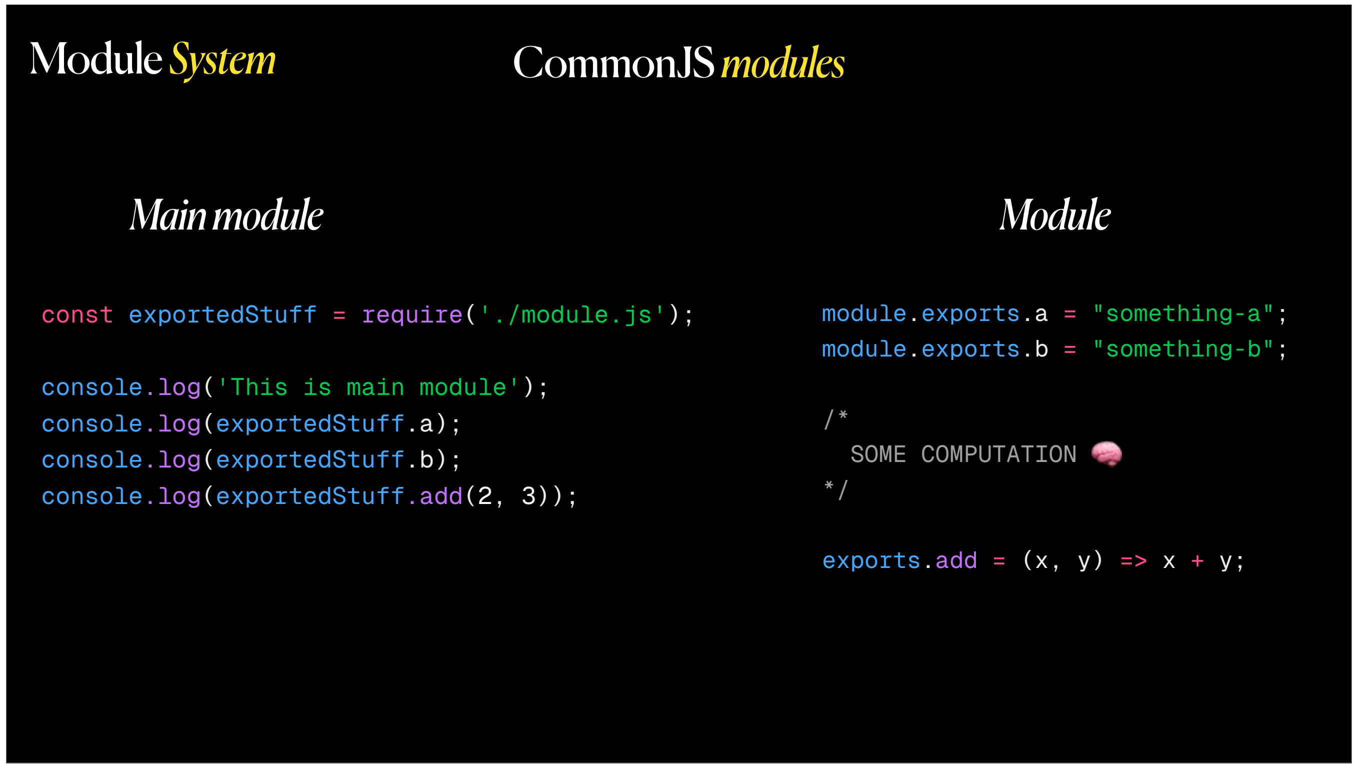


Module *System*

CommonJS *modules*

CommonJS modules are the original way to package JavaScript code for Node.js

Starting with commonJS, it is the original way to write modules for nodeJS.
>> Lets take



Module *System*

CommonJS *modules*

Main module

```
const exportedStuff = require('./module.js');

console.log('This is main module');
console.log(exportedStuff.a);
console.log(exportedStuff.b);
console.log(exportedStuff.add(2, 3));
```

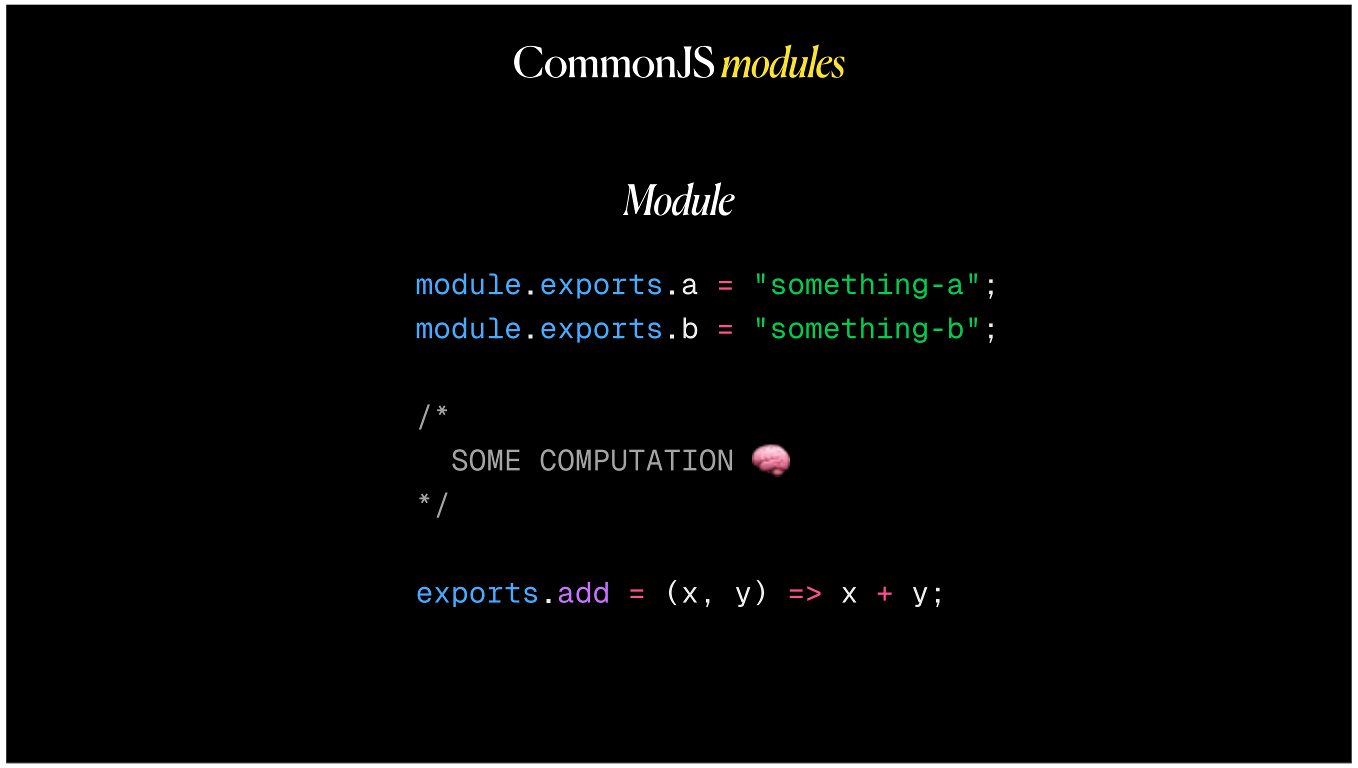
Module

```
module.exports.a = "something-a";
module.exports.b = "something-b";

/*
  SOME COMPUTATION 🧠
*/

exports.add = (x, y) => x + y;
```

Two files:

1. One is the module -> that exports some stuff.
2. Other is our source file (main module) that will be executed.

>> These are the examples in particular we are gonna talk about and dig into how module system works here.

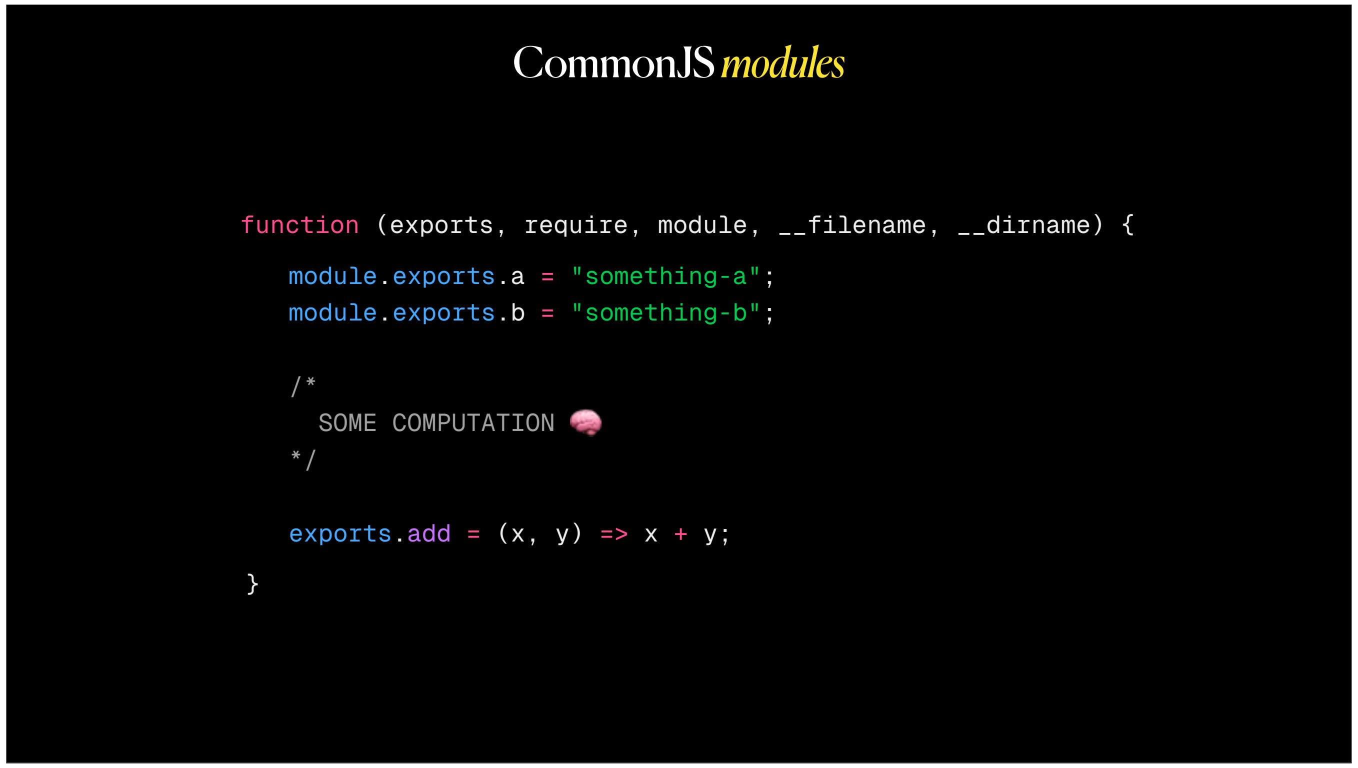


CommonJS *modules*

Module

```
module.exports.a = "something-a";
module.exports.b = "something-b";

/*
  SOME COMPUTATION 🧠
*/

exports.add = (x, y) => x + y;
```

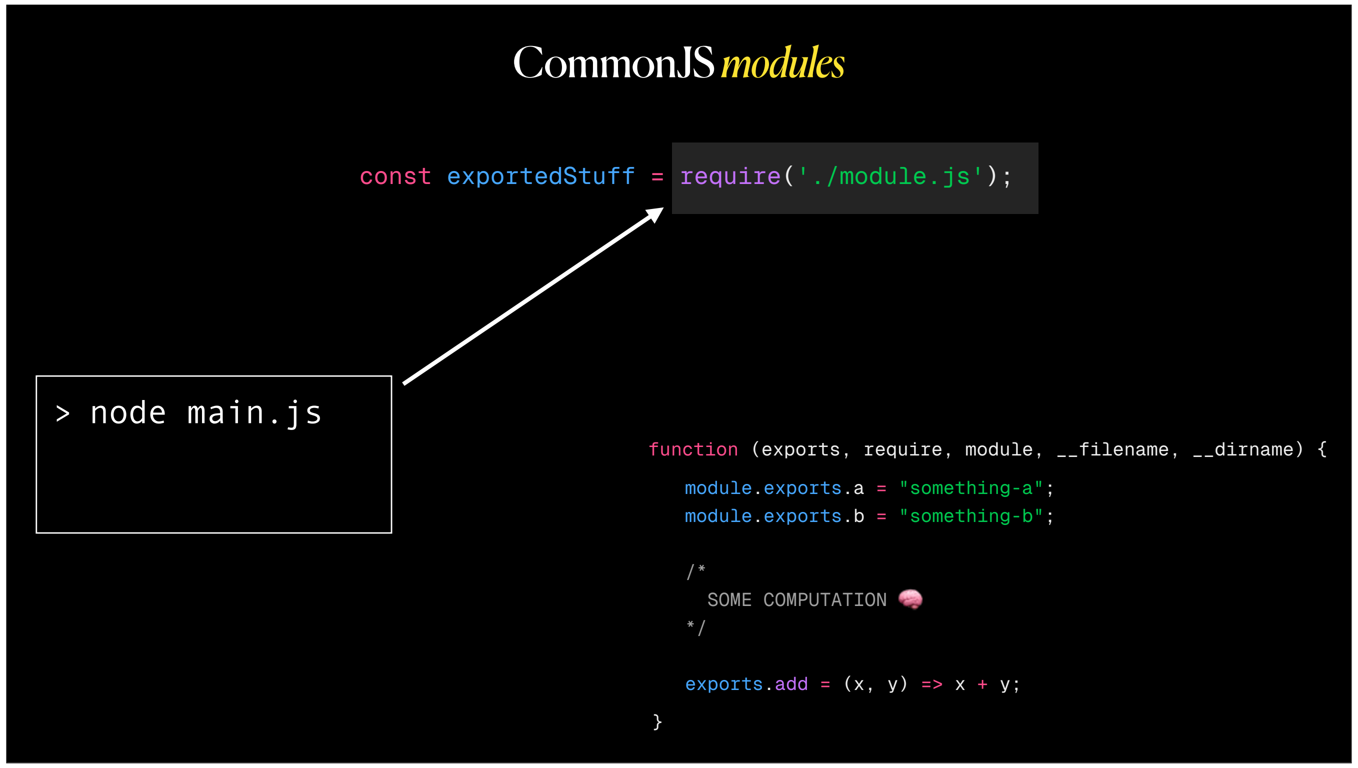
Now I said that both of these are modules; when I say something is a module more specifically commonJS module, it no longer is just the code we wrote in the file.
It now has access to the globalThis object > which using proto inheritance, enables the module to access lots of functionality
among which, the one of our interest right now is that it wraps our code in a function
>> something like this



CommonJS *modules*

```
function (exports, require, module, __filename, __dirname) {

   module.exports.a = "something-a";
   module.exports.b = "something-b";

   /*
     SOME COMPUTATION 🧠
   */

   exports.add = (x, y) => x + y;
}
```

And this is the very reason, we are able to use module, __filename, __dirname inside the function, cz they are literally just parameters. We call this function, that is the wrapped up" version of our code as "callee".
>> Now lets get back



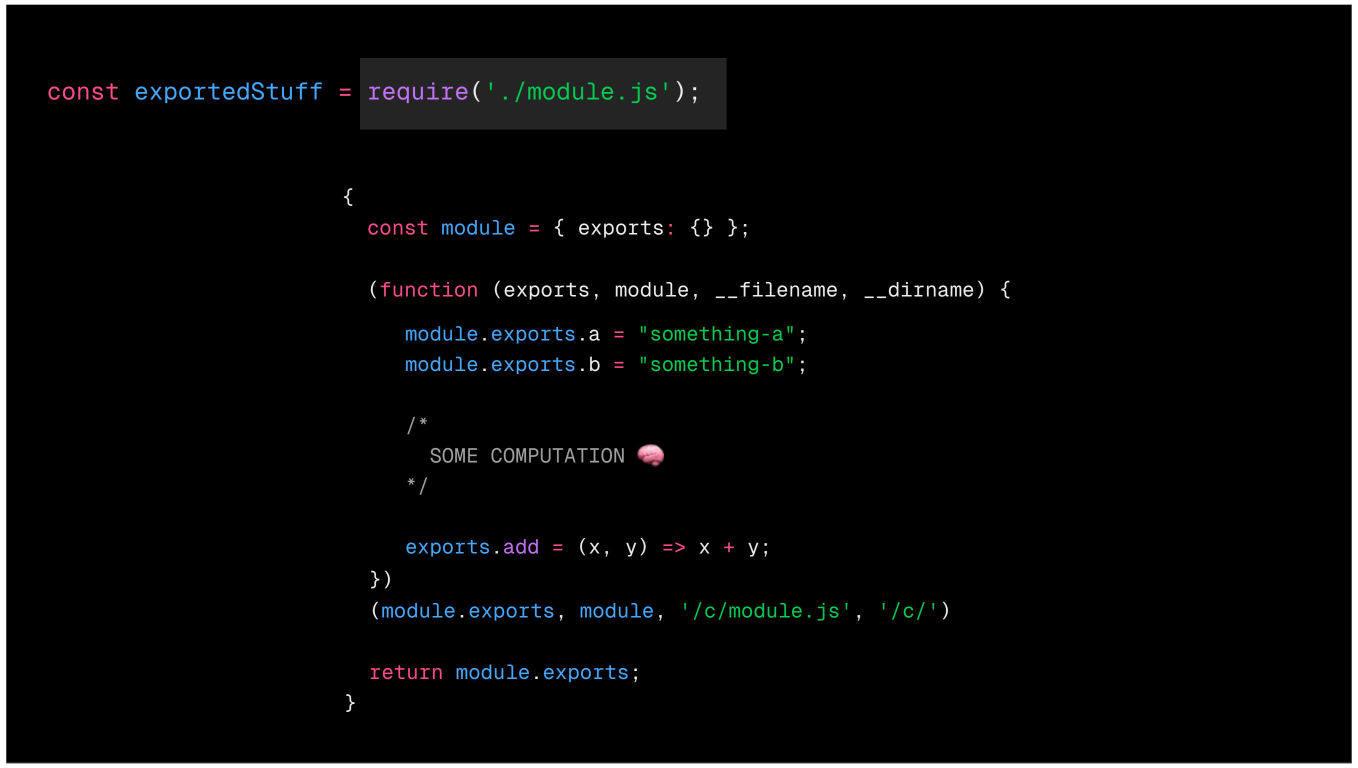
CommonJS *modules*

```
const exportedStuff = require('./module.js');
```

```
> node main.js
```

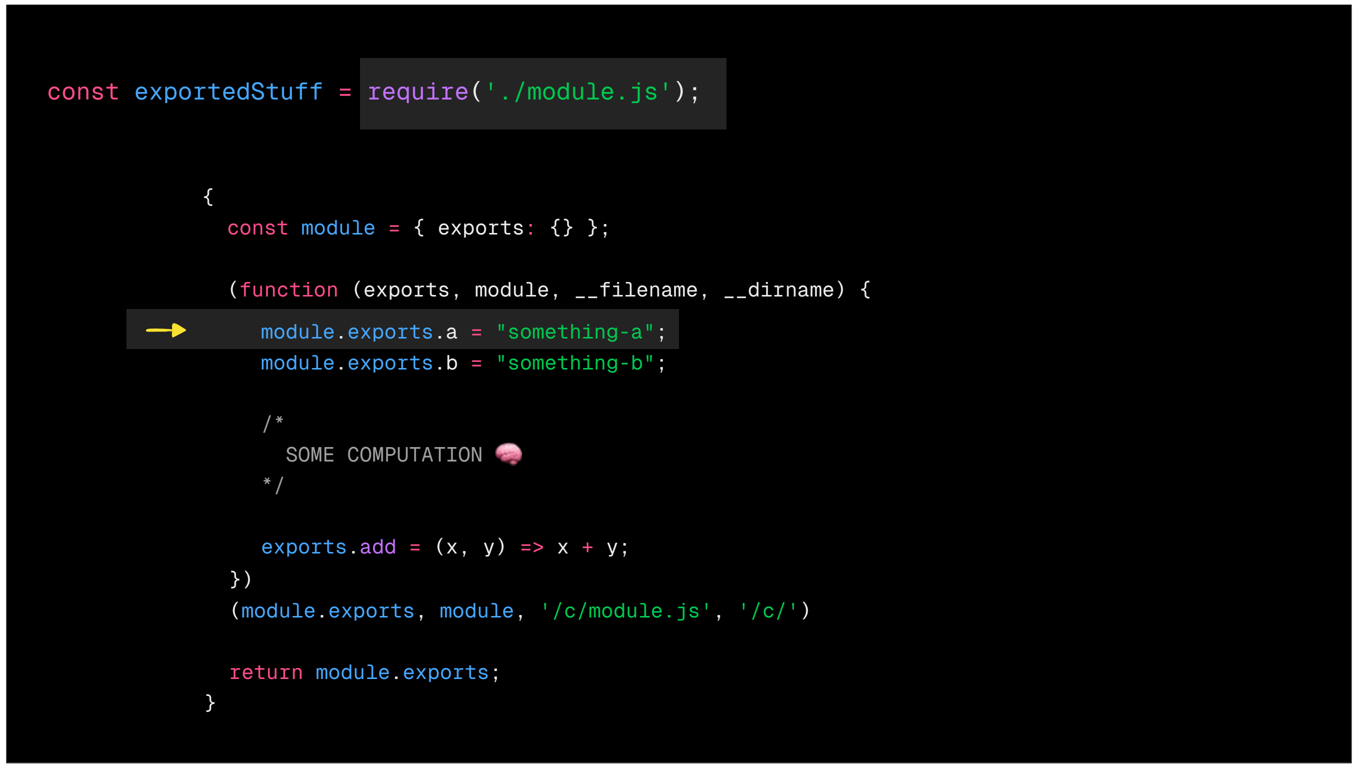
```
function (exports, require, module, __filename, __dirname) {
   module.exports.a = "something-a";
   module.exports.b = "something-b";

   /*
     SOME COMPUTATION 🧠
   */

   exports.add = (x, y) => x + y;
}
```

When we will run our main file using “node main.js”,
the first line will execute, it will call the require function; “Now remb it is a function” so it will follow the same journey like first in “callstack” then “tread of execution”.
>> So what



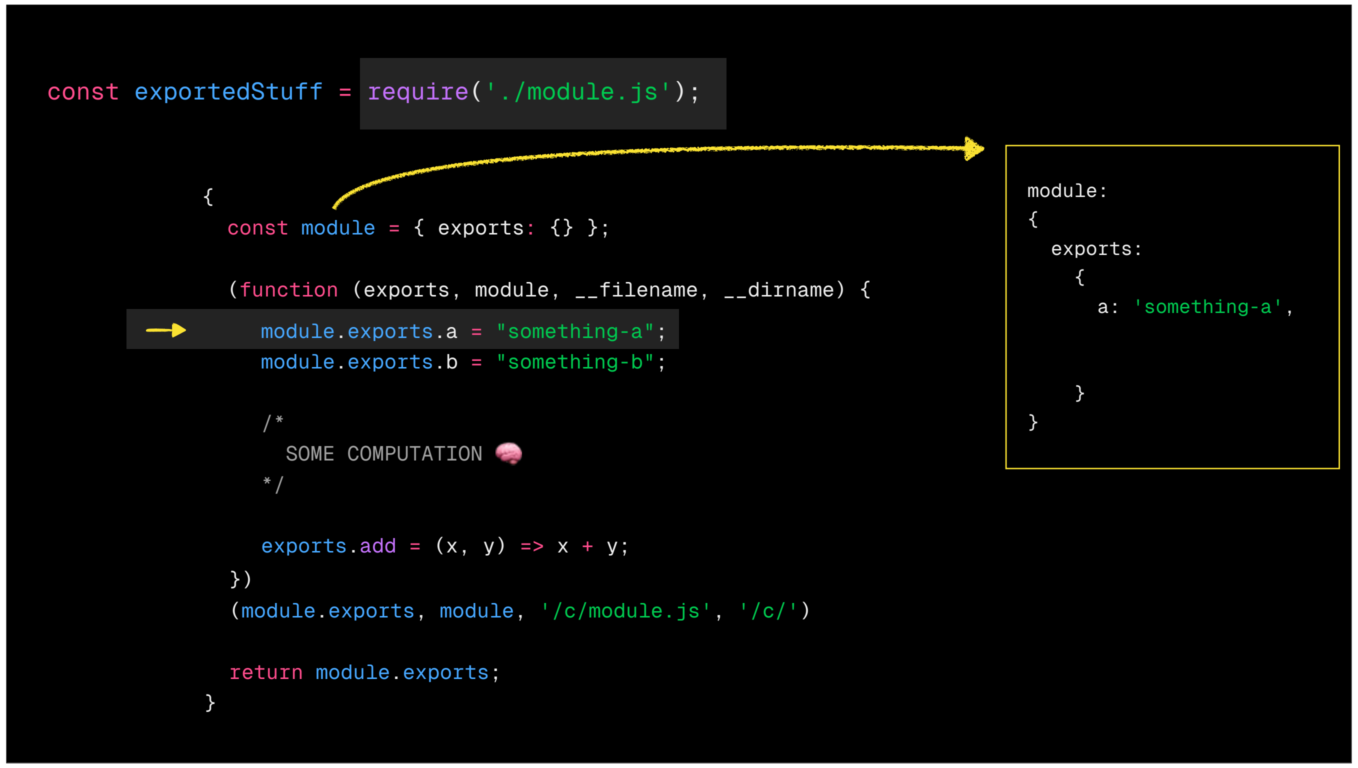
```
const exportedStuff = require('./module.js');


                        {
                          const module = { exports: {} };

                          (function (exports, module, __filename, __dirname) {

                             module.exports.a = "something-a";
                             module.exports.b = "something-b";

                             /*
                               SOME COMPUTATION 🧠
                             */

                             exports.add = (x, y) => x + y;
                          })
                          (module.exports, module, '/c/module.js', '/c/')

                          return module.exports;
                        }
```

this require function does it calls our “callee function” as an IIFExpression with appropriate arguments as you can see and at last you can see that it returns the module.exports
which is reference to the same object, we are adding our exports to.
>> which means



```
const exportedStuff = require('./module.js');

    {
      const module = { exports: {} };

      (function (exports, module, __filename, __dirname) {
 →        module.exports.a = "something-a";
          module.exports.b = "something-b";

          /*
            SOME COMPUTATION 🧠
          */

          exports.add = (x, y) => x + y;
      })
      (module.exports, module, '/c/module.js', '/c/')

      return module.exports;
    }
```

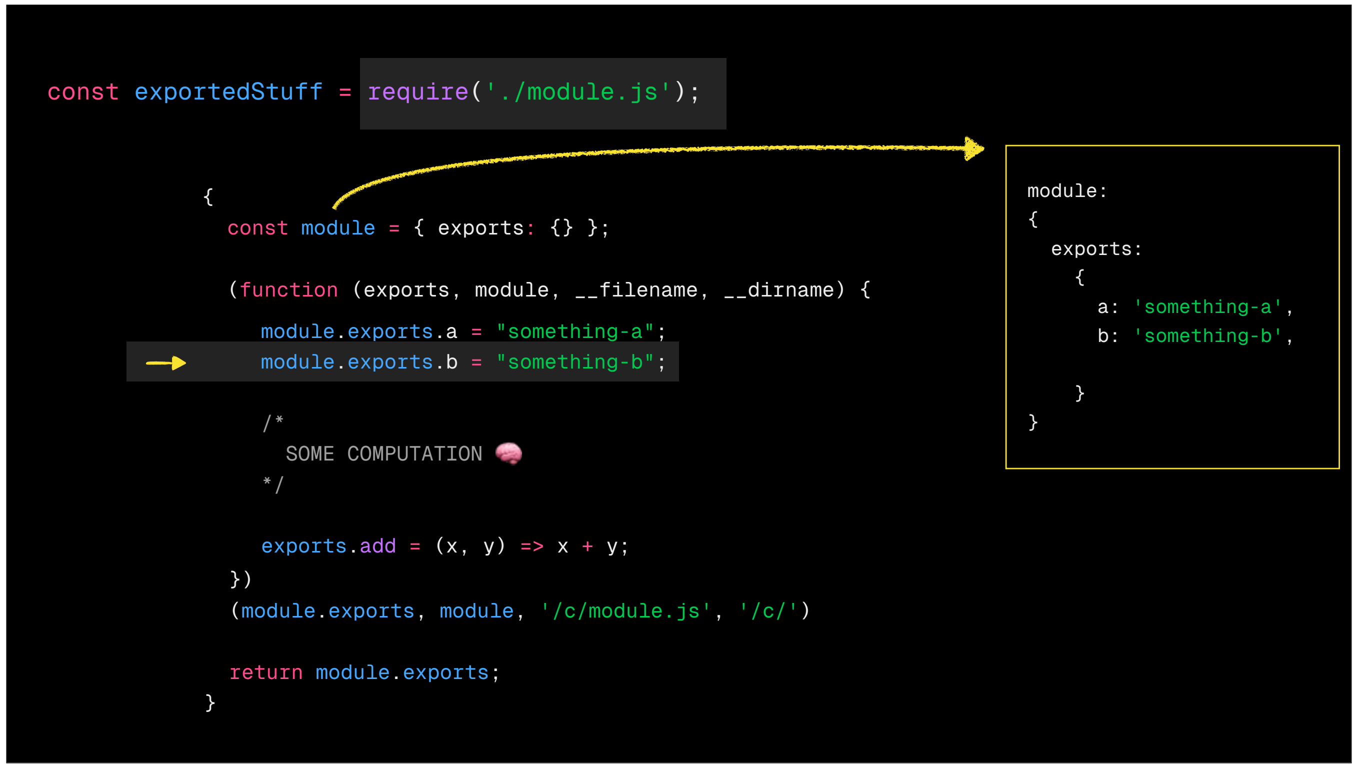
When this line in our code will execute,
>> the



```
const exportedStuff = require('./module.js');

{
  const module = { exports: {} };

  (function (exports, module, __filename, __dirname) {
     module.exports.a = "something-a";
     module.exports.b = "something-b";

     /*
       SOME COMPUTATION 🧠
     */

     exports.add = (x, y) => x + y;
  })
  (module.exports, module, '/c/module.js', '/c/')

  return module.exports;
}
```

```
module:
{
  exports:
    {
      a: 'something-a',


    }
}
```

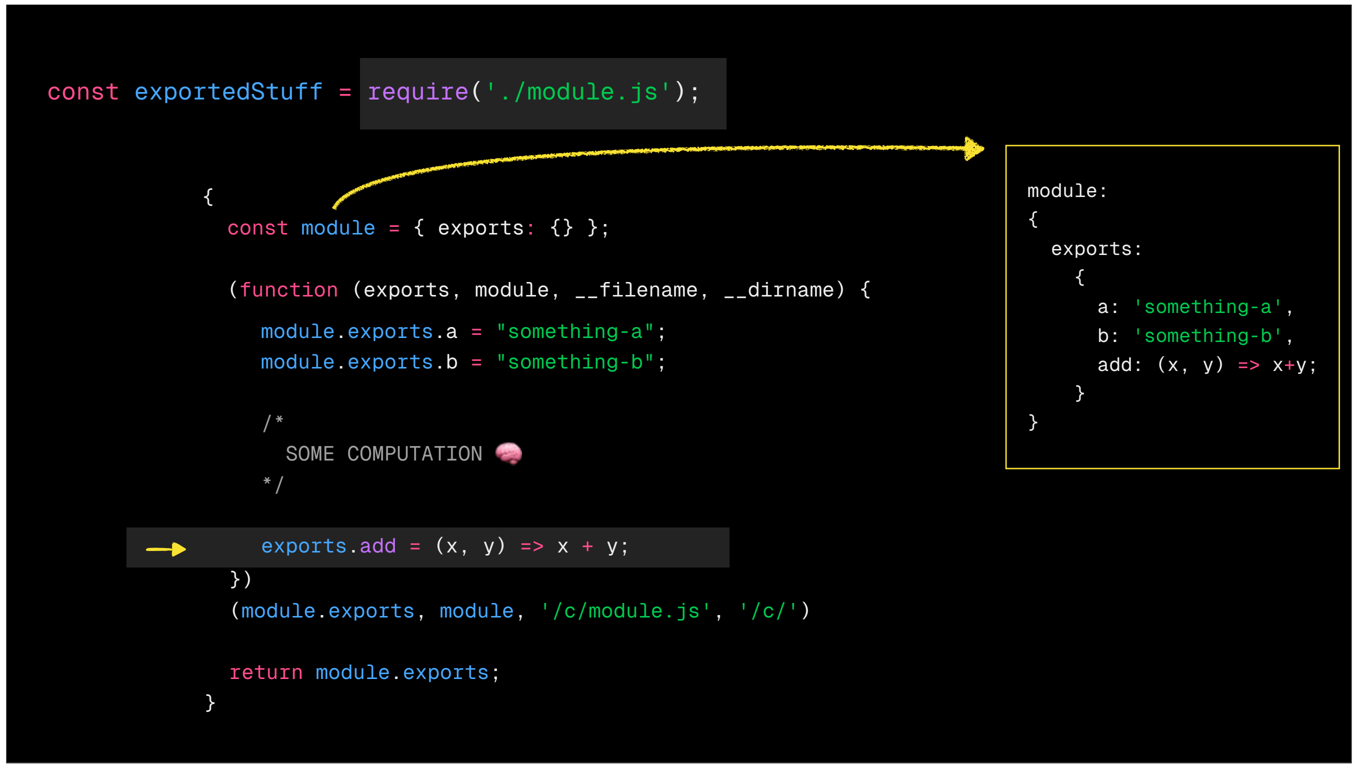
property “a” will get added to our exports object
>> same



```
const exportedStuff = require('./module.js');

    {
      const module = { exports: {} };

      (function (exports, module, __filename, __dirname) {
        module.exports.a = "something-a";
        module.exports.b = "something-b";

        /*
          SOME COMPUTATION 🧠
        */

        exports.add = (x, y) => x + y;
      })
      (module.exports, module, '/c/module.js', '/c/')

      return module.exports;
    }
```

```
module:
{
  exports:
    {
      a: 'something-a',
      b: 'something-b',

    }
}
```

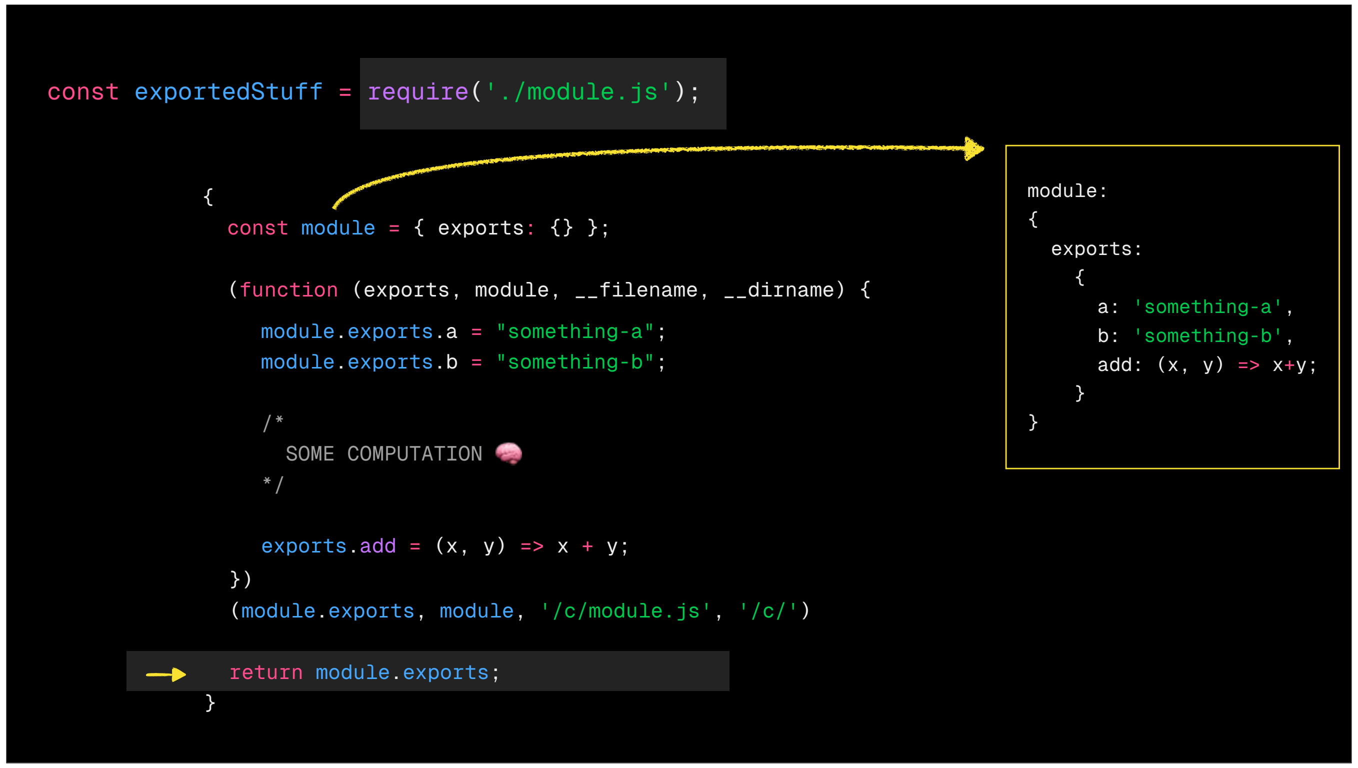
Happens when next line gets executed;
>> and



```
const exportedStuff = require('./module.js');

      {
        const module = { exports: {} };

        (function (exports, module, __filename, __dirname) {
           module.exports.a = "something-a";
           module.exports.b = "something-b";

           /*
             SOME COMPUTATION 🧠
           */

           exports.add = (x, y) => x + y;
        })
        (module.exports, module, '/c/module.js', '/c/')

        return module.exports;
      }
```

```
module:
{
  exports:
    {
      a: 'something-a',
      b: 'something-b',
      add: (x, y) => x+y;
    }
}
```

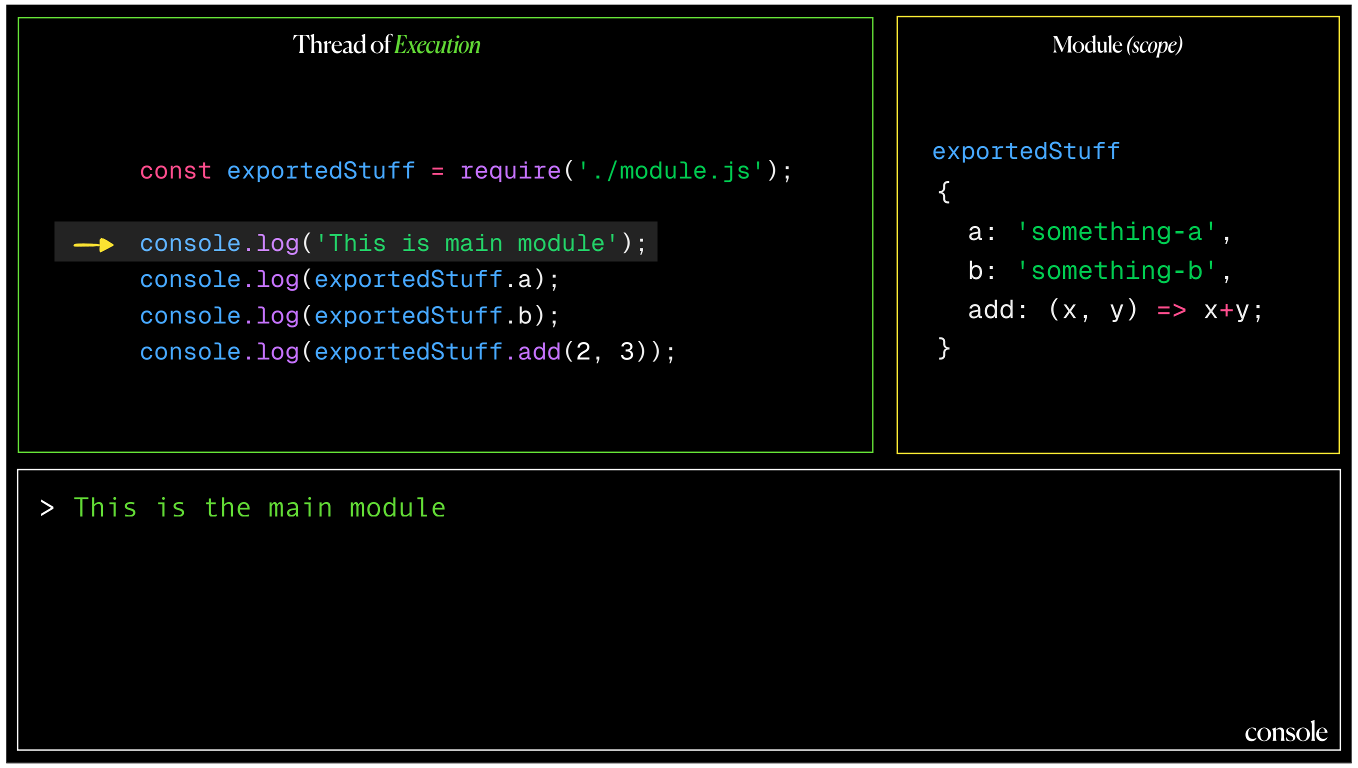
Same goes for the “add” property we are exporting at the end.
This means it can have values other than Primitives too, just like normal object would have.
And notice, we are using exports here not “module.exports”, this is to show that they both point to same object; although there are cases where if we assign exports to a new object, we will loose reference to the module.exports which is the what actually gets returned.
>> and at last, we



```
const exportedStuff = require('./module.js');

        {
          const module = { exports: {} };

          (function (exports, module, __filename, __dirname) {
             module.exports.a = "something-a";
             module.exports.b = "something-b";

             /*
               SOME COMPUTATION 🧠
             */

             exports.add = (x, y) => x + y;
          })
          (module.exports, module, '/c/module.js', '/c/')

          return module.exports;
        }
```

```
module:
{
  exports:
    {
      a: 'something-a',
      b: 'something-b',
      add: (x, y) => x+y;
    }
}
```

reach end of the code,
>> where we return this object

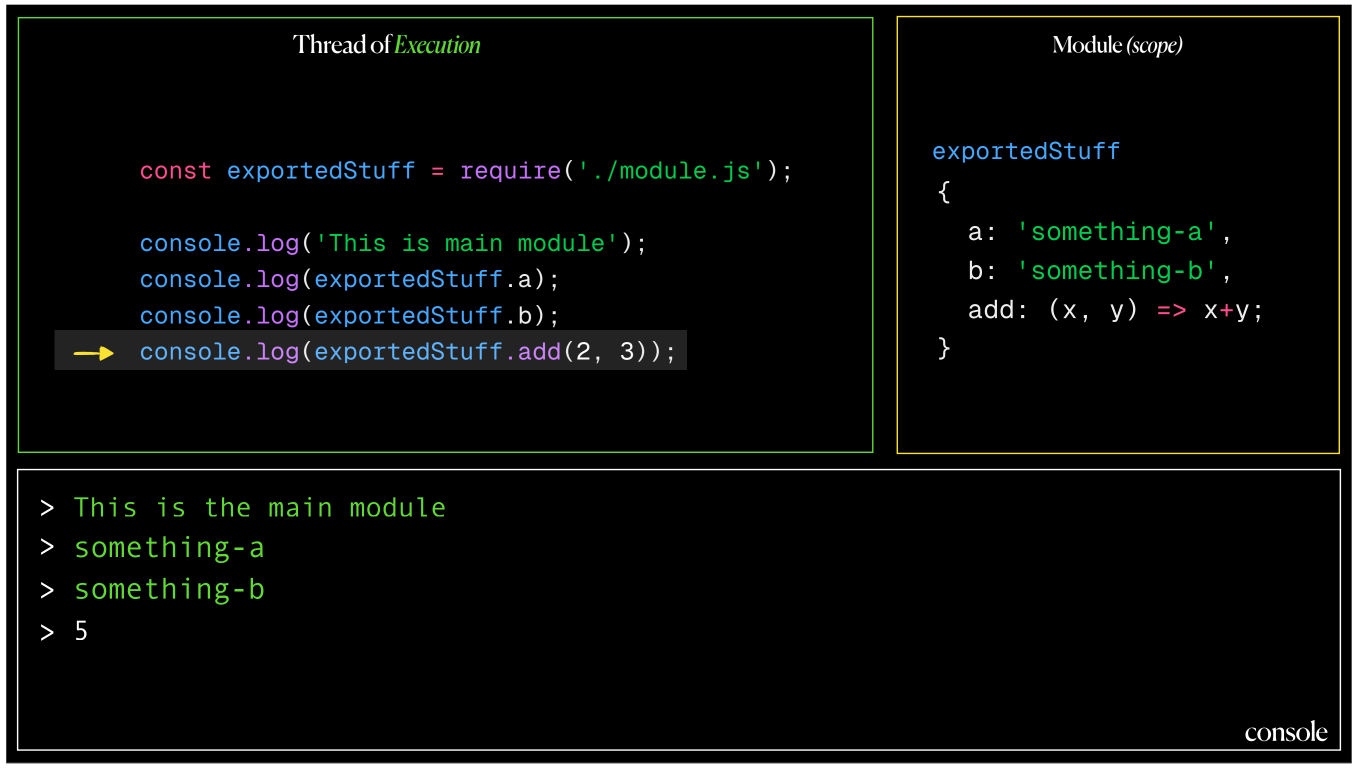


```
const exportedStuff = require('./module.js');

console.log('This is main module');
console.log(exportedStuff.a);
console.log(exportedStuff.b);
console.log(exportedStuff.add(2, 3));
```

```
exportedStuff
{
  a: 'something-a',
  b: 'something-b',
  add: (x, y) => x+y;
}
```

```
> This is the main module
```

to our main module, in which the “exportedStuff” variable will now hold the reference to this object;
>> now forwarding the execution,



```
const exportedStuff = require('./module.js');

console.log('This is main module');
console.log(exportedStuff.a);
console.log(exportedStuff.b);
console.log(exportedStuff.add(2, 3));
```

```
exportedStuff
{
  a: 'something-a',
  b: 'something-b',
  add: (x, y) => x+y;
}
```

```
> This is the main module
> something-a
> something-b
> 5
```

We see in console, the same stuff we exported from our module being logged; and even the result of add function is same as expected.
And that was an idea of how CommonJS module system works.
>> Now moving on to the ESM module system (which I hope you have forgotten is even a thing xD).

[redacted]