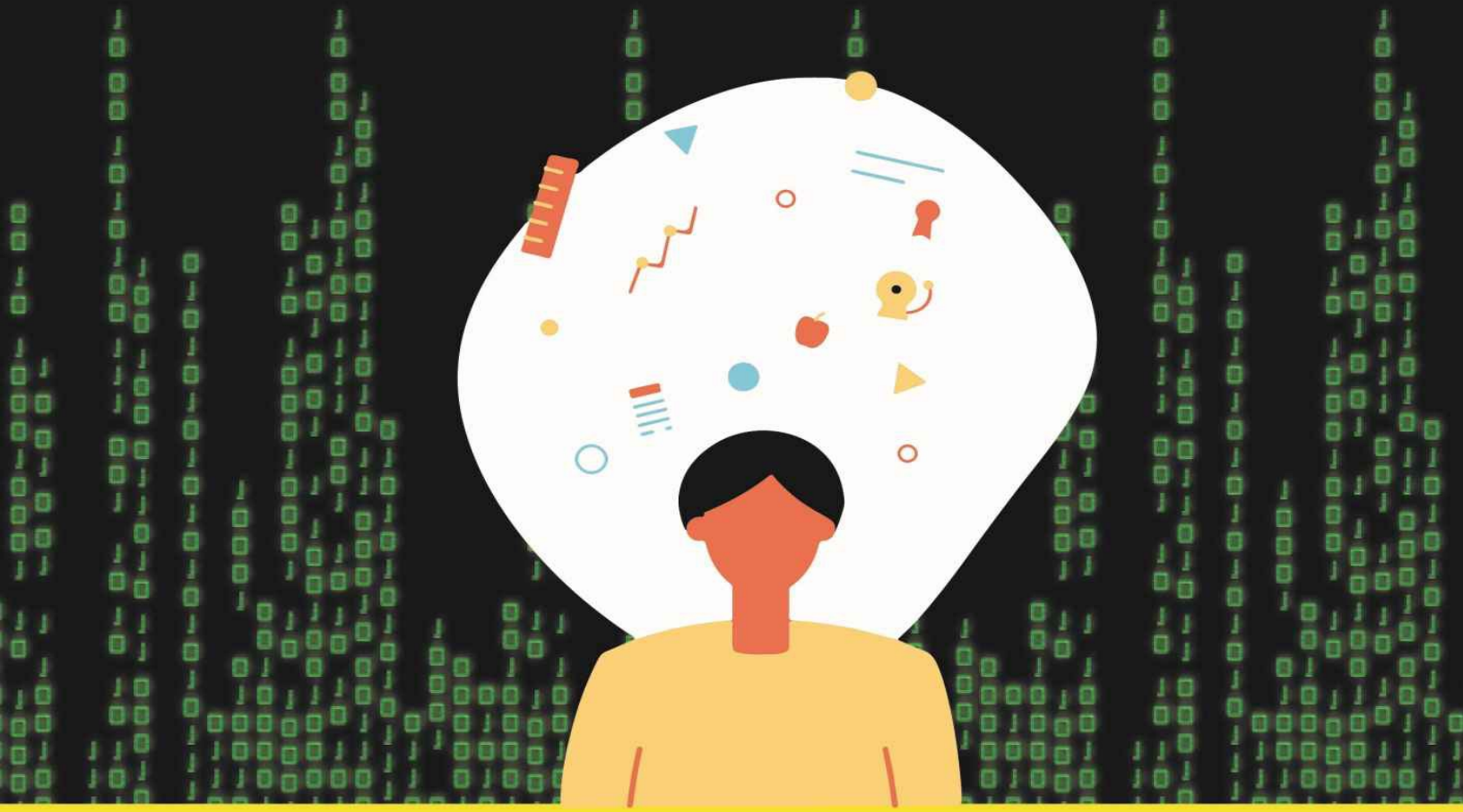
ADVANCED JAVASCRIPT VISUALIZED



// MADE BY MEET PATEL WITH ❤

Meet Patel

Advanced JavaScript Visualized

First edition

This book was professionally typeset on Reedsy
Find out more at reedsy.com

I like to dedicate this book to my engineering which made me realize I should have started writing earlier :)

The true sign of intelligence is not knowledge but imagination.

- Albert Einstein

Contents

1

Execution Context, Thread & Callstack

```
We all know that we always had and have a love and hate relationship with
JavaScript from the day you or I started learning it. But, sometimes philosophy
around those weird JavaScript concepts can easily be understood if we just dig down
the basics and fundamentals.
```

This is the most important and fundamental chapter of all. So if you get this, then almost everything will be easy to follow and understand. Welcome to the exciting journey of **JavaScript beyond normal expectations.**

Terminologies

- Memory

- Processes and Threads
- Execution Context
- Scope
- The Thread of Execution
- Callstack

```
Memory:- A computer is just a machine that takes a set of instructions and executes it, but during that time computer needs to hold some information for the calculation or for any other task. So, we have storage devices like RAM and ROM (HDDs, SSDs) which hold information temporarily or permanently. So, memory is just a part where our information is held.
```

```
Process:- A currently executing/running program (bundled up instructions) is called process. Each process is just a blueprint for what the computer needs to do in order to finish a given task.

Thread:- A process is a set of instructions bundled up, but each instruction might need to be executed as well to achieve the original process to be finished and so the thread is just a simple lightweight running process (a small set of instruction/s in execution).
```

```
Execution Context:- It's nothing but a currently running code inside memory(every running program needs memory) which could be normal function execution or normal JavaScript code like if-else, loop, or any other expression.

Memory + Process == Execution Context
```

```
Scope:- A Scope is an allocated memory to a specific or currently executing function or block of code. That is to say, variables(memory) is not available outside of it to functions or block of code.

Scope == Memory Restriction/s for execution context/s
```

```
The Thread of Execution:- A currently running(executing) function or block of code is a thread of execution. That is to say, it is a small running process inside our main process.
```

```
Callstack:- It's a container in which you can add or remove code block/functions such that most recently added stuff in this container will get priority for the next thread of execution. That is to say Last In First Out (LIFO) pattern.
```

Before we move further let me give you a quick overview of how processes are managed at a high level.

1. Initially, any program(set of instructions) is collected by the memory(usually in the RAM) and then the processor takes the initiative to complete that program.
2. A processor then take the program and start executing it and during that process processor might need to do extra stuff to accomplish a given task i.e. processor might perform a program inside the program(Set of instruction might need other instructions as well) AKA thread/s also needs to be done as well.
3. After finishing the program, the Operating System will take care of the cleanup and some other dependent task/s.
4. Now, any program is called **single-threaded** when the given program can perform a **set of instructions one by one, and no extra internal set of instructions is done(Running JavaScript program is a single-threaded process).**

* * *

After getting an idea about termonologies Let’s now further understand stuff with a little example

```
1   let outerGoal = 0;
2
3   console.log(outerGoal);
4
5   function addGoal(count) {
6       let innerGoal = 0;
7       innerGoal = innerGoal + count;
8       return innerGoal;
9   }
```

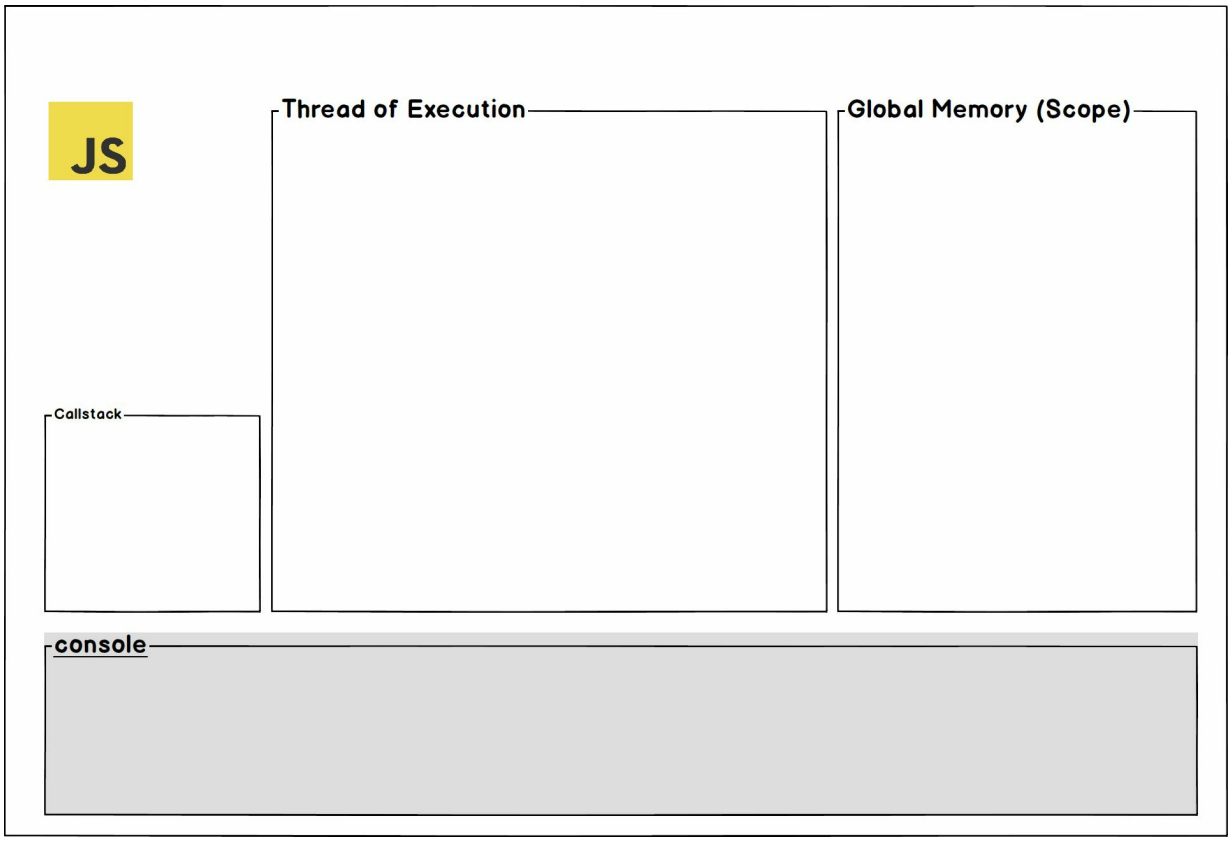
Example 1.1

- First thing if you don’t know then, JavaScript is a single-threaded and synchronous language which mean JavaScript engine/Interpreter can execute one line or block of code at one time. Which means you can’t do multiple things at the same time. Also, you go through line by line while executing code (synchronously).

- when you execute JavaScript code on a browser or any other enviroment(nodejs, react-native etc) we run our code in something called global execution environment(global memory space for browser tab or global memory for node enviroment first).

- every function we defined are actually first goes inside global memory space (ignore function defined inside another functions for now). When we put these function into execution, first they are put on callstack first and from that callstack functions are getting called where latest function are made available in thread of execution!

- Before we start understanding the given code it's important to create "mental model" for all the stuff we learned so far.



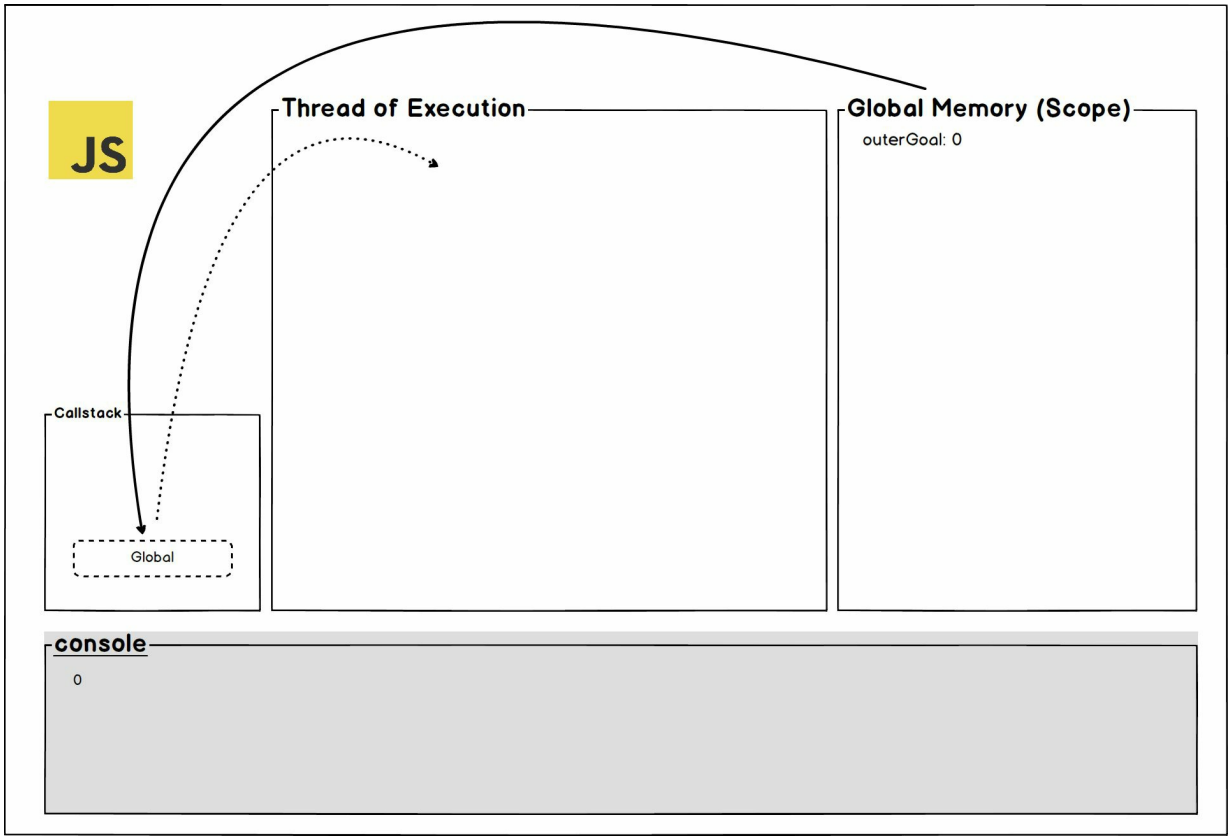
Current JavaScript Mental Model

- In our mental model, we can see that every JavaScript program gets Global memory space (Global Scope). Every function and topmost declared variables are hoisted in the global scope.
- **Hoisting** is **JavaScript's** default behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function) - W3SCHOOL (I don't prefer w3school but sometimes it explains stuff easily😊).
- Each line of code or function definition is not called directly in JavaScript. But instead, it goes to callstack, and then based on callstack current priority function/s is put in the Thread of execution (Last In First Out). Why not understand this mental model via example.
- Let's start reading the below code and explore and keep the above mental model in mind.

```
1   let outerGoal = 0;
2
3   console.log(outerGoal);
4
5   function addGoal(count) {
6       let innerGoal = 0;
7       innerGoal = innerGoal + count;
8       return innerGoal;
9   }
10
11  const storedEvaluatedValue = addGoal(1);
12
13  console.log(storedEvaluatedValue);
```

Example 1.1

- In the first line as you can see we are declaring a variable called **“outerGoal”** and assigning a value of 0. That is to say, we are consuming small memory space inside global memory (scope) and giving the label called **“outerGoal”** to that memory.
- In the next line of code, we are simply logging that value to the console. That is to say, we are simply logging value at a label called **“outerGoal”**.
- One thing to note, I will not draw a global thread of execution diagram but it’s always when we start executing our javascript code.
- Let’s see our **“mental model”** again how it applies to the current situation.



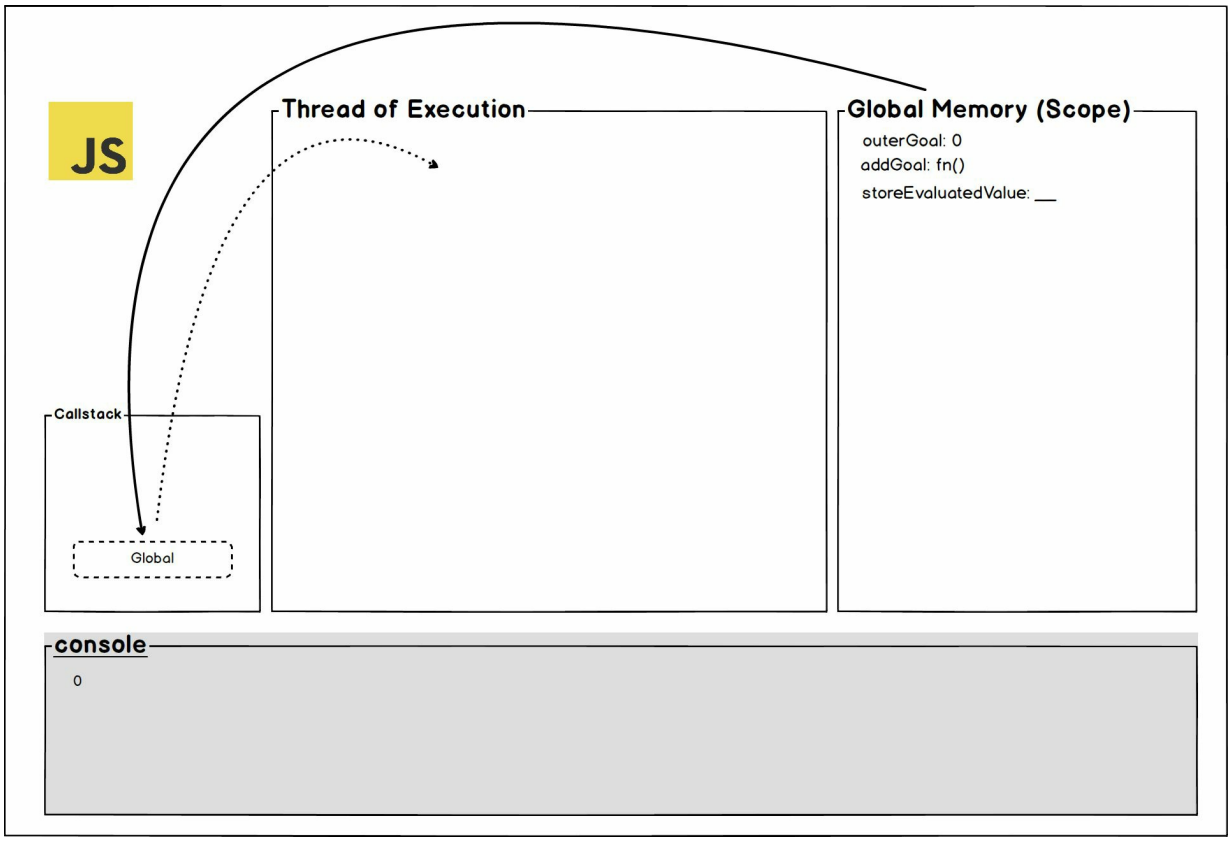
Current JavaScript Mental Model

let’s further understand the below code

```
1   let outerGoal = 0;
2
3   console.log(outerGoal);
4
5   function addGoal(count) {
6       let innerGoal = 0;
7       innerGoal = innerGoal + count;
8       return innerGoal;
9   }
10
11  const storedEvaluatedValue = addGoal(1);
12
13  console.log(storedEvaluatedValue);
```

Example 1.1

- As we can see, we are declaring function(a special type of object in js) and giving the label **“addGoal”**.
- Note that this function declaration has 1 parameter called **count** which means **count** is a label for future memory space address which will be going to consume by this function later.
- Inside function definition whatever we are doing is noting that much important right now (will see detail when we call it).
- After function definition, we are actually declaring variable “storedEvaluatedValue” in global scope which will be return or evaluated value from addGoal function call.
- In case if you didn’t notice then, we are not leaving global scope or global thread of execution here.
- It’s time to see our **“mental model”** again.



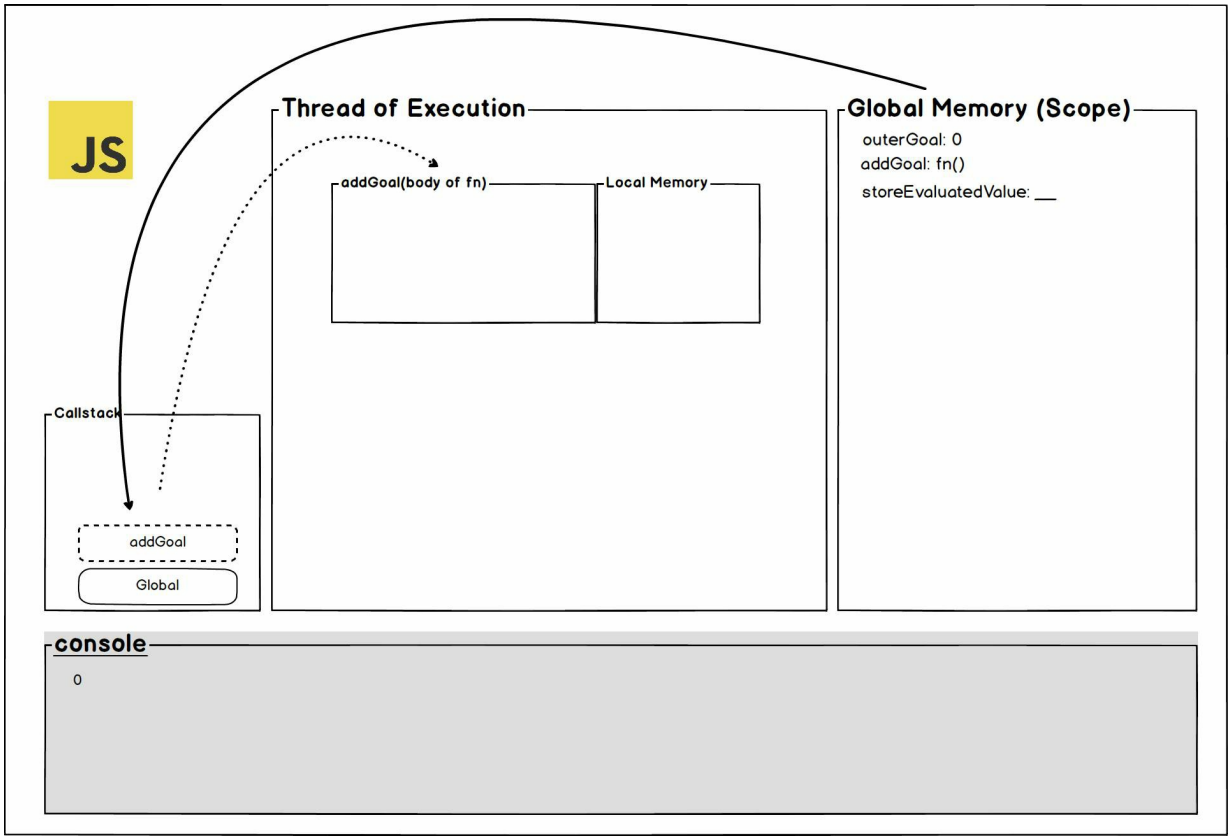
Current JavaScript Mental Model

let’s continue our understanding of code

- Here is the important part comes, we are calling **“addGoal()”** by putting braces around it. As soon as we did that we create a brand new execution context (or local memory space for that function).
- Just to know that making the execution context doesn’t happen magically in JavaScript. Actually, when we call a function instead of putting that function directly into Execution Thread, we first put that function into call stack first.
- Remember, **“stack”** data structure means Last In First Out, which means if we put a function call on the call stack, as long as we don’t

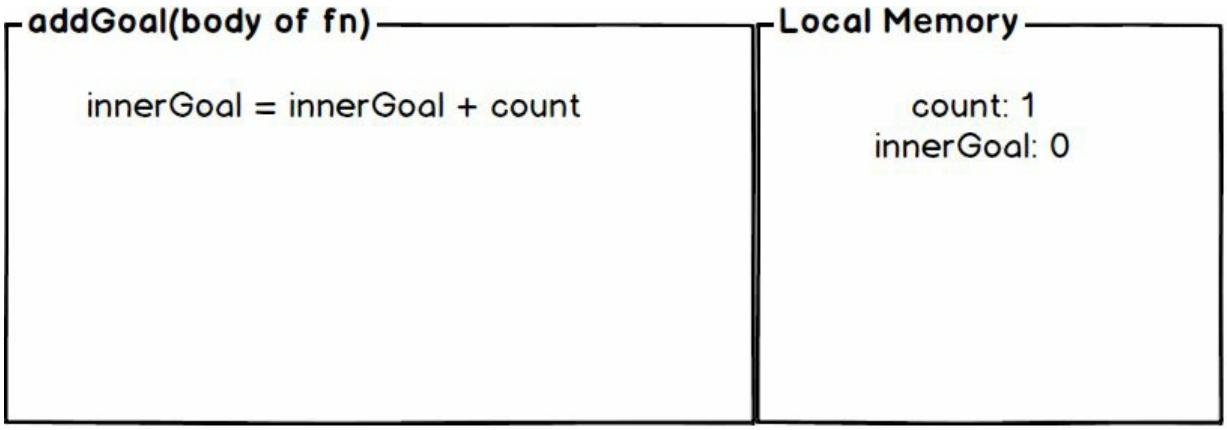
have extra function calls immediately afterward, we take the topmost function of the call stack & put it into the execution thread.

It's time for our **"mental model"** again.



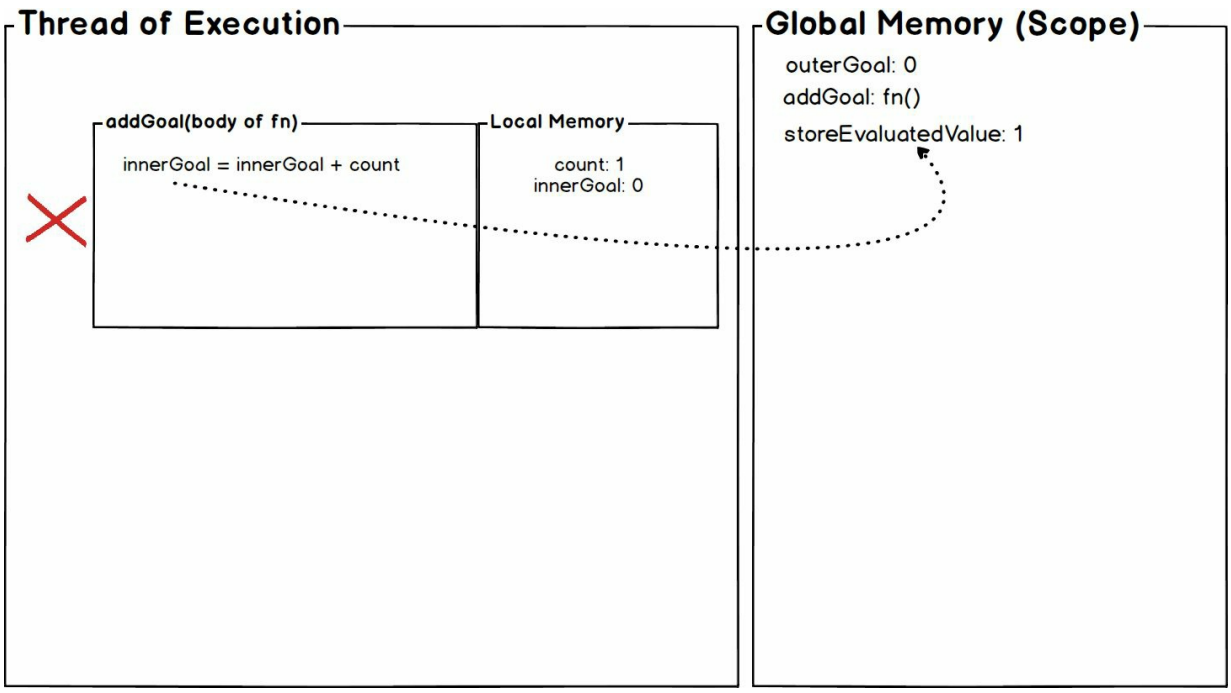
Current JavaScript Mental Model

Before we move further, we need to understand the local scope as well and what is happening inside the local scope as well.

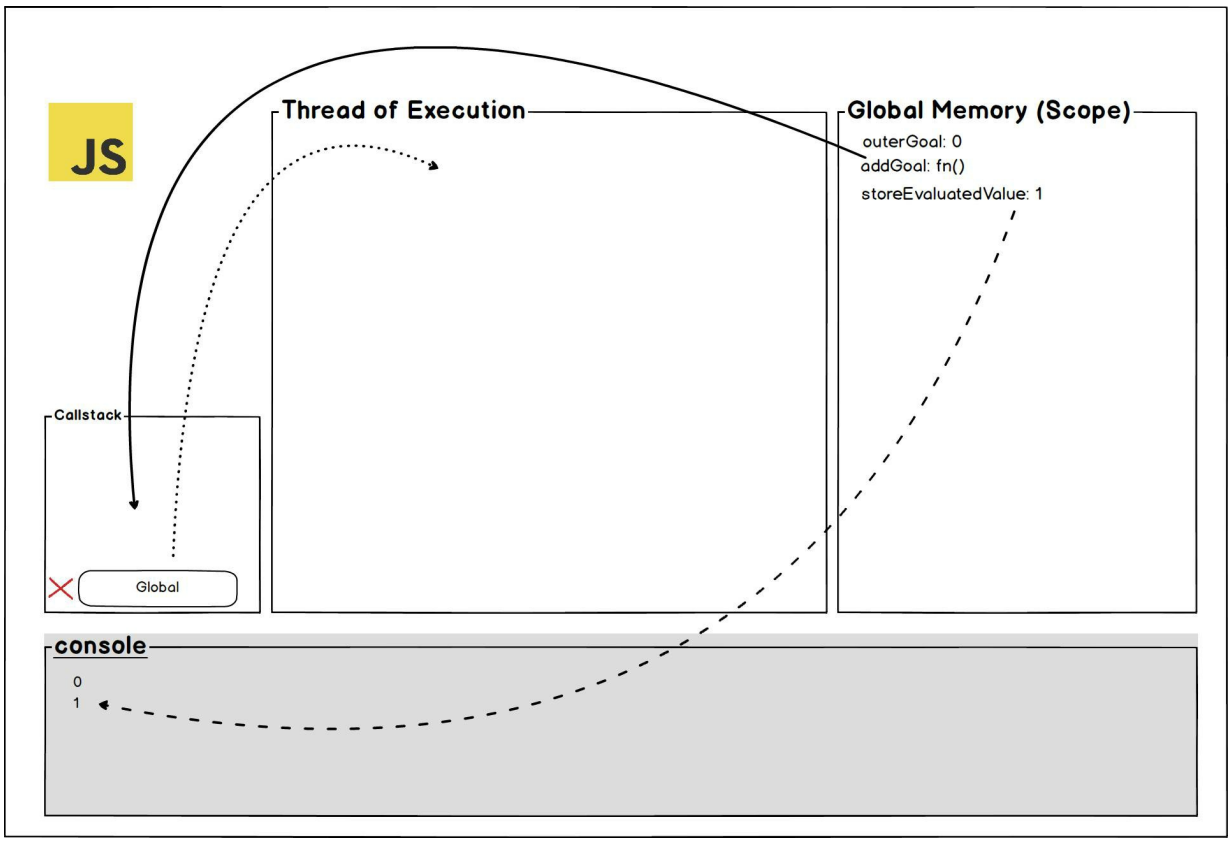


Mental Model of local function and its scope

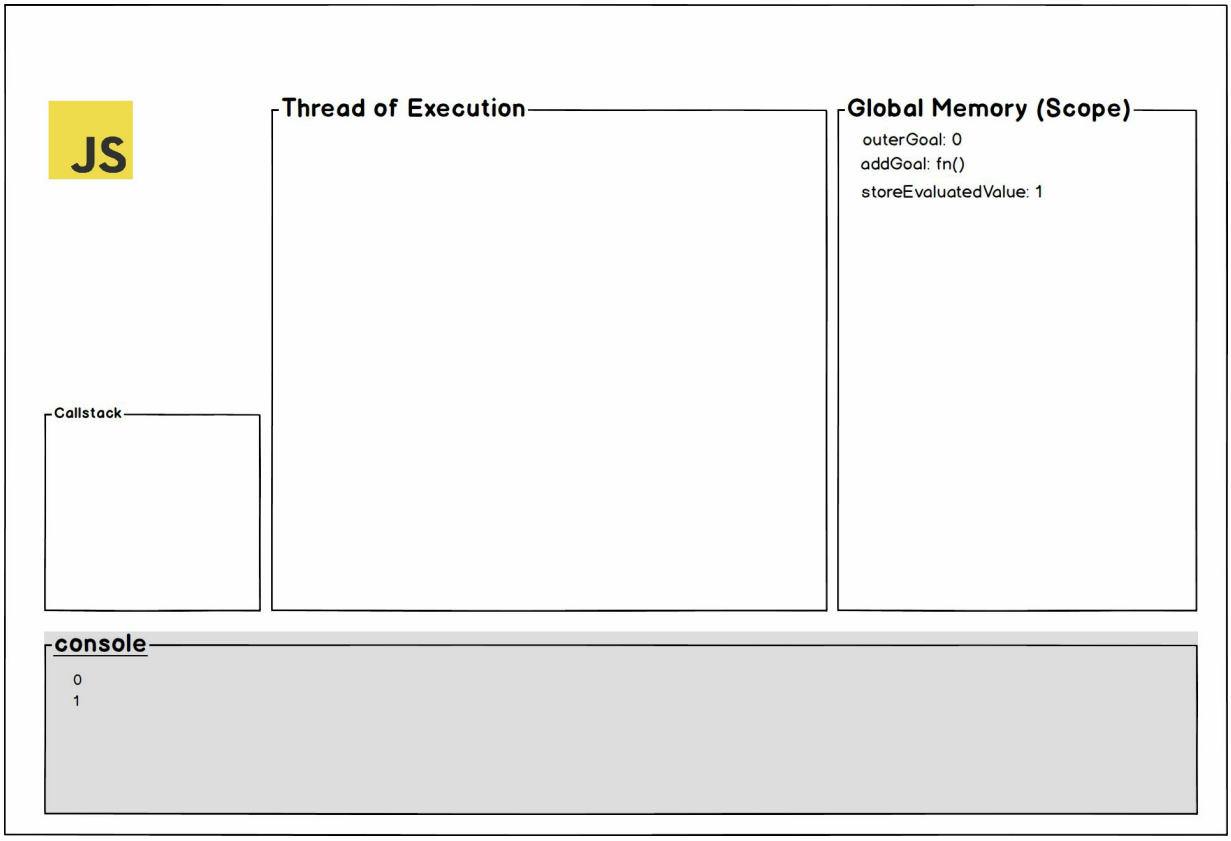
- As you can see in function execution in execution thread, does create local memory & we are manipulating local variable called “innerGoal” inside that call.
- But one thing to note, parameter “count” is also part of local memory, which means the parameter becomes argument only when we run function & it occupies space in memory only when it is used. “Placeholder” or so-called “parameters” do not occupy space.
- After the function body about to finish execution, we are returning the **“innerGoal”** variable. That is to say, the value of it has the label **“innerGoal”.**
- When we exit out of function execution, the JavaScript engine destroys the local function execution context. if a function returns a value then that value will become an evaluated value for that function otherwise it returns undefined by default.



Mental Model of local scope execution



Mental Model of the current execution



Final Mental Model after execution

Quick Notes:- How variables are stored

- Always remember computers are made up of a lot of space (memory addresses).
- when we store/use a space, we are effectively using those memory addresses of the computer.
- Because humans are not good at remembering those memory addresses, we give a label (variable name) that points to that memory address/es.

TLDR;

- The mental model for any programming language is really important and it plays an exceptional role in the understanding of core concepts.
- Keep this mental model for a further chapter in mind and you will enjoy learning.

* * *

2

Closure - A Deep Dive

```
When it comes to the most widely used programming languages like JavaScript &
Python, we can't ignore its most used feature called "Closure" directly or
indirectly. So, Let's understand how it works under the hood. Also, Before we start
let's understand what really JavaScript functions are ?!
```

JavaScript Functions

- A JavaScript function is an **“Object”!**
- An Object in JavaScript means it can have key-value pairs, nothing fancy more than that. (We will understand Objects in much detail in “Prototype Chapter”).
- Functions are a special type of Object, and one of the special features is to store blocks of code and invoke(run) that later when needed!
- Because the function is an object we can store key-value pairs to the function as well, it seems a little strange if you are coming from other programming languages. But it’s true!

JavaScript Functions Are First Class

- you can **pass a function as an argument** to function for different kinds of purposes.
- we have a function that can be returned from another function or we can store an anonymous function in a variable as well (Store in memory)
- each function has its own **local memory/local variable scope** (explained in chapter 1).

*Having a feature of adding any key-value pair to function is a superpower and that superpower helps JavaScript enable an amazing feature called **closure.** Let's see how!*

There are definitions for closure available on the internet but none of them actually helped me for understanding how the hell it actually works under the hood. after some research and experiment, I finally understood what actually it really means and how it works under the hood. let's see how with example.

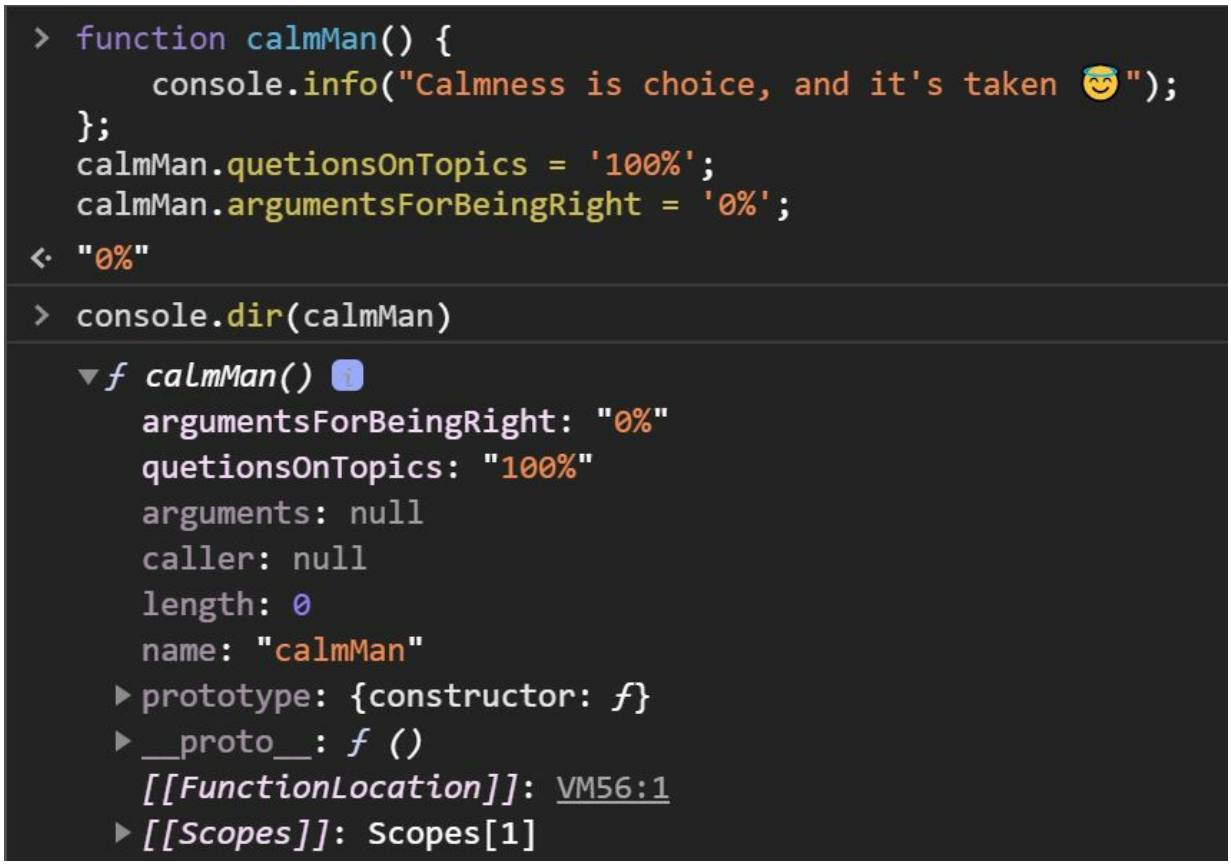
```
14
15  function calmMan() {
16      console.info("Calmness is choice, and it's taken 😇");
17  };
18
19  calmMan.argumentsForBeingRight = '0%';
20  calmMan.quetionsOnTopics = '100%';
21
```

functions are objects

- As you can see from the above example we created a function with the name of **calmMan** and later using object literal syntax we added 2

properties to the function itself(Yes we can do that !!).

- These properties become part of our function object and remain attached to it always. Let's see how they look like under the hoods!



function's internal look

- As we can see from the image both properties belong to the function object and part of it. We can see some other properties (Key-Value pairs) were added by JavaScript itself as well for its internal use(we don't need to worry about that, let JavaScript Engine handle it😒).
- Now, we have proof of work that functions are objects and can hold properties in themselves. Now, what if these functions use this key-value pair internally and make available those key-value pairs to us? and

here are my friends the magic starts.

- By the way, if you have noted in the above image we already have 2 of those internal properties (key-value pairs) made available by JavaScript(i.e. ***[[Scopes]]***). We will look deep into that but let’s take a formal definition of closure now and understand it.

MDN

```
A closure is a combination of a function bundled together (enclosed) with
references to its surrounding state (the lexical environment). In other words, a
closure gives you access to an outer function's scope from an inner function. In
JavaScript, closures are created every time a function is created, at function
creation time.
```

It’s just not easy to grasp and the first time and I didn’t like the mdn definition and explanation also here (It’s the right definition though, it’s just not easy to grasp with buzzwords).

W3SCHOOL

```
JavaScript variables can belong to the local or global scope.
Global variables can be made local (private) with closures.
(try to remember our mental model, we will draw it out soon!)
```

I don’t prefer w3school as a reliable source😊, but this time I felt w3school is actually more precise, easy to grasp, and has a clear definition.

Now it’s our turn to understand from visualization and for that, I am going to take a simple example.

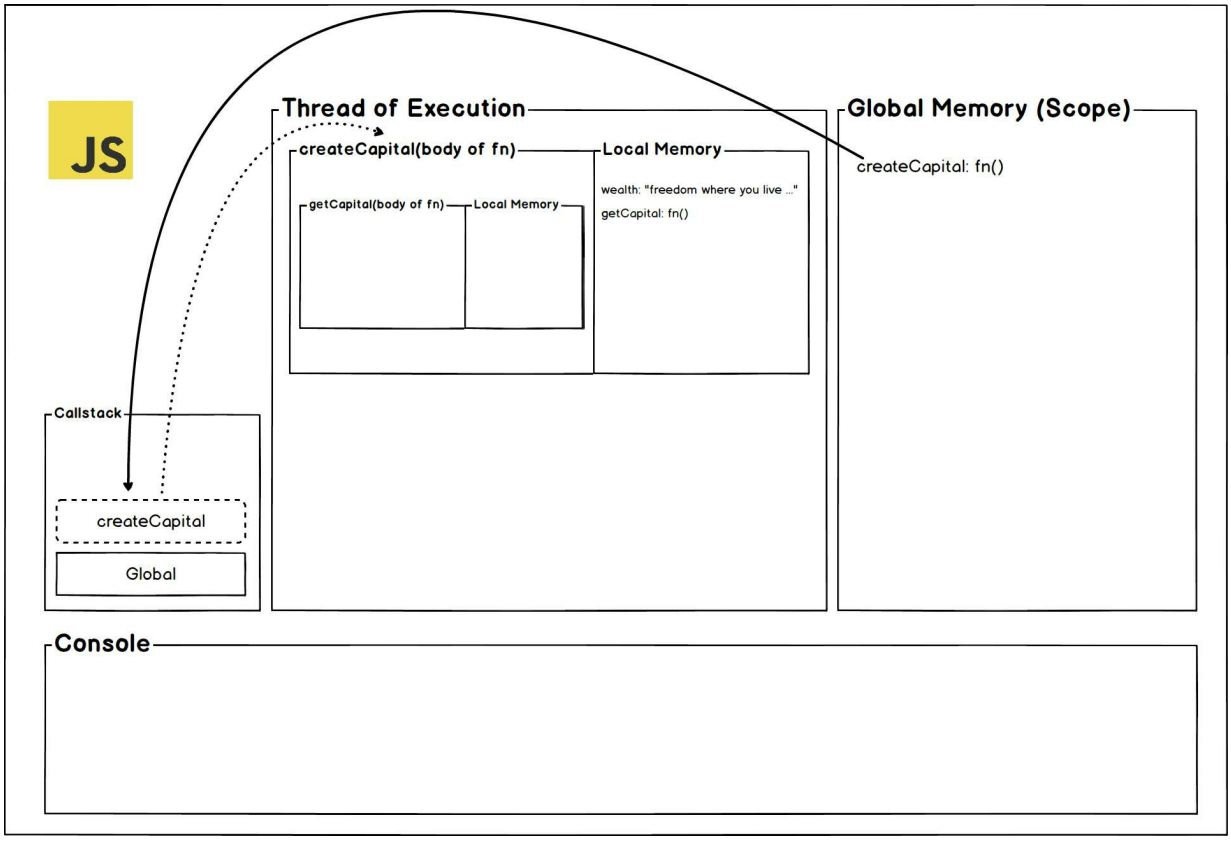
```
21
22  function createCapital() {
23      let wealth = 'freedom where you live your time';
24
25      function getCapital() {
26          console.log(wealth);
27          return wealth;
28      }
29
30      return getCapital;
31  };
32
```

Function definition

```
33
34    const accessCapital = createCapital();
35
36    accessCapital();
37
```

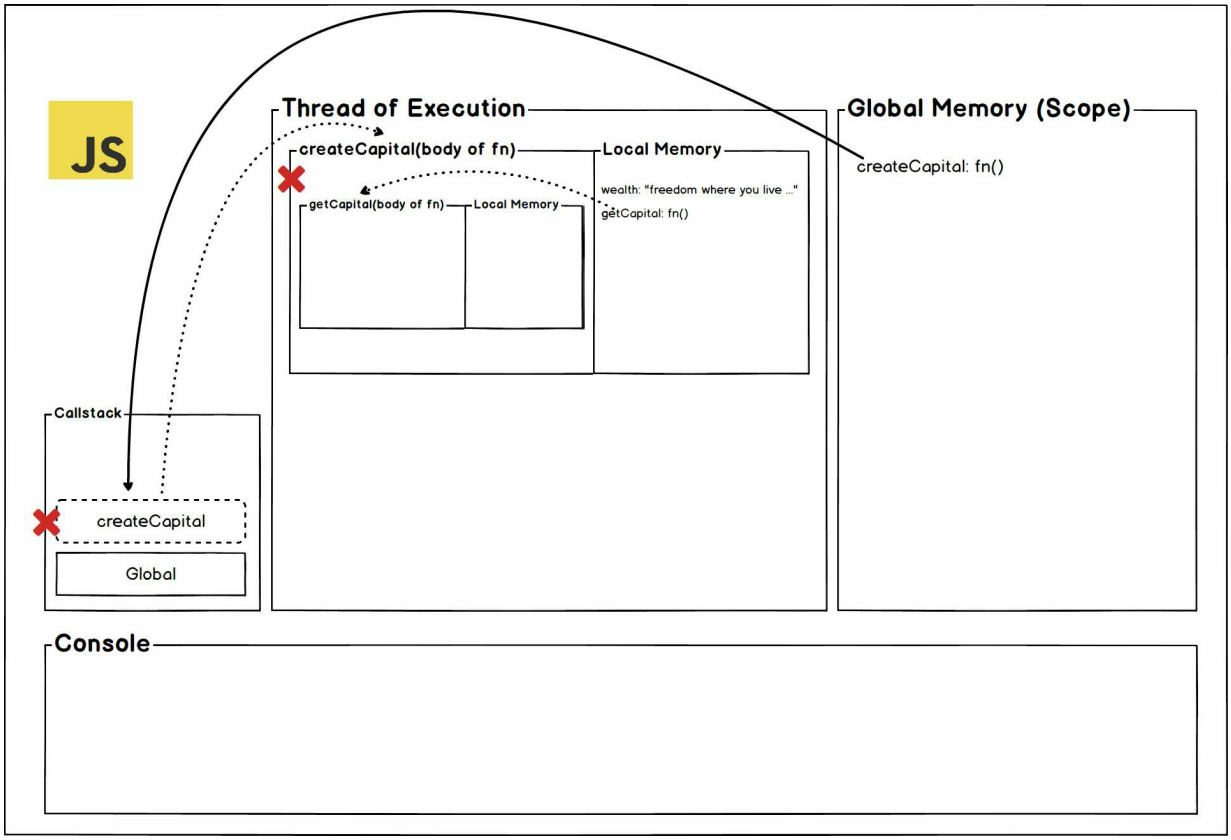
Function execution

Our first Mental model for function definition goes here.

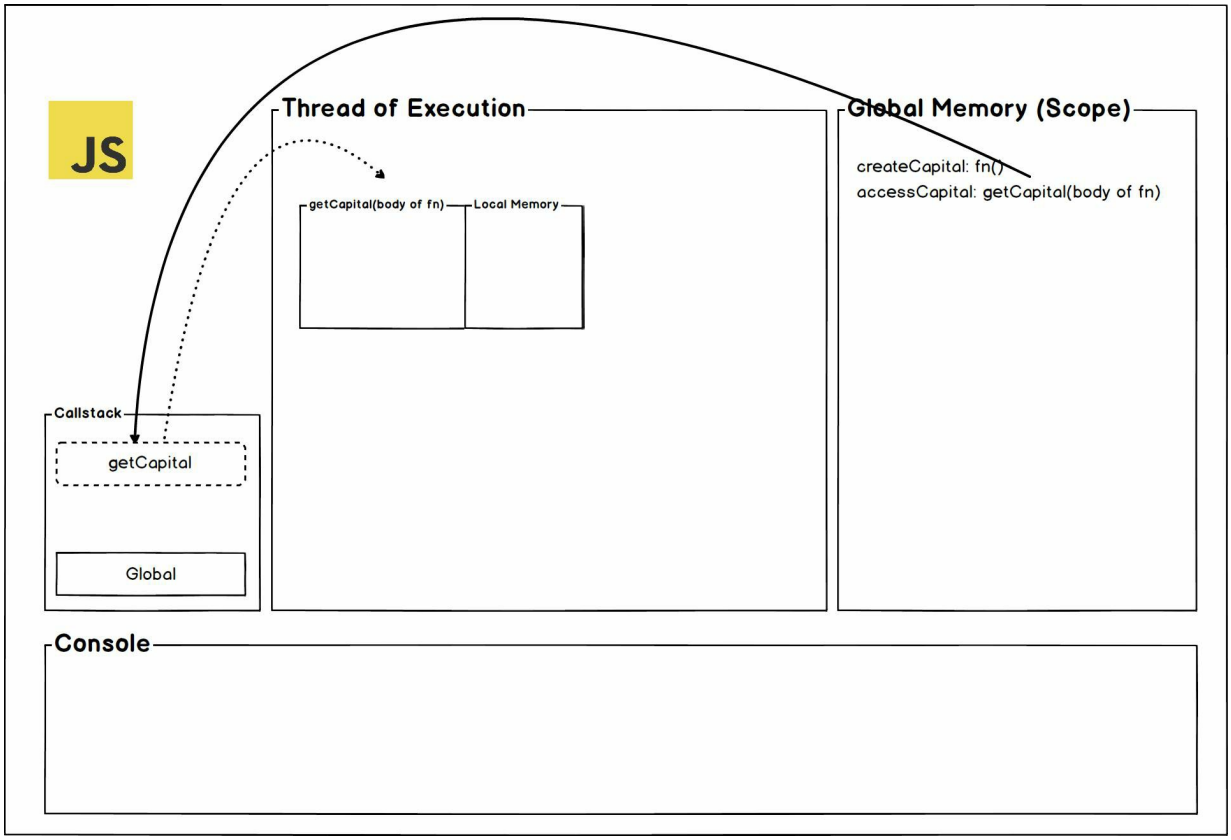


- As per the current mental model as you can see that, from global memory space to callstack and then from callstack to execution thread we have put the **createCapitial** function into execution.
- A function **createCapital** has its own local scope (environment) and inside that, we have **getCapital** function definition and stored.
- Now let's see what happens when we finish executing **createCapital** and we have returned the **getCapital.**
- Your best guess should be, the entire body and local scope of **createCapital** will be destroyed and the **getCapital** function will be

stored in the variable called **accessCapital.** Let's look at our mental model again.



Function createCapital in execution

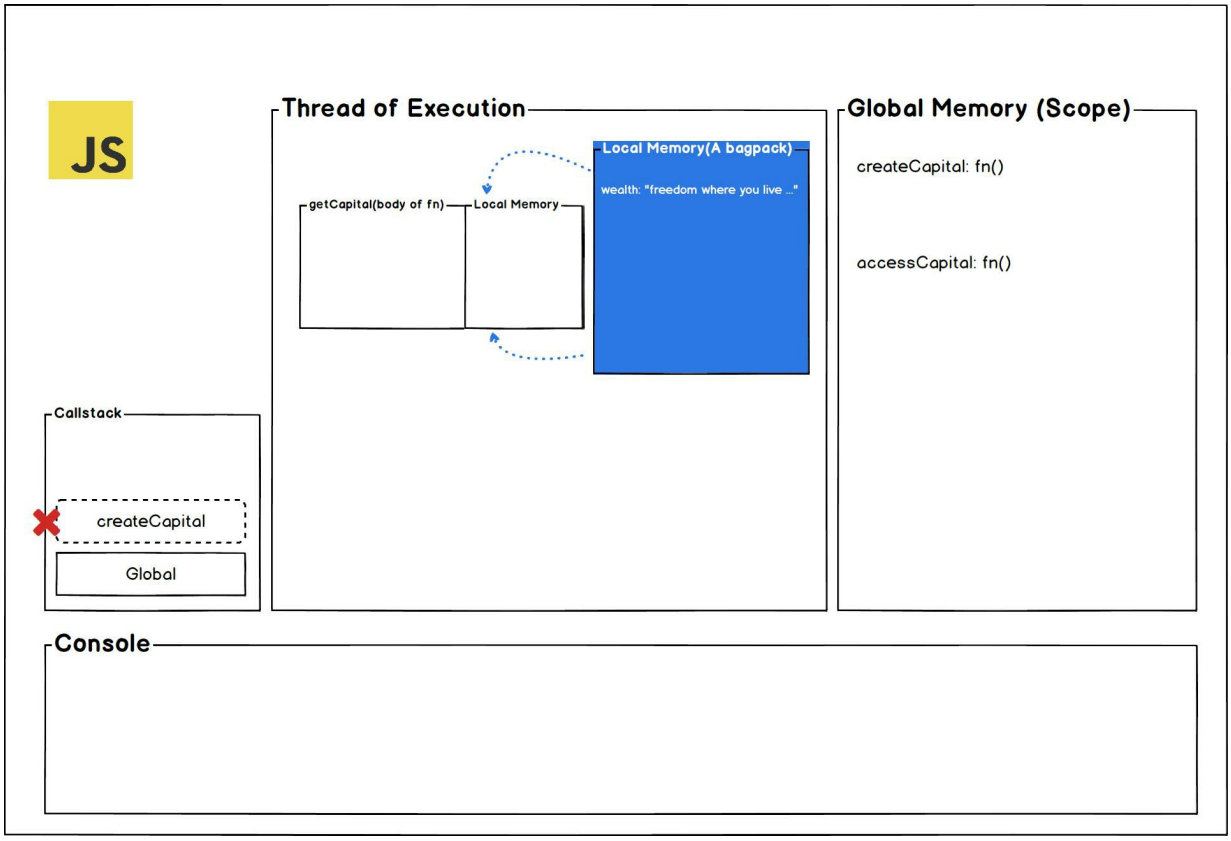


Function getCapital in execution

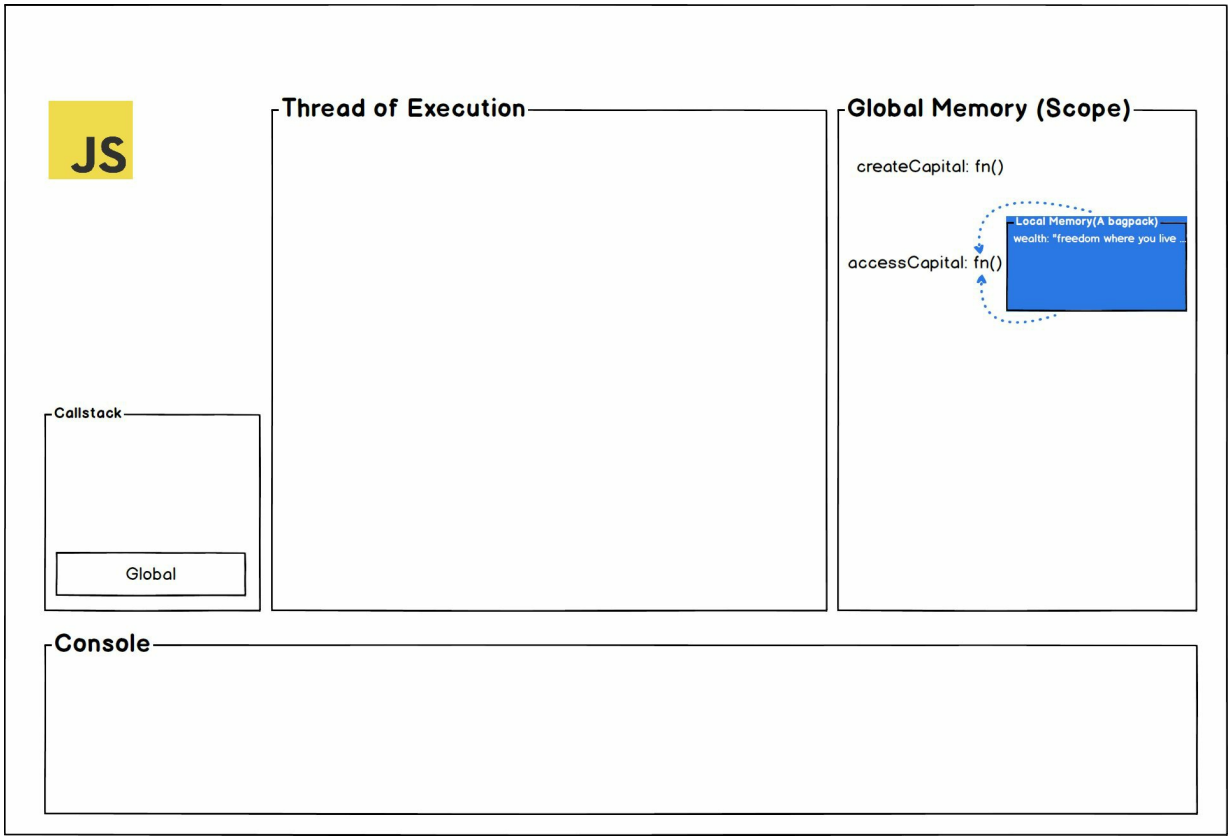
- As you can see from both images, we have finally reached the point where we are going to put the **getCapital** function into execution(**accessCapital** variable is used which is referring to our original **getCapital** function definition).
- But wait, as per normal behavior we should expect that variable **wealth** should be **undefined** when we execute the **getCapital** and it should not have access to the local memory of the executed function(aka memory of **createCapital**)**.** But, life is not that simple man😂**!!**
- Introducing a new friend called **backpack** to rescue operation.

```
A backpack is a simple memory store with strings attached to it(imagine as a real backpack🎒), and where these strings of the backpack will going to be attached is a question .
```

Let’s actually visualize what actually happened when the execution of our outer function createCapital finished!

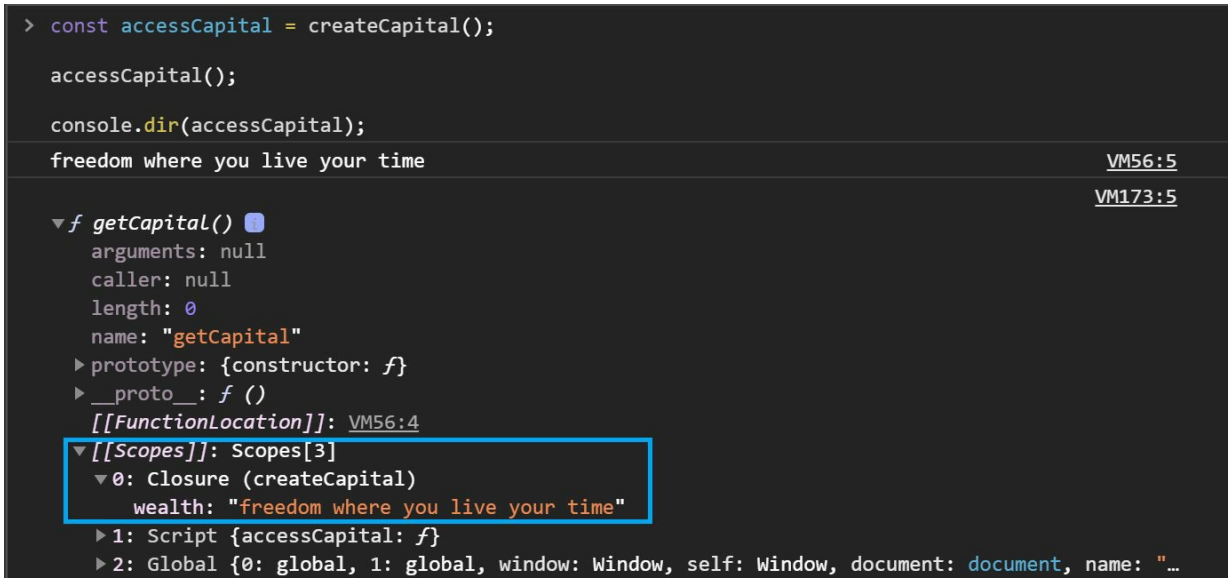


Attached backpack to the inner function



As you can see our backpack is nothing but a variable/s of an outer function's scope, and when JavaScirpt sees this kind of outer scope combine with inner scope(outer function returning inner function), instead of wiping the memory it keeps those variables as internal properties of inner function!

Also, if we want to see those internal properties then we can. Let's see in the below image where those are!

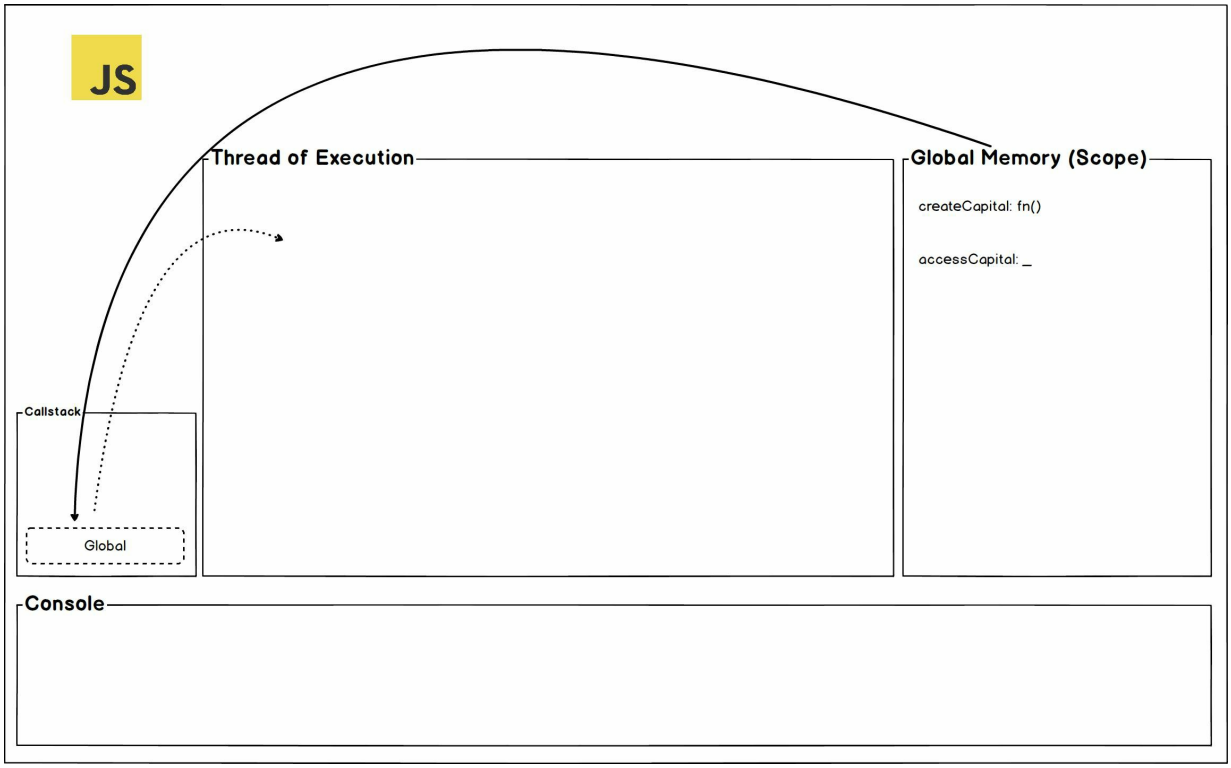


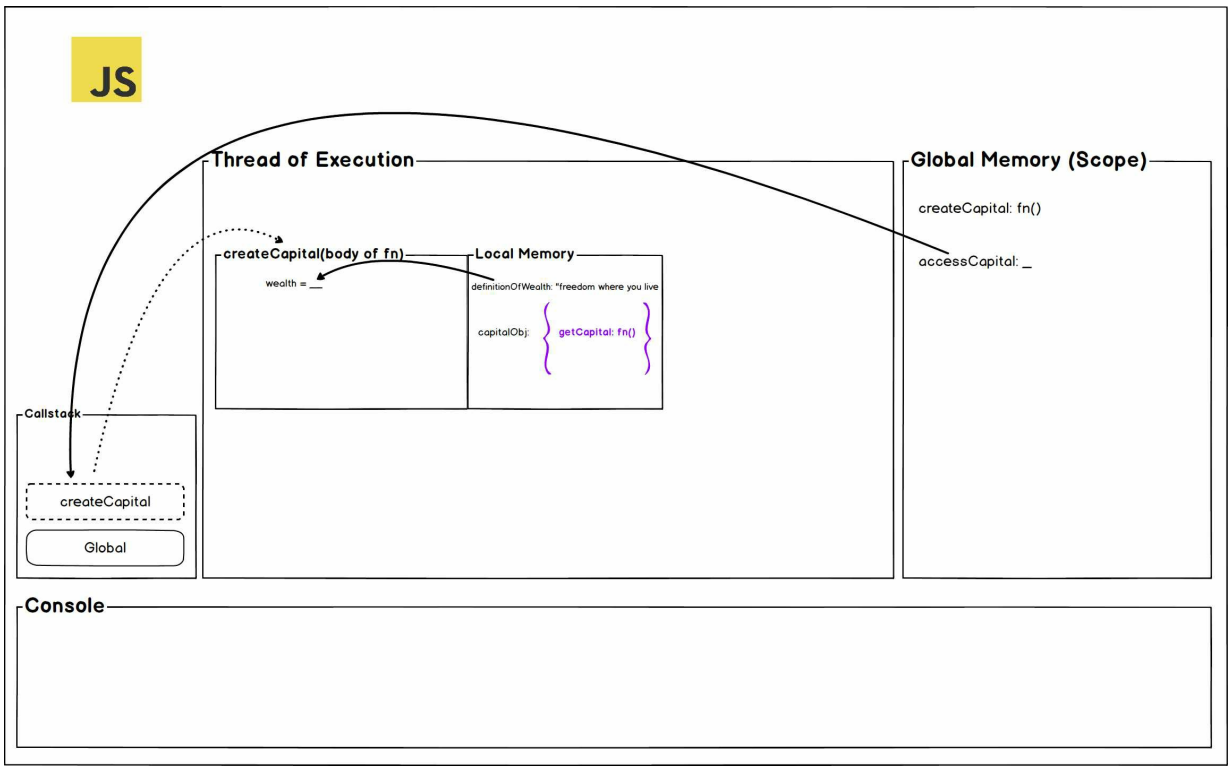
Hidden backpack

- As we can see above, the local scope (local memory) of the outer function (createCapitial) gets attached to the inner function's internal key-value pair! (***[[Scope]]***).
- ***[[Scope]]*** key has 3 array elements and the first element is where our backpack memory is stored, that is to say, it's our **Closure.**
- Also, the second element of that function is actually stat that, for which function that **Closure** is!
- The last element is actually our global scope which is actually available to every function (It's like global closure for each function object)
- There you have it, my friends. this is closure with a mental model😎!
- We can take one example on it to understand it better. Because even if don't return an inner function from the outer function, we could have closure. Let's see with an example.

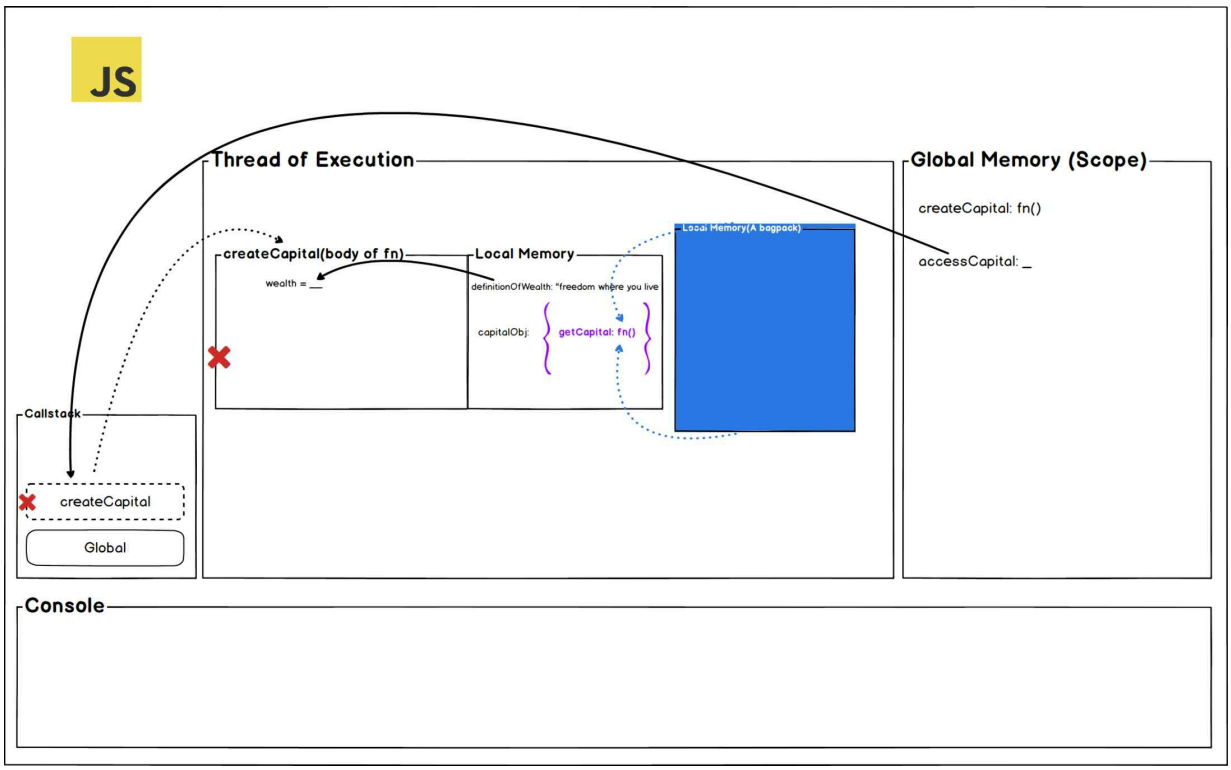
```
39
40  function createCapital(definitionOfWealth) {
41      let wealth = definitionOfWealth; // your definition for wealth😅
42
43      const capitalObj = {
44          getCapital: function() {
45              console.log(wealth)
46          }
47      }
48
49      return capitalObj;
50  }
51
52  const accessCapital = createCapital("freedom where you live your time");
53
54  accessCapital();
55
```

- This example is almost similar to the previous one except here we are returning an object instead of a function. This object however has a function in it. Let's start visualizing as well.

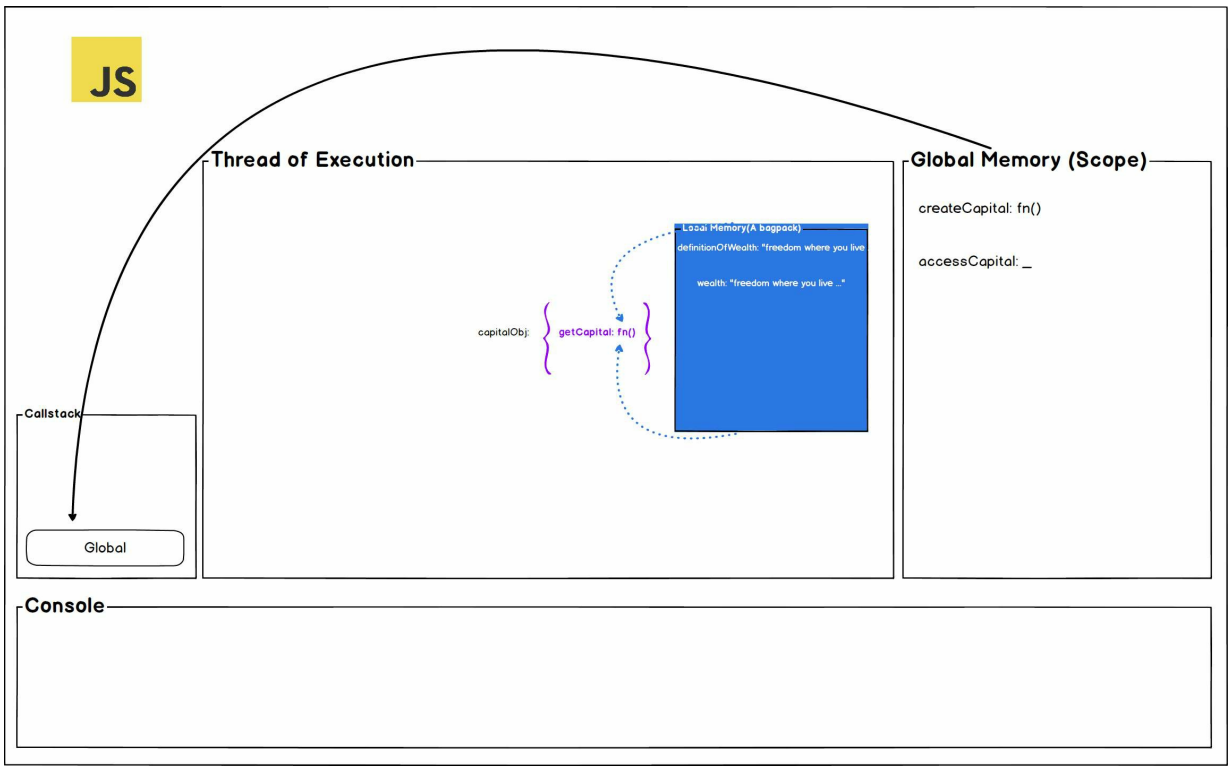


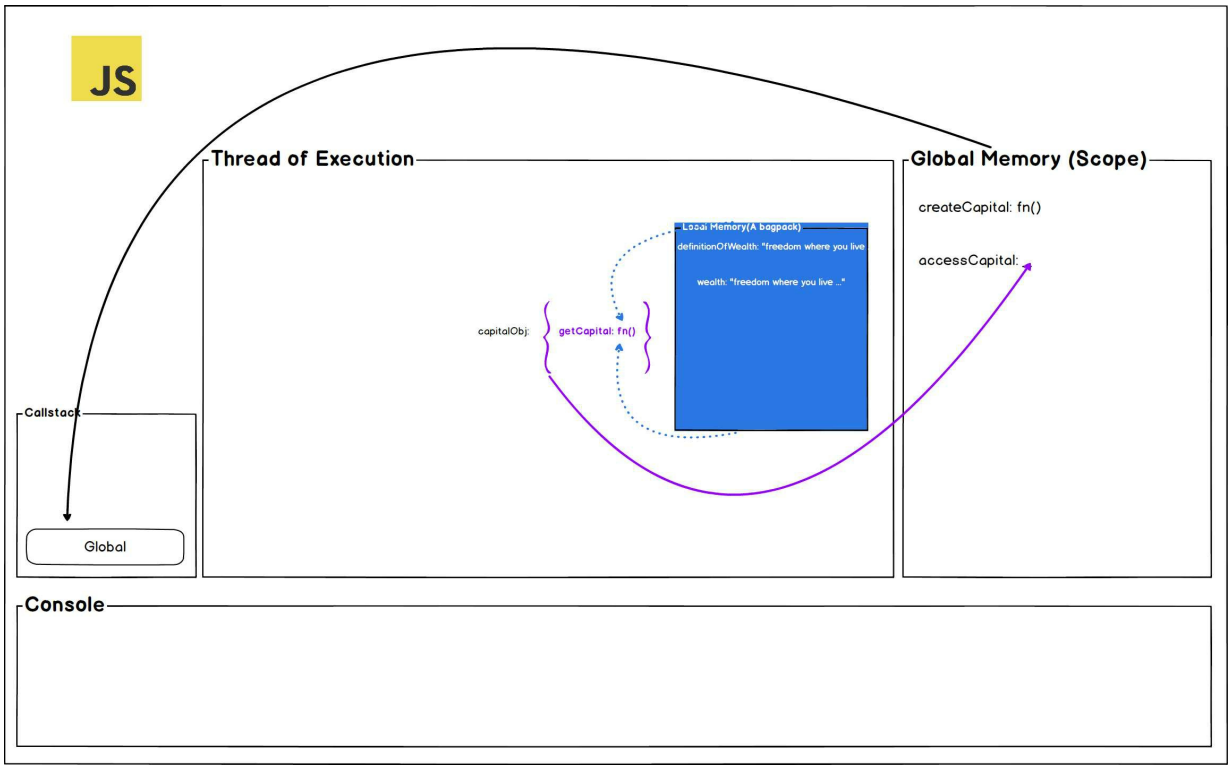


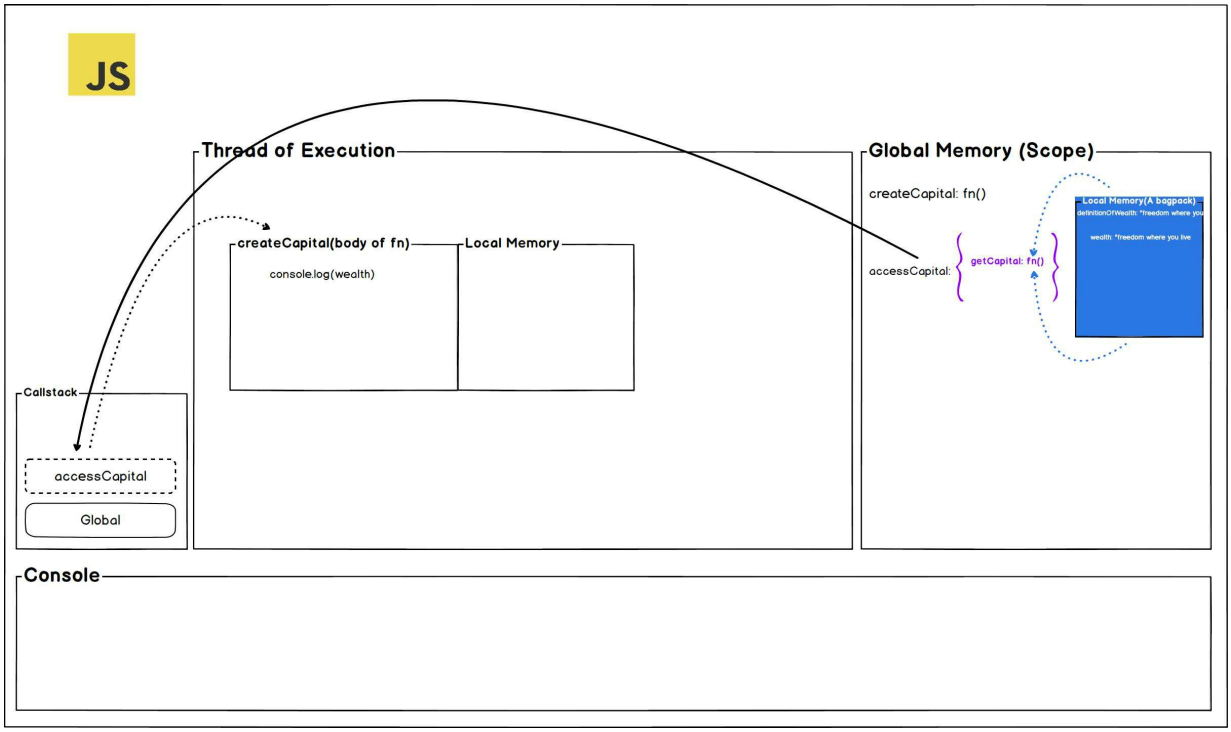
- Arguments in function are also part of local memory for function execution.



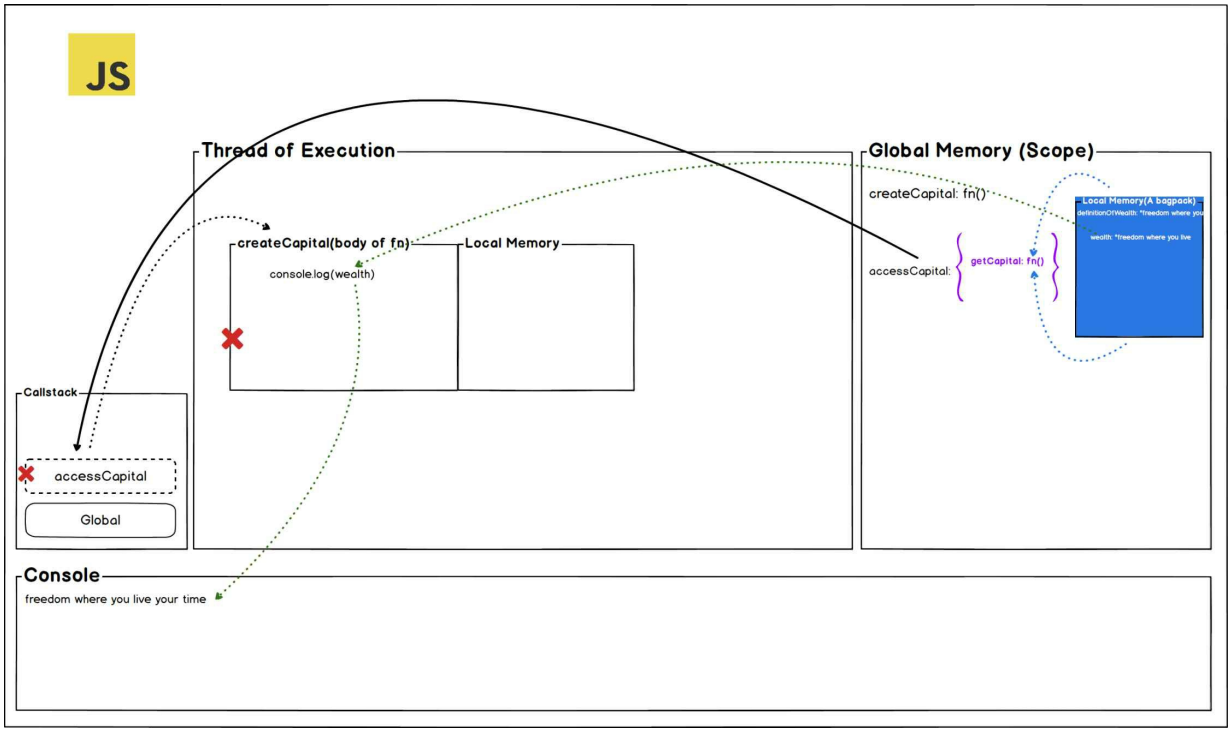
- The function inside the object will get local memory of the outer function when the execution of the outer function gets destroyed. see below😁.


JS
Thread of Execution
Global Memory (Scope)
createCapital: fn()
accessCapital: _
Local Memory(A bagpack)
definitionOfWealth: "freedom where you live
wealth: "freedom where you live ..."
capitalObj:
getCapital: fn()
Callstack
Global
Console

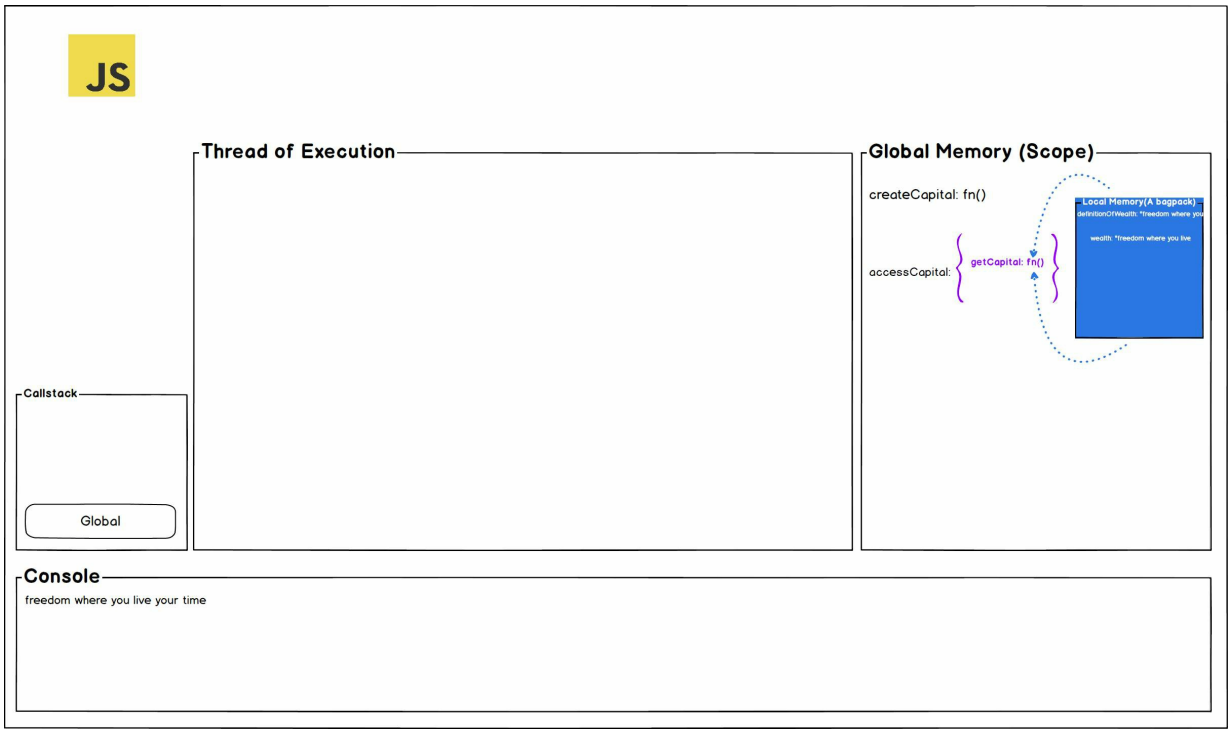

JS
Thread of Execution
Global Memory (Scope)
createCapital: fn()
accessCapital:
Local Memory(A bagpack)
definitionOfWealth: "freedom where you live
wealth: "freedom where you live ..."
capitalObj:
getCapital: fn()
Callstack
Global
Console



- Even without outer function, we are now able to access variables of it through closure as shown. Moreover, note that both **local variables and arguments** are made available through closure.



- finally, we completed our execution and the console gave the output.



That is all my friends for closure, sometimes they call our closure/backpack **‘Lexical Scope’** as well. let’s just a have quick note before we move on to another chapter with a new visualization journey.

```
Whenever we have an object with function property in it or function inside a
function, all surrounding variable is made available to object's function property
or inner function even after execution of parent(outer) function get finished
through closure. we understood both by example. we will later in the chapter use
this concept as well.
```

* * *

3

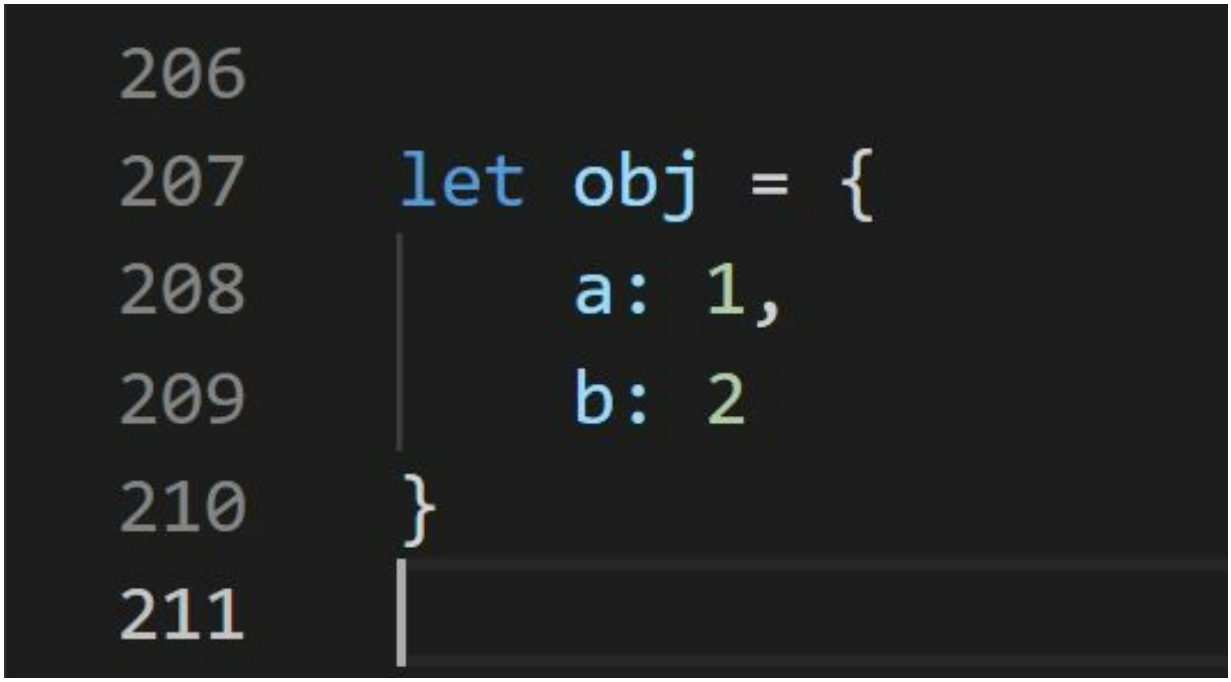
Prototype , __proto__ & Objects

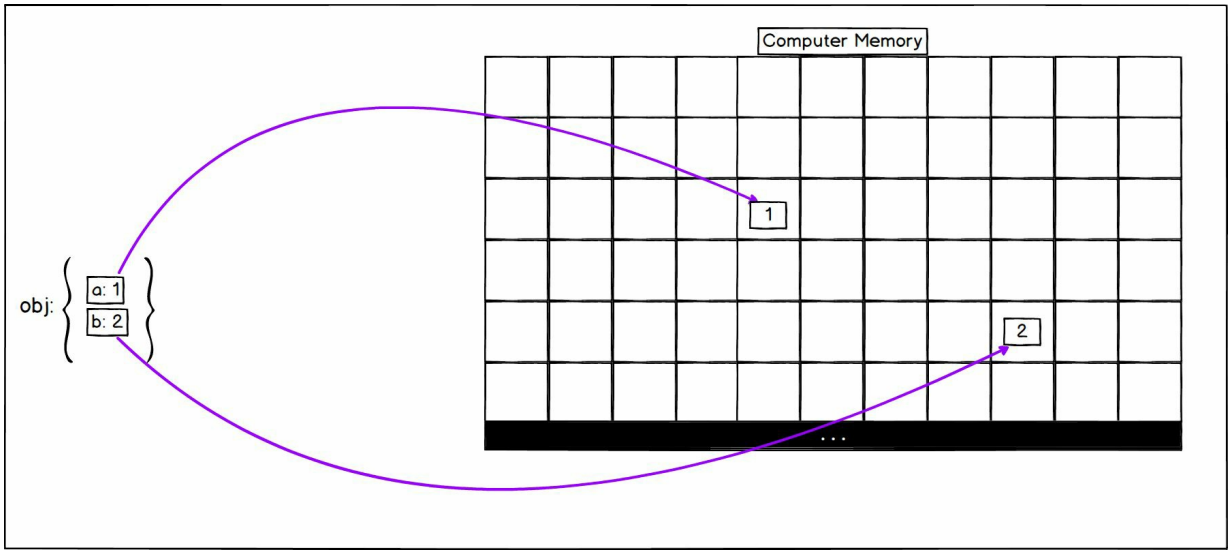
```
This chapter is all about shifting your mind if you are coming from other
programming languages or have traditional OOPs concepts in your mind. JavaScript
and its prototype nature is a completely different concept. Moreover, JavaScript's
lot of features try to mimic syntax from other programming languages, and because
of that a lot of confusion starts. In this chapter, we will see an in-depth look at
how things work under the hood with visuals.
```

Let’s start with very simple stuff and that is **Objects.**

Objects

- In JavaScript, objects are simply key-value pairs(you can think of a collection of variables are put together).
- However, just because key-values are put together doesn’t mean they are stored together in memory. Under the hood, all variables are stored randomly in memory, but references for those variables know how to target value exactly based on a key is accessed.
- Take a look at how the below object is stored in computer memory.

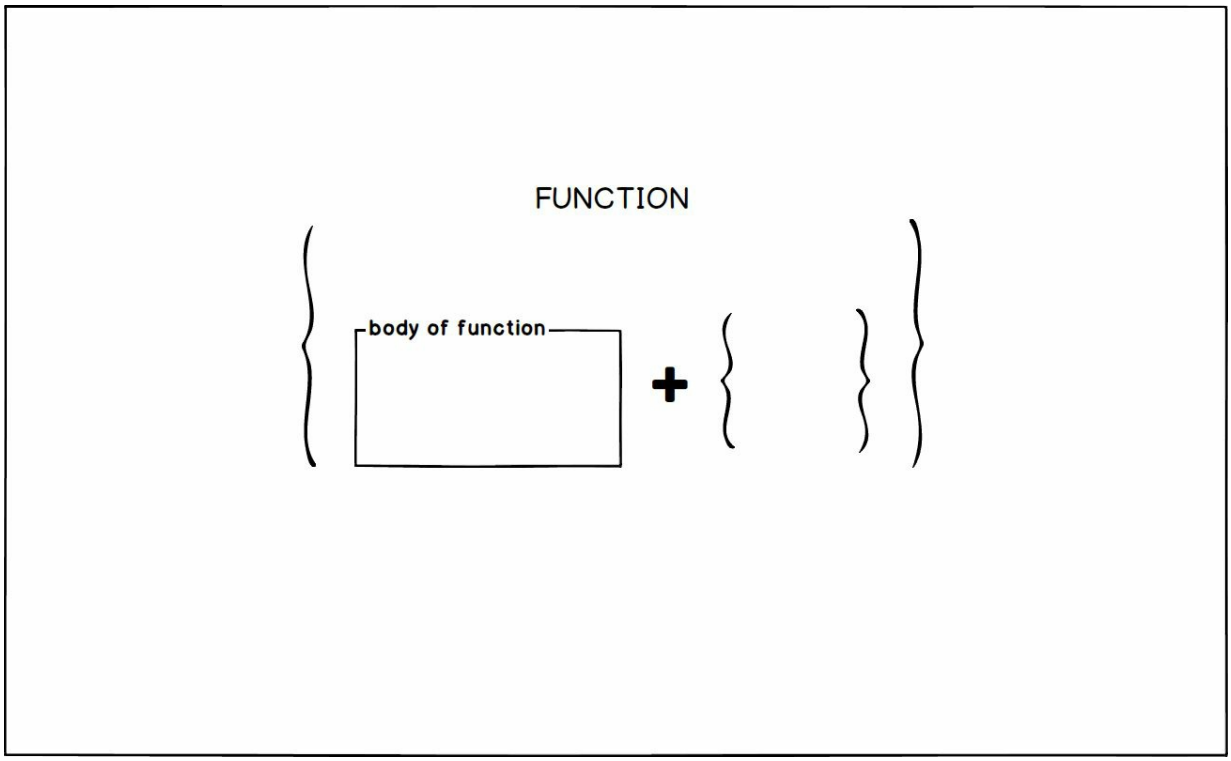




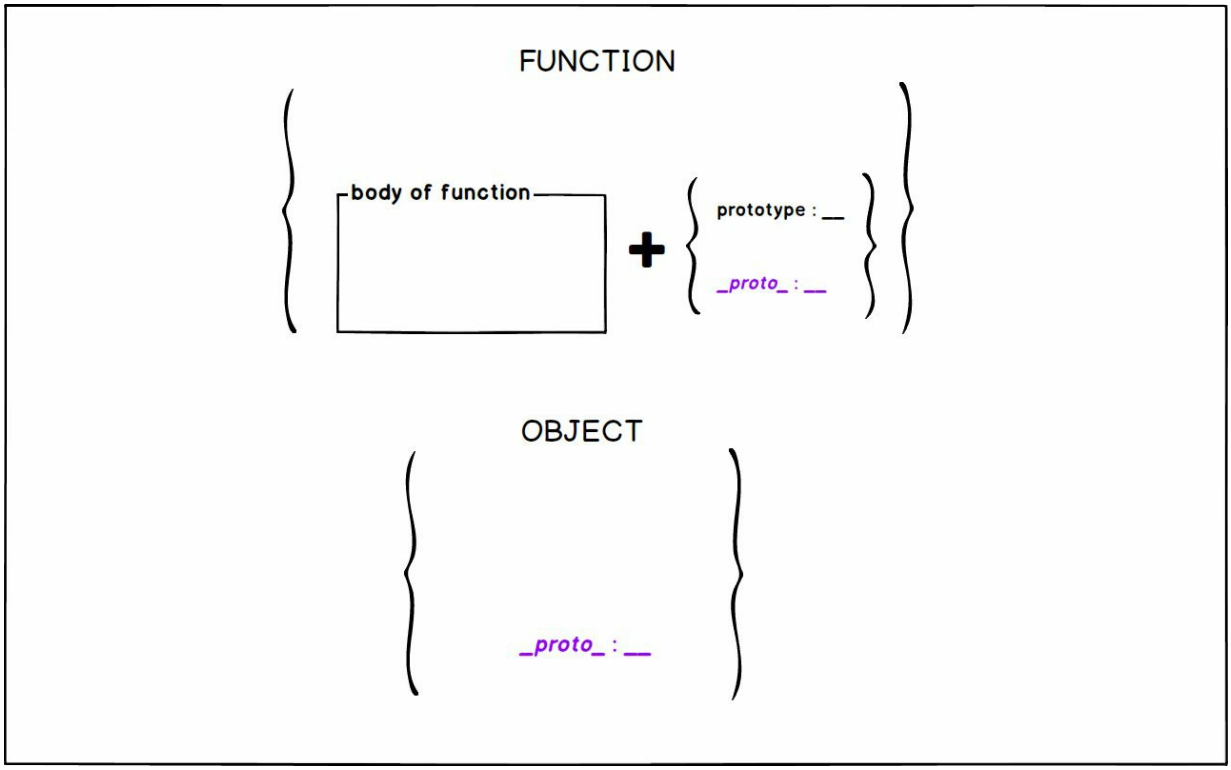
The main reason to show the above diagram is to show that logically put together values is not how exactly it is stored in memory.

Objects and Primitive values

- In JavaScript everything except **primitive values** (strings, numbers, boolean, etc) are **objects.**
- **Functions are objects as well.**
- **Arrays are objects**
- **Math, Date, and other helper functions are also objects.**
- Out of all this, functions are most important. Functions are objects, but it's a special type of object.
- Functions have two parts available to use. The first part is the function body where we can store a block of code and later invoke it whenever we need it. The second part is the object part where we can store key-value pairs.



- We have already seen proof of work in chapter 2 where we added properties to the function object(recheck if have forgotten😄). Over here, we are going to see further what we can do or available to us.
- Interestingly, every function object has a **prototype** property available by default and every object has **__proto__** property available as well.
- so, every function will have a **prototype and __proto__** property and every object will have **__proto__*(no prototype property on a normal object, only available to function object only)*** available to us.



- In this chapter I will not draw the entire javascript environment as it's not needed here, we will purely focus on how objects and function with their linking works. Let's start with a simple example below(don't try to understand right now, will do that).

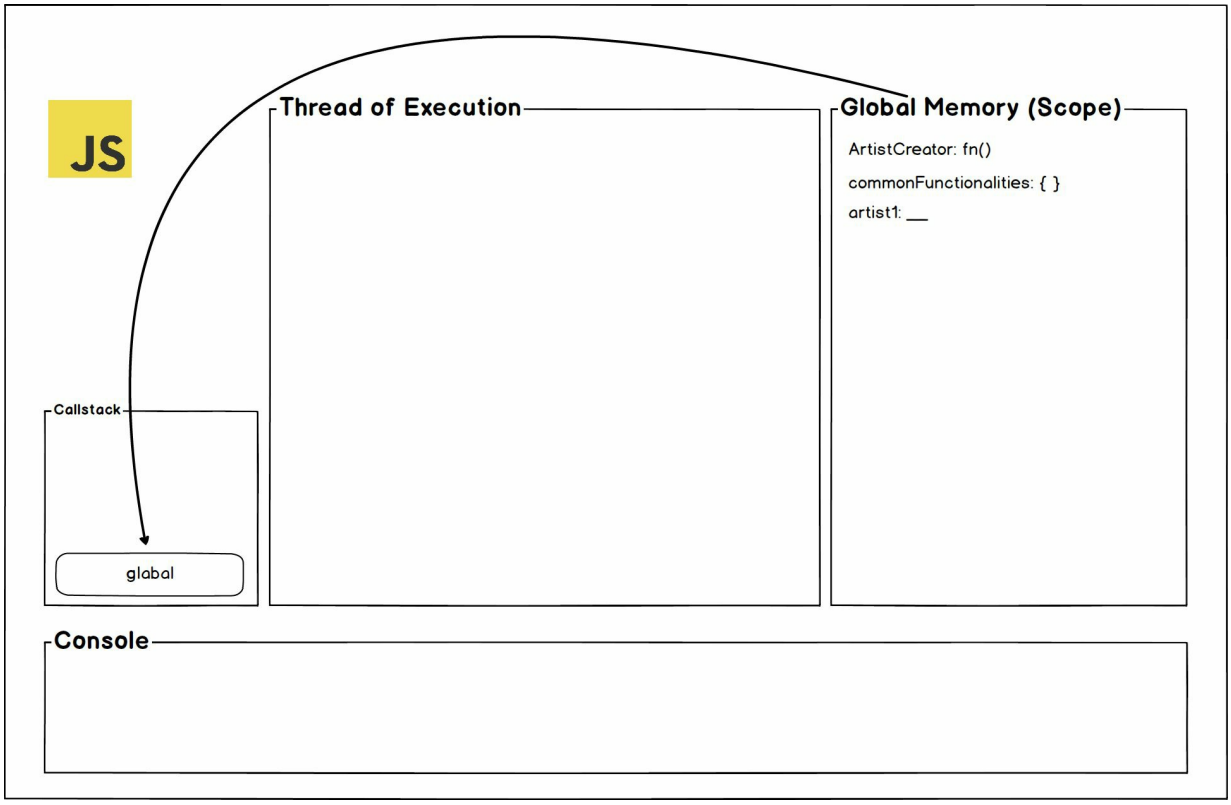
```
212
213  function ArtistCreator(name, art) {
214      const newArtist = Object.create(commonFunctionalities);
215
216      newArtist.name = name;
217      newArtist.art = art;
218
219      return newArtist;
220  }
221
222  const commonFunctionalities = {
223      isHuman: function() {
224          console.log(this.name);
225          return true;
226      },
227      isCuriuos: function() {
228          console.log(this.name);
229          return true;
230      }
231  }
232
233  let artist1 = ArtistCreator('John', 'Painter');
234
```

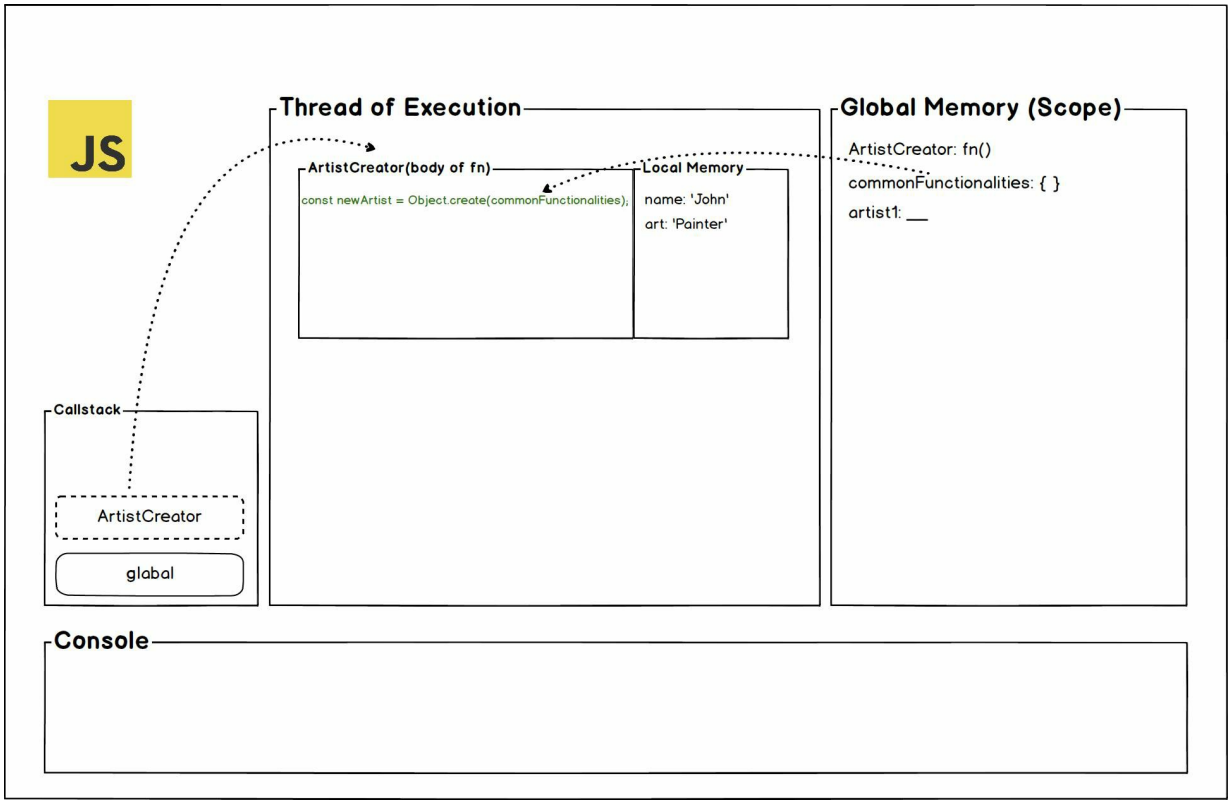
- As you can see we have created the **ArtistCreator** function, which helps us to create objects. you can think of it's a common function to create artist objects. we soon visualize it but before we do that let's understand what **Object.create()** does for us.

```
212
213  function ArtistCreator(name, art) {
214      const newArtist = Object.create(commonFunctionalities);
215
216      newArtist.name = name;
217      newArtist.art = art;
218
219      return newArtist;
220  }
221
```

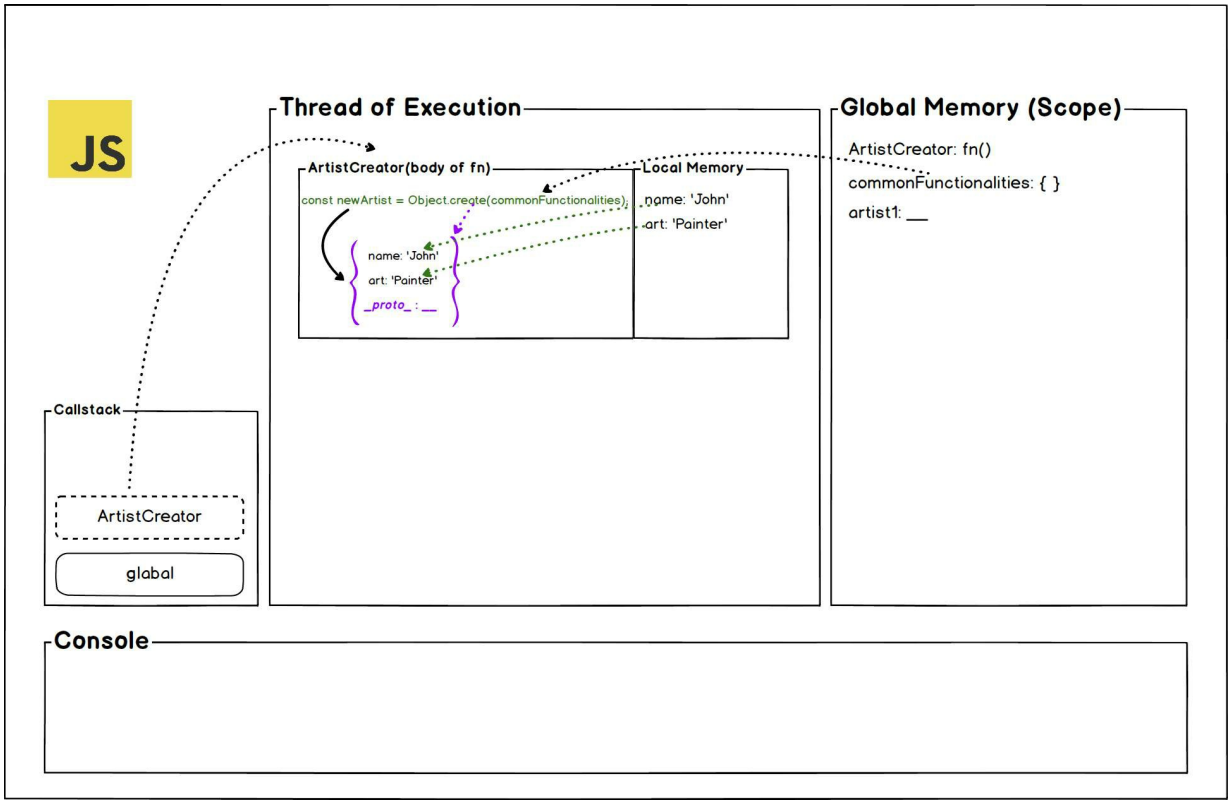
- We have a lot of global objects(key-value pairs) available to us in the javascript. **Object**(notice capital ‘O’) is one of them. which has a lot of key-value pairs and it is made available by javascript.
- The **“create”** is a method of **Object** is available to us and it has fundamental below task.

1. Create an empty object.
2. And because every object has __proto__ property available, after the creation of an empty object whatever we pass as an argument to the **“create”** method, will be available to this __proto__ directly to the newly created object. let’s see how.

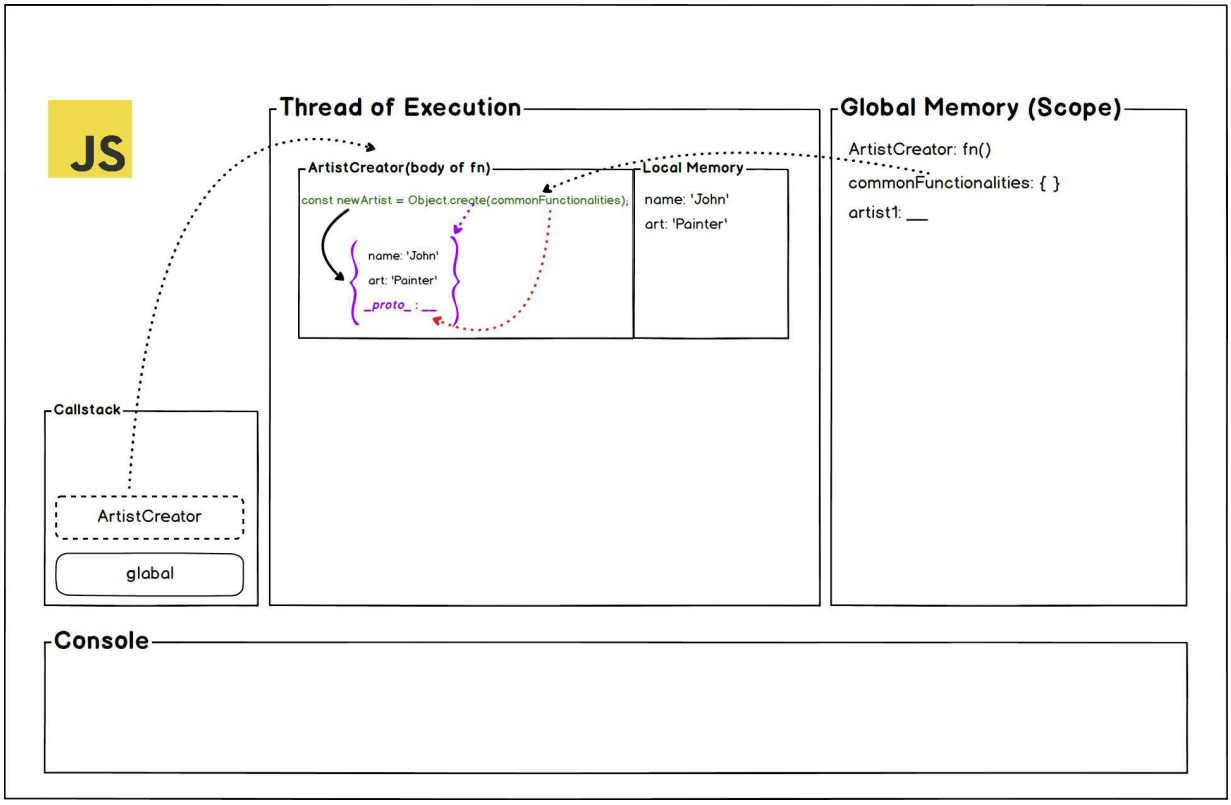


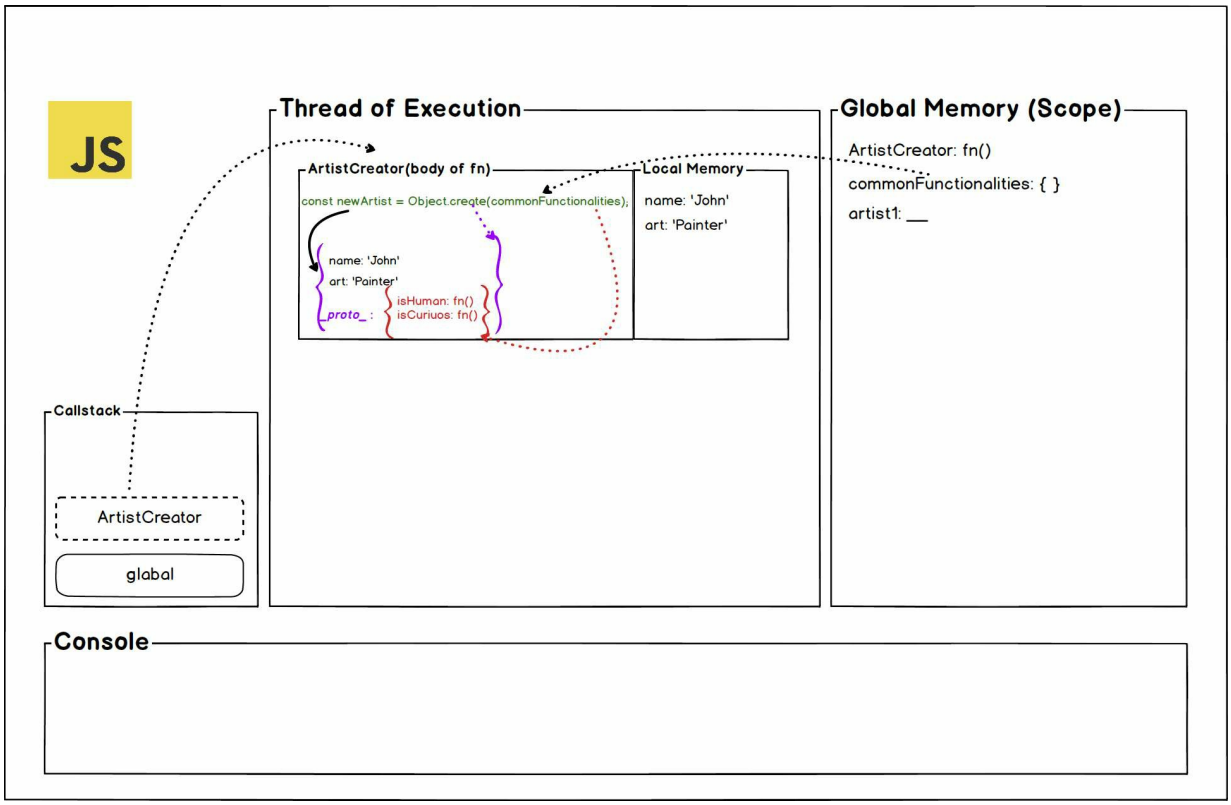


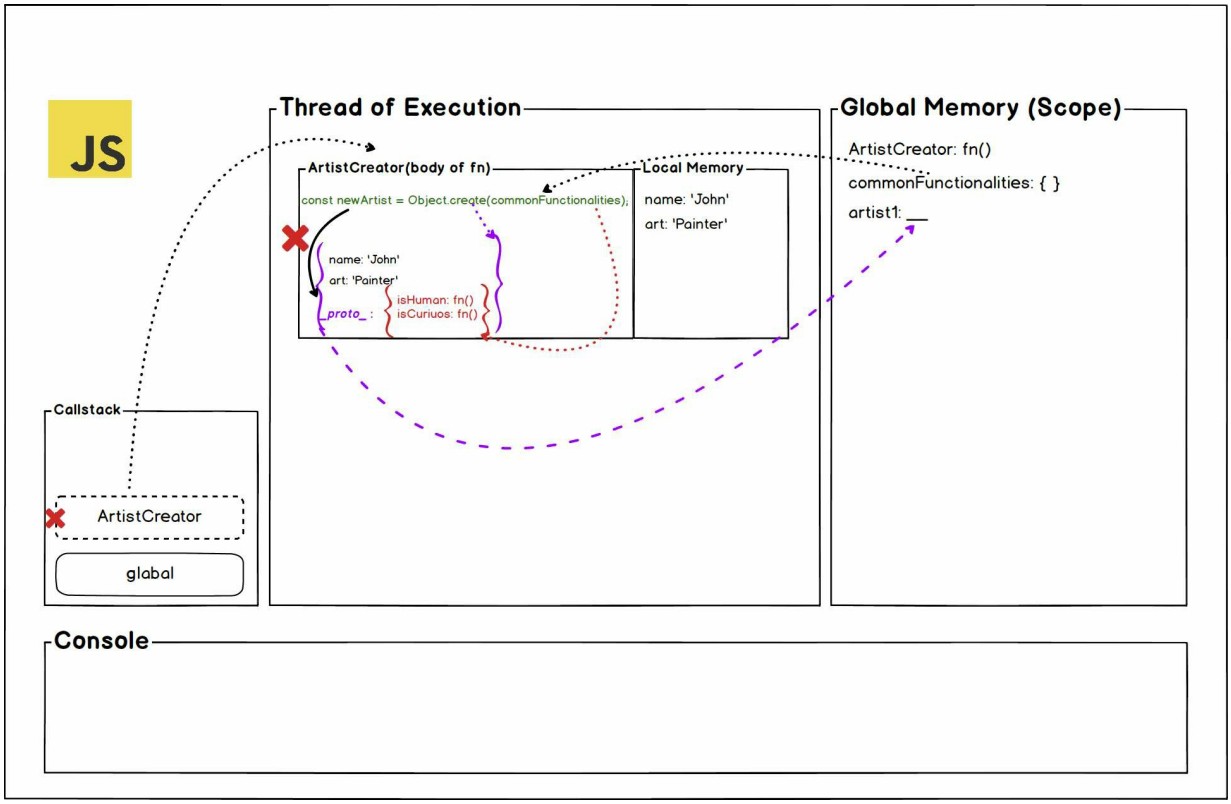
- we are executing the ArtistCreator function and it’s executing the Object.create() method with our globally stored commonFunctionality object.

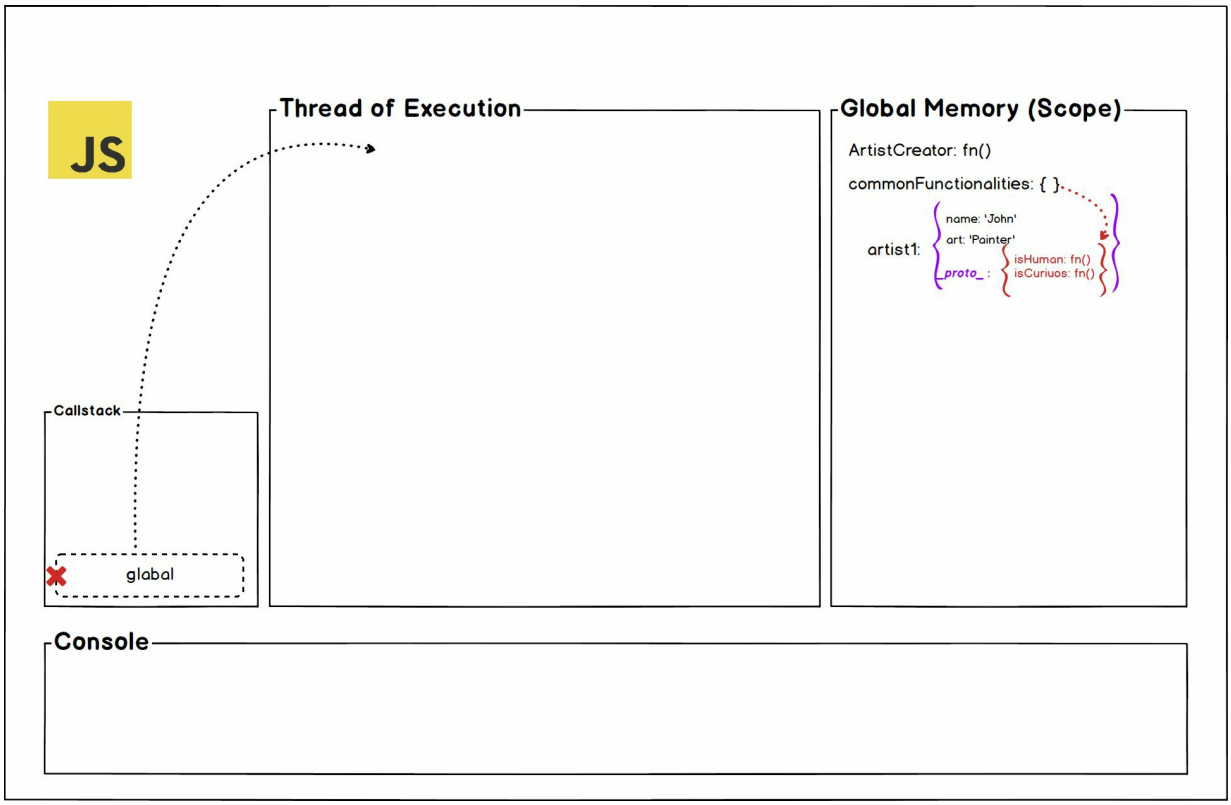


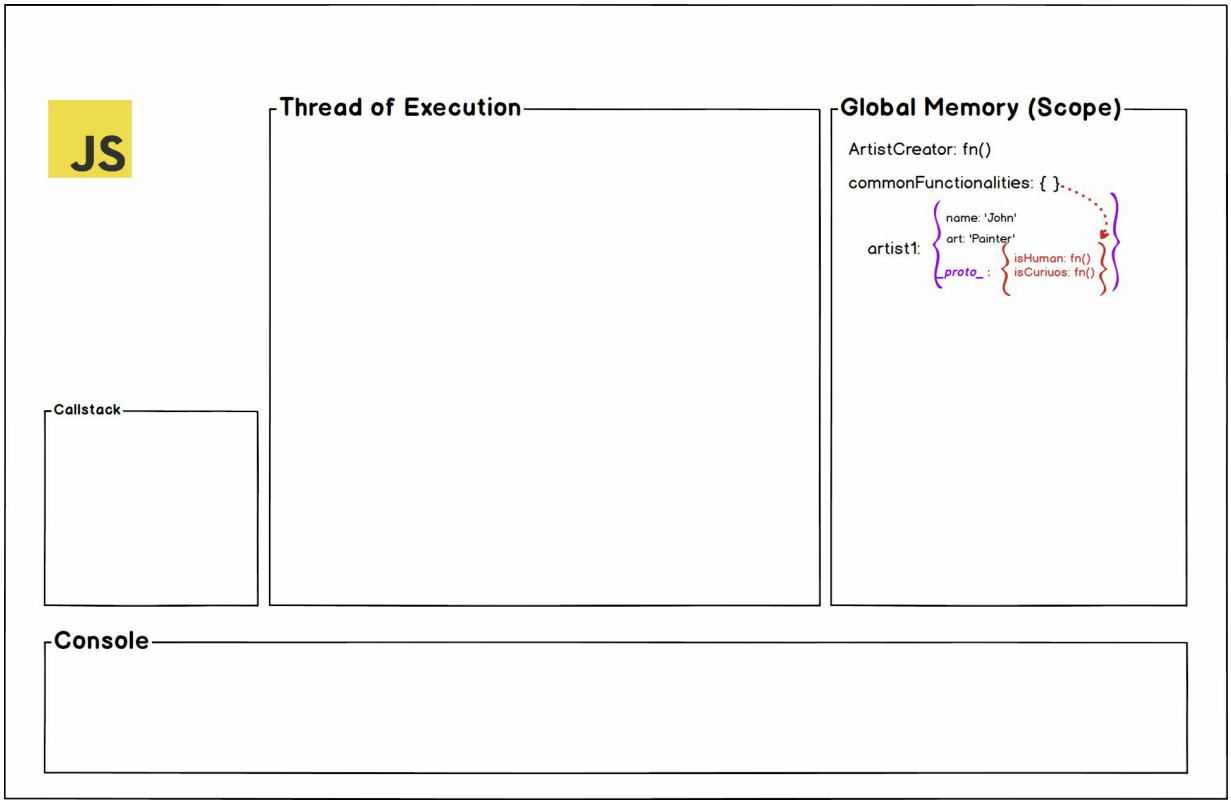
- Object.create() not just created a new empty object but also added a __proto__ reference to our globally stored **commonFunctionalities** object**.** so, whatever we passed to the create()'s first argument will available to __proto__ property of the newly created object. Let's finish our visualization.


JS
Thread of Execution
ArtistCreator(body of fn)
const newArtist = Object.create(commonFunctionalities);
name: 'John'
art: 'Painter'
proto : __
Local Memory
name: 'John'
art: 'Painter'
Global Memory (Scope)
ArtistCreator: fn()
commonFunctionalities: { }
artist1: __
Callstack
ArtistCreator
global
Console


JS
Thread of Execution
ArtistCreator(body of fn)
const newArtist = Object.create(commonFunctionalities);
name: 'John'
art: 'Painter'
proto :
isHuman: fn()
isCuriuos: fn()
Local Memory
name: 'John'
art: 'Painter'
Global Memory (Scope)
ArtistCreator: fn()
commonFunctionalities: { }
artist1: __
Callstack
ArtistCreator
global
Console


JS
Thread of Execution
Global Memory (Scope)
ArtistCreator: fn()
commonFunctionalities: { }
artist1: __
ArtistCreator(body of fn)
Local Memory
const newArtist = Object.create(commonFunctionalities);
name: 'John'
art: 'Painter'
name: 'John'
art: 'Painter'
__proto__ :
isHuman: fn()
isCuriuos: fn()
Callstack
ArtistCreator
global
Console

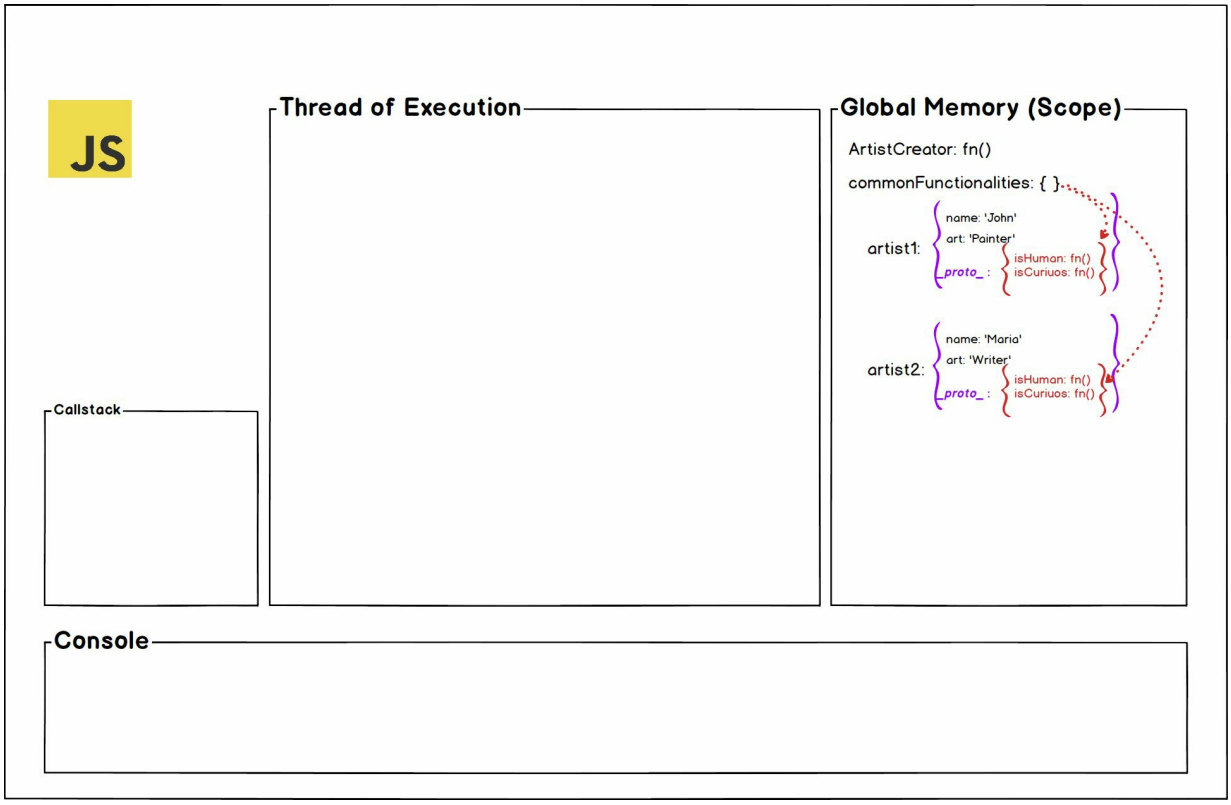

JS
Thread of Execution
Global Memory (Scope)
ArtistCreator: fn()
commonFunctionalities: { }
artist1:
name: 'John'
art: 'Painter'
__proto__ :
isHuman: fn()
isCuriuos: fn()
Callstack
global
Console



- As you can see, we can create a new object using a function, and not only that we can reuse this function again and again as well.

```
235
236    let artist2 = ArtistCreator('Moria', 'Writer');
237
```

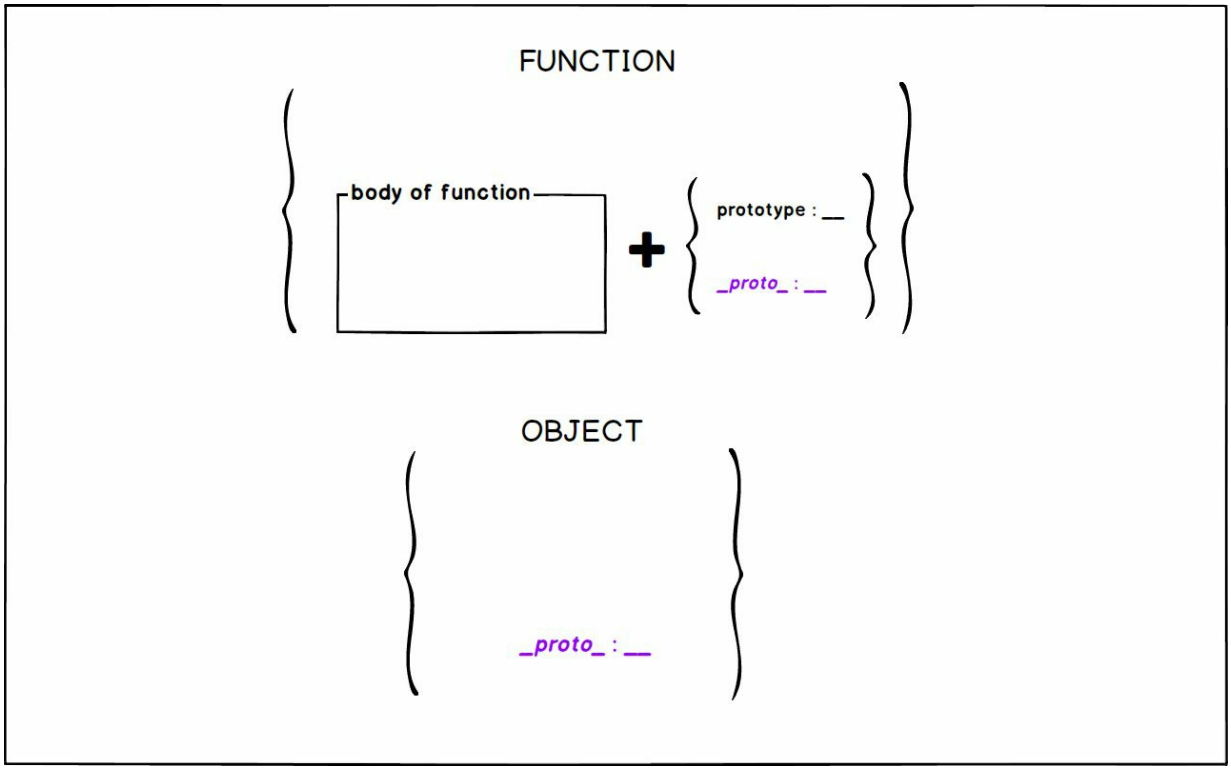
- If we run the same function again, as shown above, we will still able to reuse its functionality and also we can utilize already declared **commonFunctionalities.** See the below diagram we did the same for the **artist2** object as well(go through the mental model on your own just like we have shown above).



- Alright, we have created functionality that allows us to create objects whenever we need and reuse some of the existing functionalities as well. Now, what if javascript has this kind of feature available to us😉.
- Gladly, we do have, and that what we are going to do next. see below example and will understand shortly.

```
238
239  function ArtistCreator(name, art) {
240      this.name = name;
241      this.art = art;
242  }
243
244  ArtistCreator.prototype.isHuman = function() {
245      console.log(this.name);
246      return true;
247  }
248  ArtistCreator.prototype.isCuriuos = function() {
249      console.log(this.name);
250      return true;
251  }
252
253  let artist1 = new ArtistCreator('John', 'Painter');
254  let artist2 = new ArtistCreator('Moria', 'Writer');
255
```

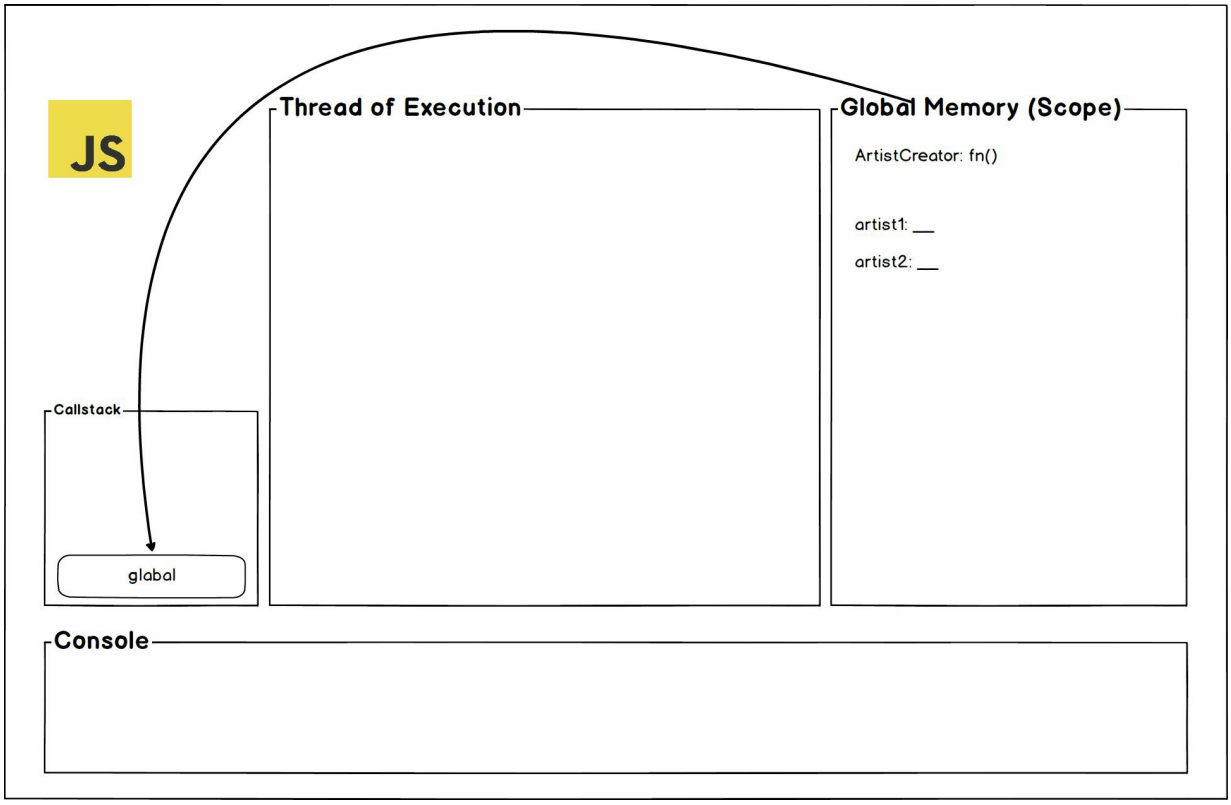
- In case you have forgotten, the function can have a prototype and __proto__ properties available by default. same way **ArtistCreator** will also have these properties available to them. Let me show that diagram again. By the way, a **prototype** is an object and we can add any key-value pairs to it.



- Now what to add to that prototype object is up to us. In our example, we added two functions **isHuman** and **isCuriuos.**

```
243
244  ArtistCreator.prototype.isHuman = function() {
245      console.log(this.name);
246      return true;
247  }
248  ArtistCreator.prototype.isCuriuos = function() {
249      console.log(this.name);
250      return true;
251  }
252
```

- Note, our example now has a **new** keyword as well. which does a lot of automation work behind the scene. let’s see how its diagram goes.

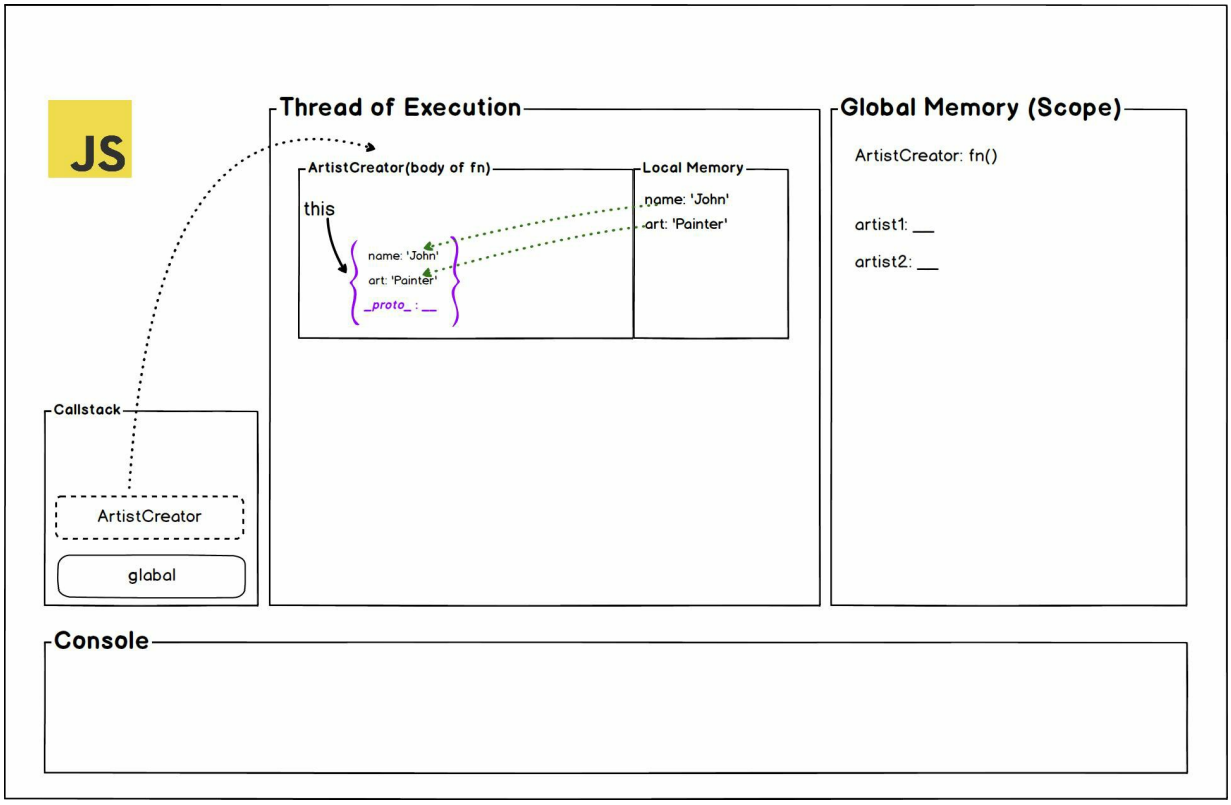


- let's execute the below line, note that the **new** keyword in front of the function call.

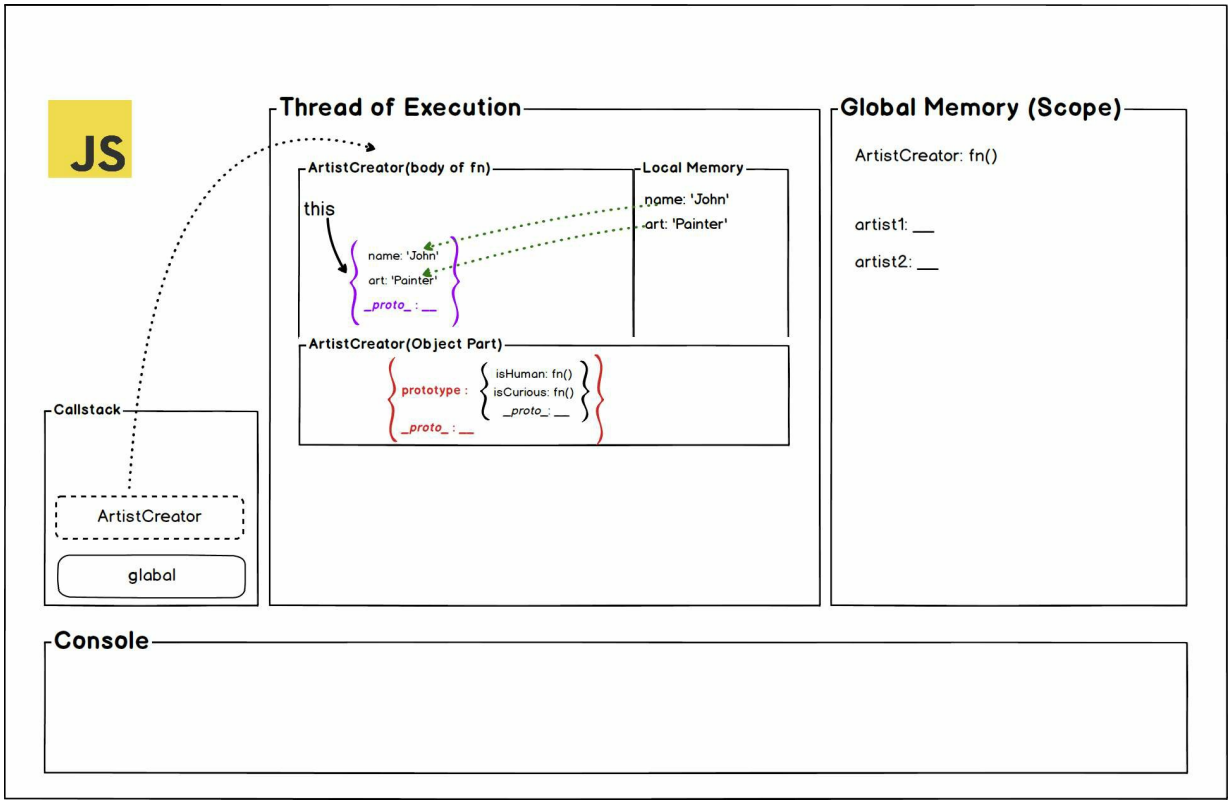
```
252
253   let artist1 = new ArtistCreator('John', 'Painter');
254
```

The **new** keyword does 2 major things.

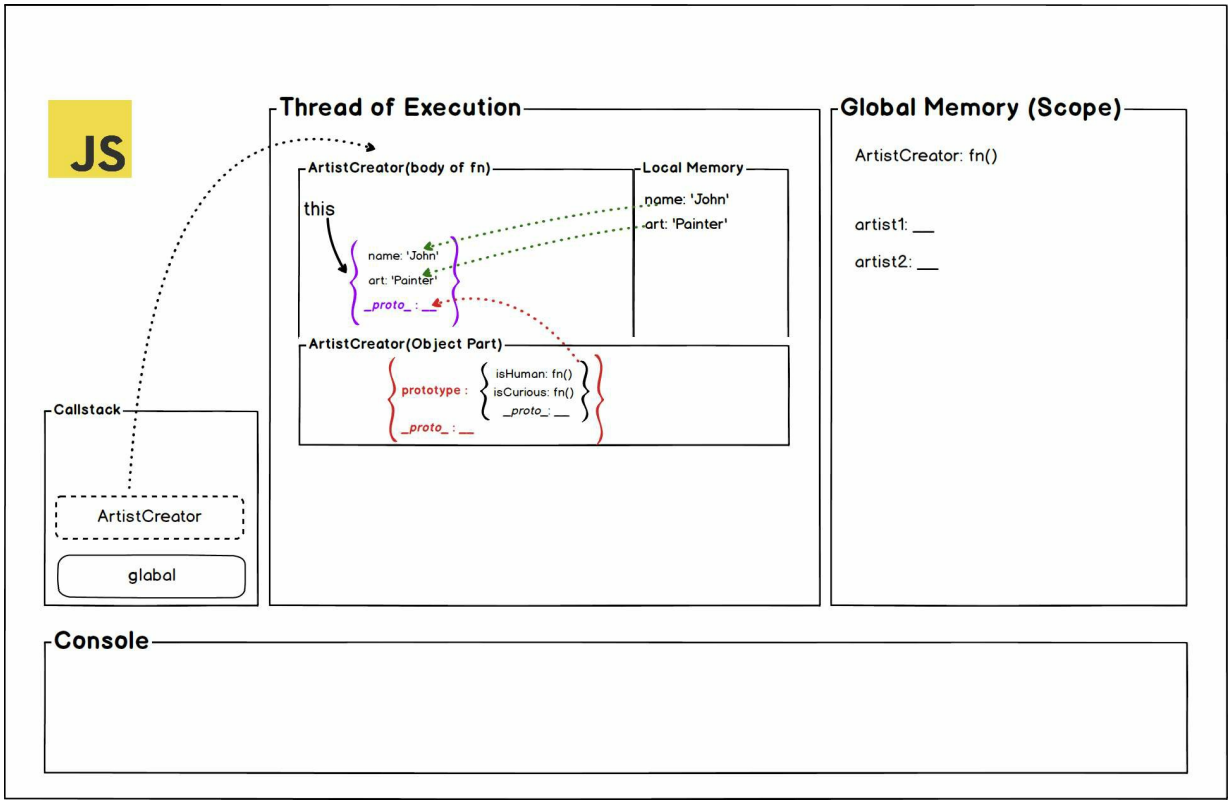
1. It will create a new **empty object** and a newly created object will get **this** reference. That means inside function execution **this** will point to newly created object by **new** keyword.
2. Moreover, whatever **prototype** object of function(**ArtistCreator** function in our example) has that will refer to __proto__ of newly created object(let's visualize of course😊).

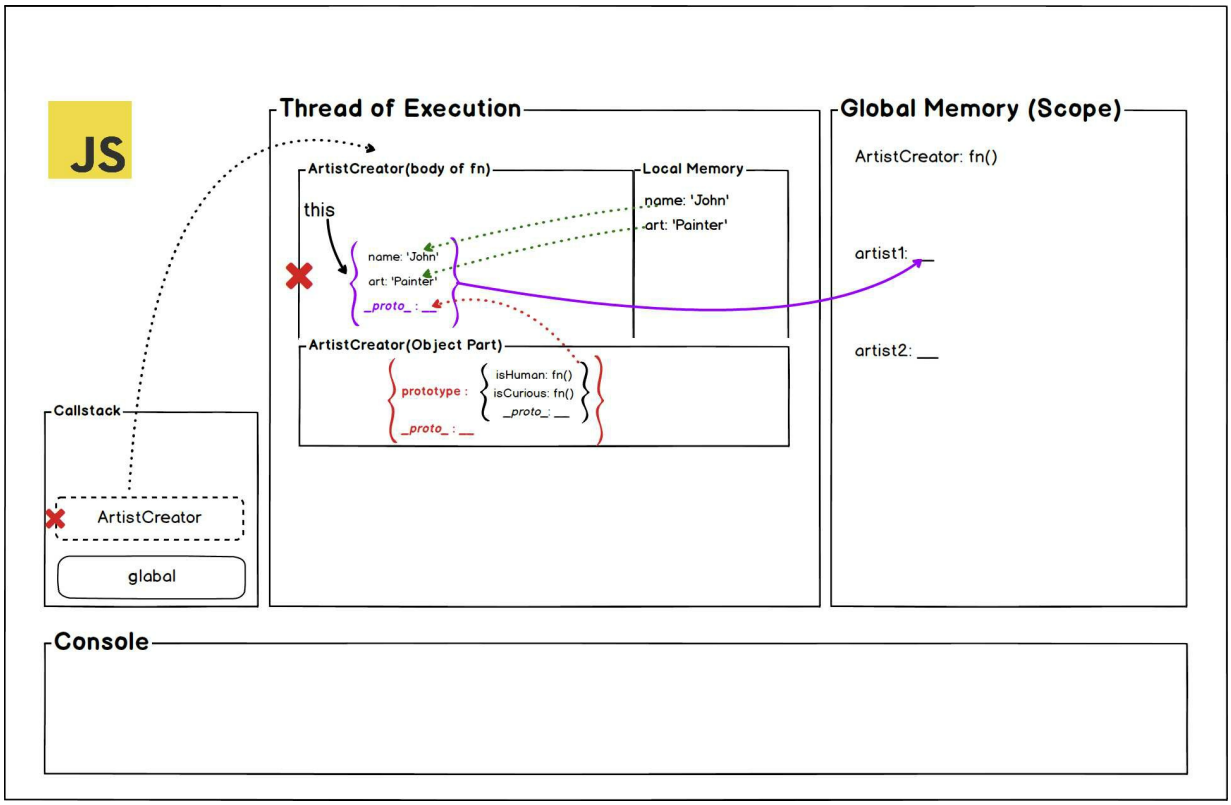


- Every function object has 2 parts as we know, one its function body, and the second is the object part.
- In the below diagram as you can see in **ArtistCreator**'s object part has a prototype and __proto__ object.
- In our code, we added two functions to the prototype object which are **isHuman** and **isCurious**.
- Every object will always have __proto__ property available by default. **That is why the prototype object also has __proto__ property as well**(where this property will point is unknown, but let's just focus first on our code execution).

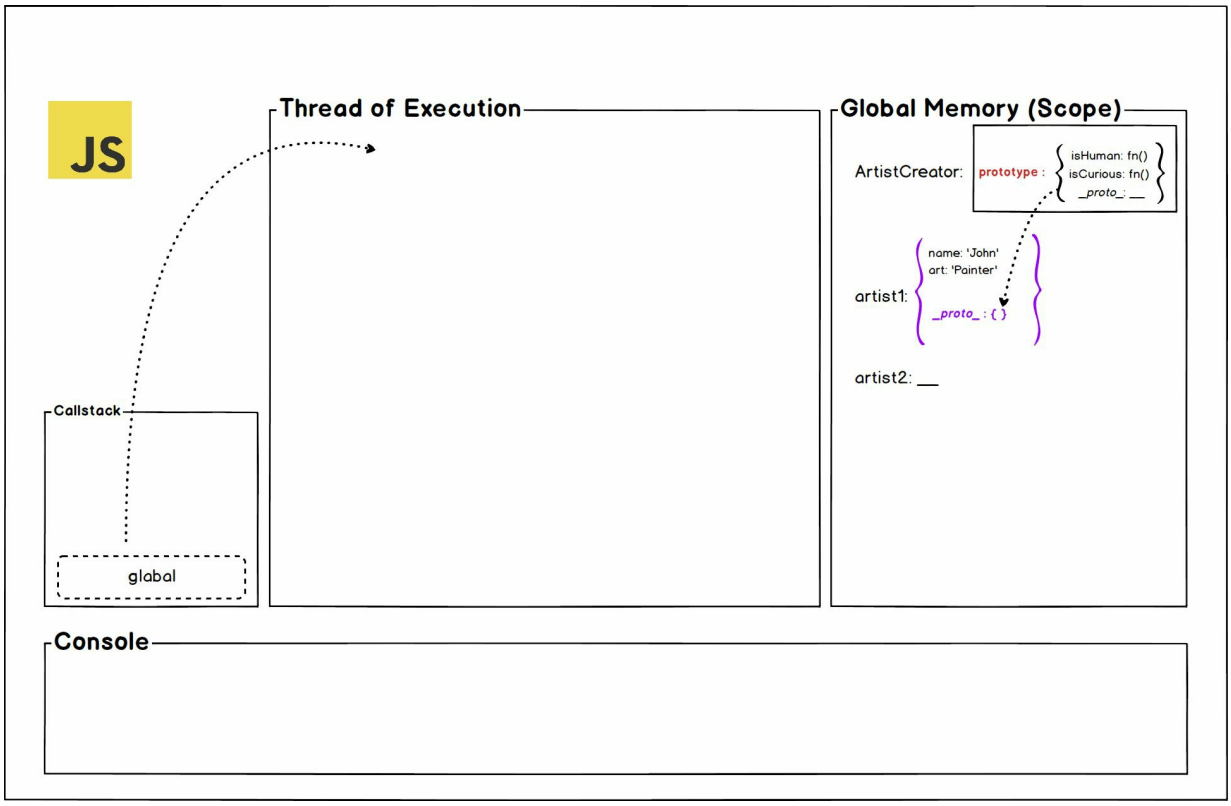


- The **prototype** object of function is referring to the __proto__ property of a newly created object.
- Now, we know the **new** keyword helps us to set the reference of the prototype object to __proto__ property of the newly created object. see below.
- One more thing, we have cleared function execution but the object part of the function remains in global memory. Here **we are not copying objects**, but we are just setting the references.

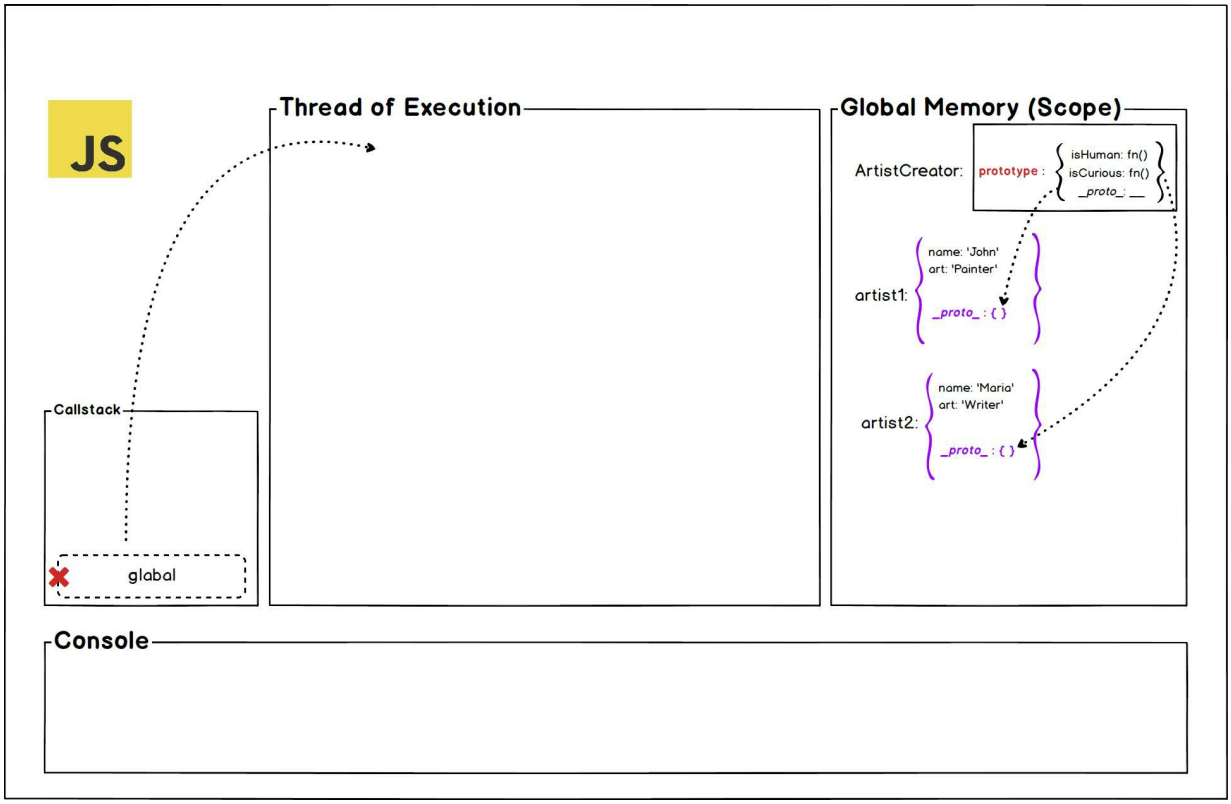

JS
Thread of Execution
Global Memory (Scope)
ArtistCreator: fn()
artist1: __
artist2: __
ArtistCreator(body of fn)
Local Memory
this
name: 'John'
art: 'Painter'
proto : __
ArtistCreator(Object Part)
prototype :
isHuman: fn()
isCurious: fn()
proto: __
proto : __
Callstack
ArtistCreator
global
Console

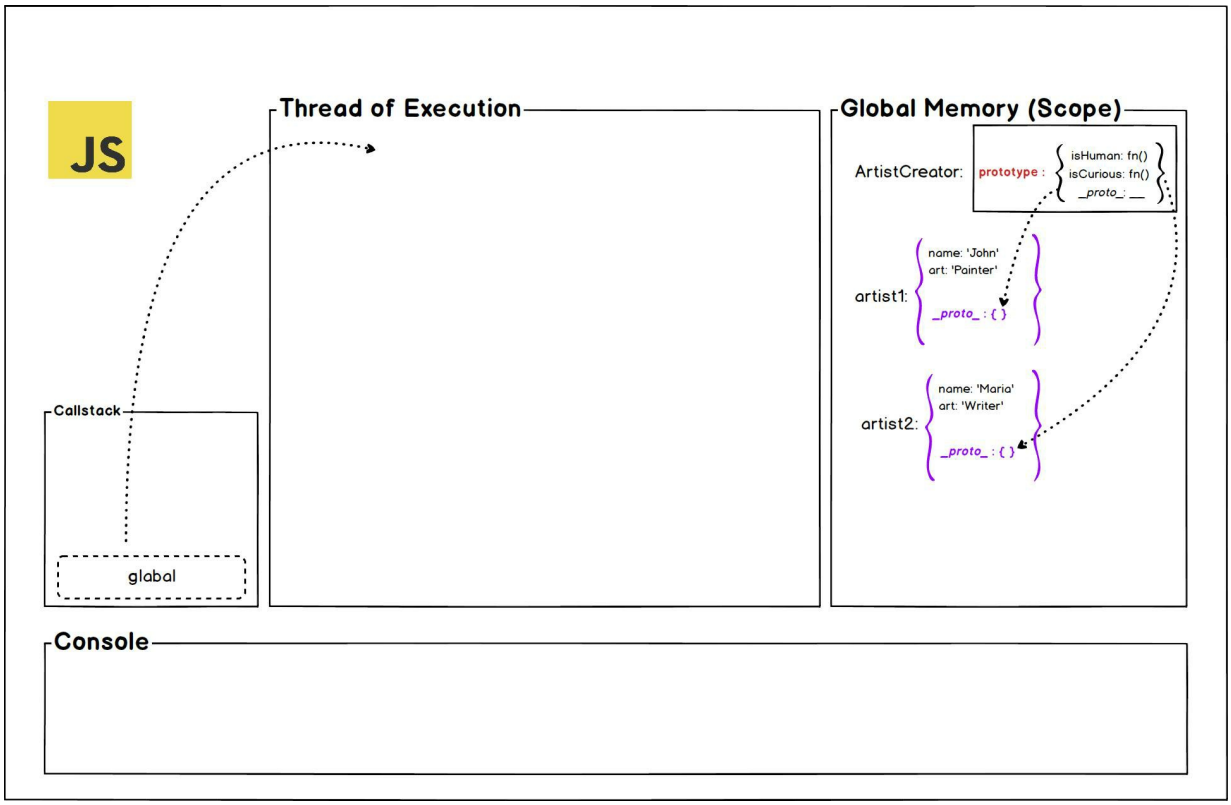

JS
Thread of Execution
Global Memory (Scope)
ArtistCreator: fn()
artist1: __
artist2: __
ArtistCreator(body of fn)
Local Memory
this
name: 'John'
art: 'Painter'
proto : __
ArtistCreator(Object Part)
prototype :
isHuman: fn()
isCurious: fn()
proto: __
proto : __
Callstack
ArtistCreator
global
Console

- Never forget, function execution has been done, but a memory for that function will not wipe out from memory. That is why ArtistCreator function with prototype object will remain in memory.

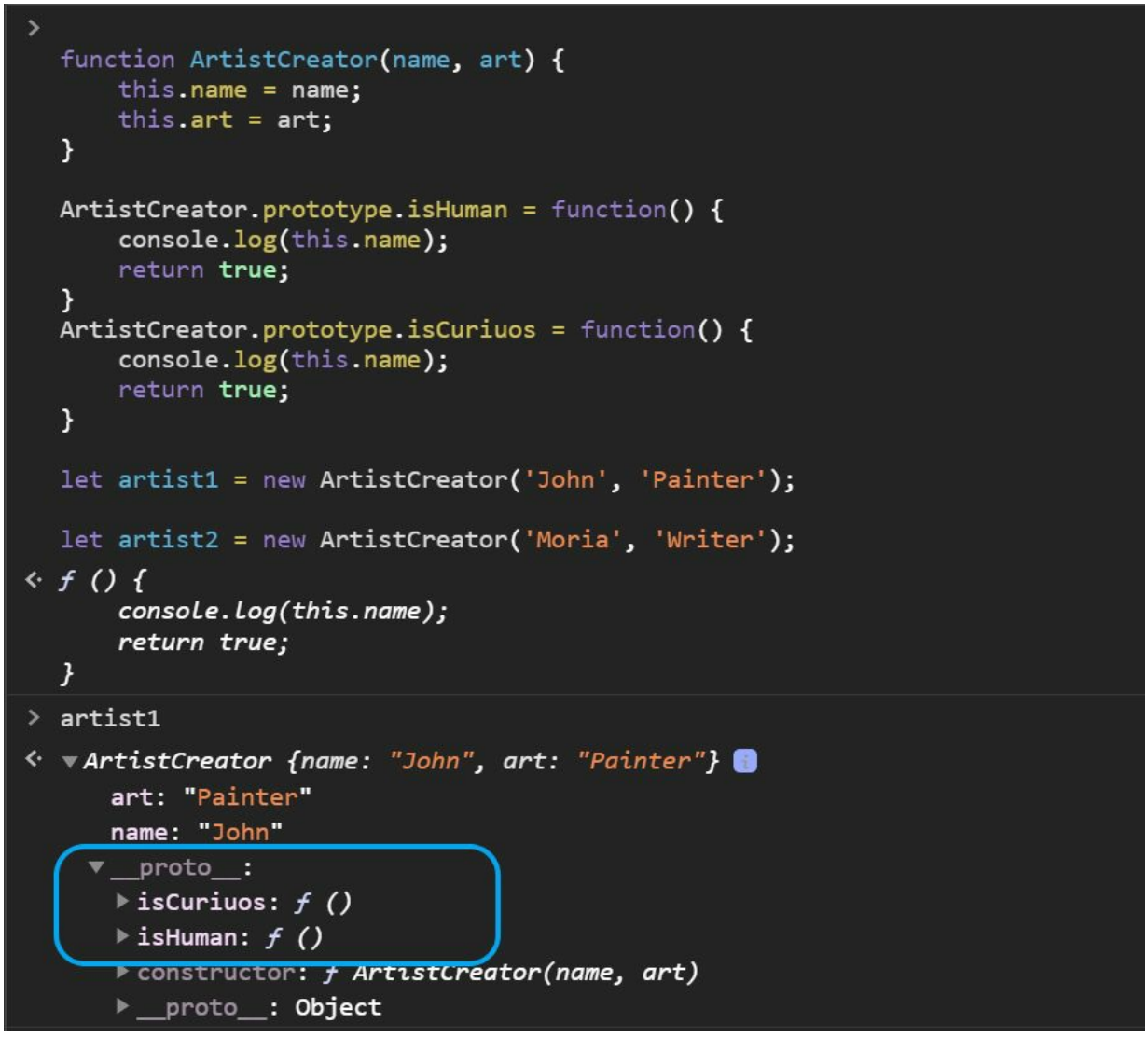


- There you have it, we are now able to create objects from function, and some common functionalities also available by prototype object as well.
- Now, the same thing happens to **artist2** as well, so I am not going to draw that part(you can visualize your own now😎).


JS
Thread of Execution
Global Memory (Scope)
ArtistCreator:
prototype :
isHuman: fn()
isCurious: fn()
__proto__: __
artist1:
name: 'John'
art: 'Painter'
__proto__ : { }
artist2:
name: 'Maria'
art: 'Writer'
__proto__ : { }
Callstack
global
Console


JS
Thread of Execution
Global Memory (Scope)
ArtistCreator:
prototype :
isHuman: fn()
isCurious: fn()
__proto__: __
artist1:
name: 'John'
art: 'Painter'
__proto__ : { }
artist2:
name: 'Maria'
art: 'Writer'
__proto__ : { }
Callstack
global
Console

- We are creating objects using the function, that is why we call it **“function constructor”.** so**,** here ArtistCreator is the constructor function.
- Before we close the above example, let’s just output in the browser console.



- So far so good, but ES6+ has new syntactic sugar available to us which does the same thing as shown in the above code. It’s just **trying to mimic and simplify the syntax.** Under the hood, working remains the

same. Let’s just see the equivalent code for the above example.

```
255
256   class ArtistCreator {
257       constructor(name, art) {
258           this.name = name;
259           this.art = art;
260       }
261       isHuman() {
262           console.log(this.name);
263           return true;
264       }
265       isCuriuos() {
266           console.log(this.name);
267           return true;
268       }
269   }
270
271   let artist1 = new ArtistCreator('John', 'Painter');
272   let artist2 = new ArtistCreator('Maria', 'Writer');
273
```

- The **class** keyword is introduced in ES6+ to mimic our previous work.
- Not too much fancy, simply our prototype methods are defined directly inside class and constructor works as a function body.

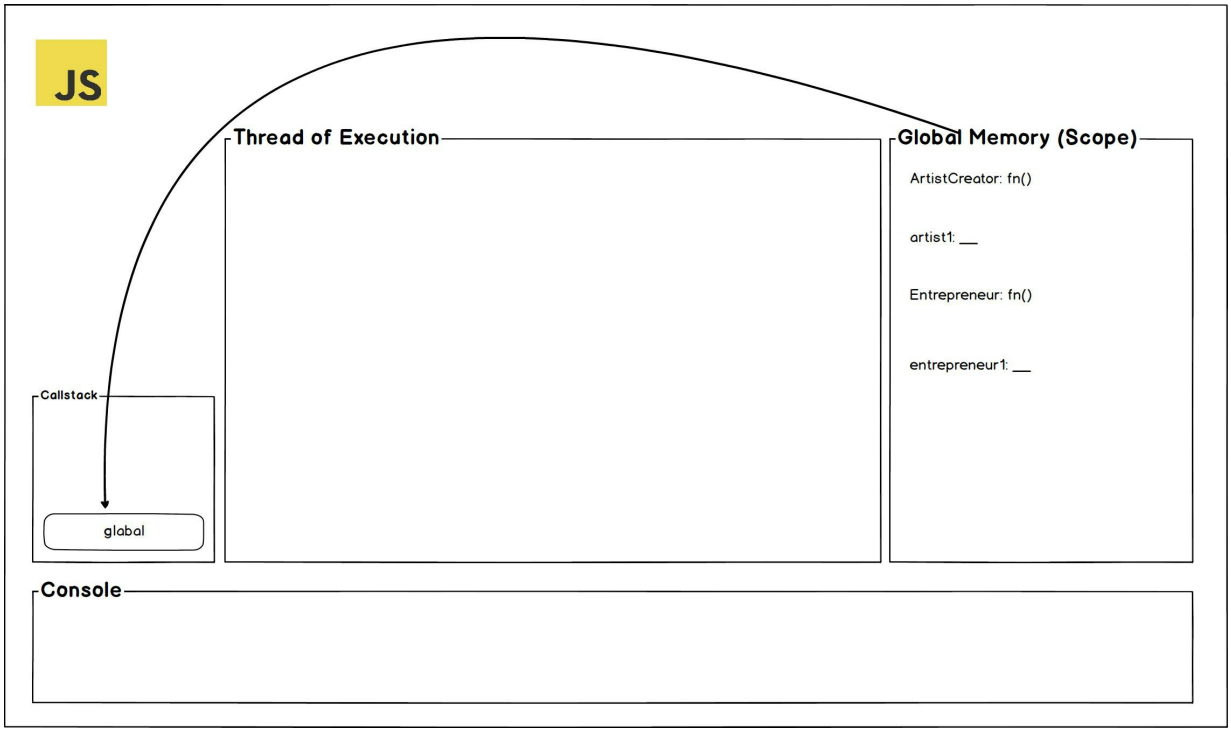
We have learned how to create dynamic and custom objects from the function constructor. Now, what about extending our functionalities and using the existing function constructor to create another function constructor such that it helps us inherit others’ features as well. let’s see with example😁.Initially we will take ***non-class*** *version exmple and later we will create equivaent class version also.*

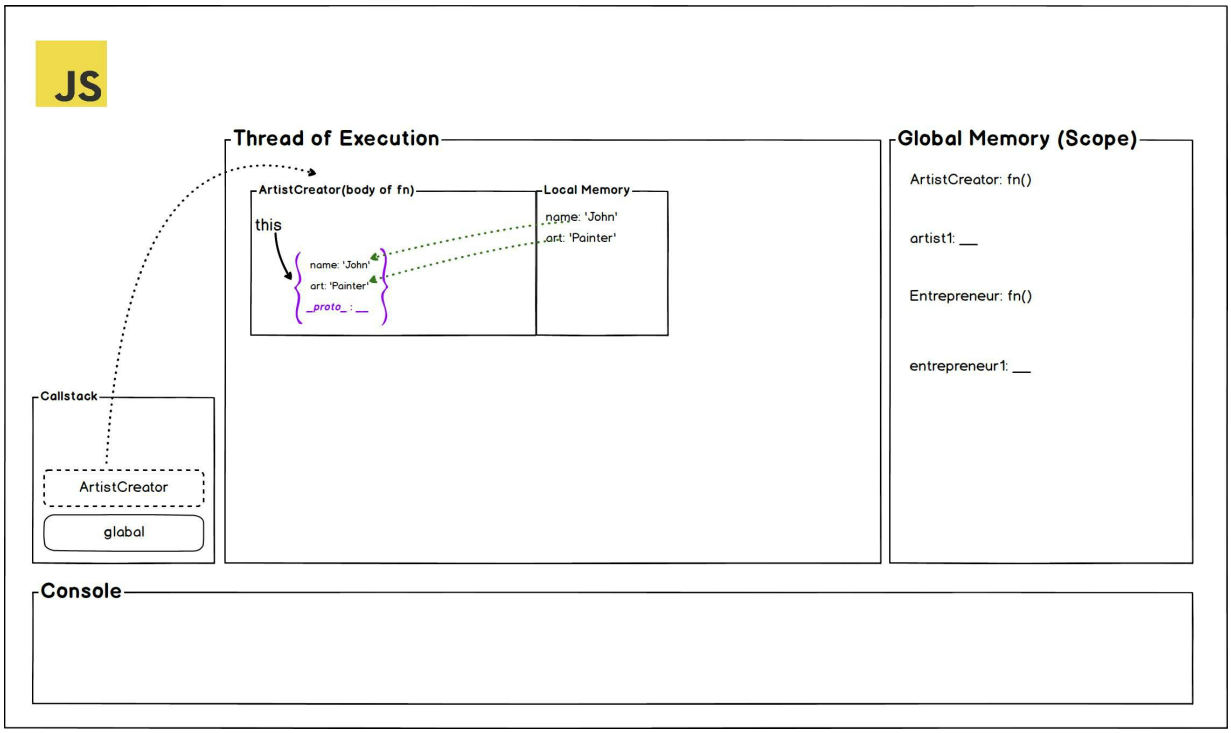
```
// ArtistCreator contructor
function ArtistCreator(name, art) {
    this.name = name;
    this.art = art;
}

ArtistCreator.prototype.isHuman = function() {
    console.log(this.name);
    return true;
}
ArtistCreator.prototype.isCuriuos = function() {
    console.log(this.name);
    return true;
}

let artist1 = new ArtistCreator('John', 'Painter');

// Entrepreneur contructor
function Entrepreneur(name, art, communicationSkill) {
    ArtistCreator.call(this, name, art);
    this.communicationSkill = communicationSkill;
}

Entrepreneur.prototype = Object.create(ArtistCreator.prototype);

Entrepreneur.prototype.hasPatience = function() {
    console.log(this.name);
    return true;
}

let entrepreneur1 = new Entrepreneur('Maria', 'Writer', 'Good');
```

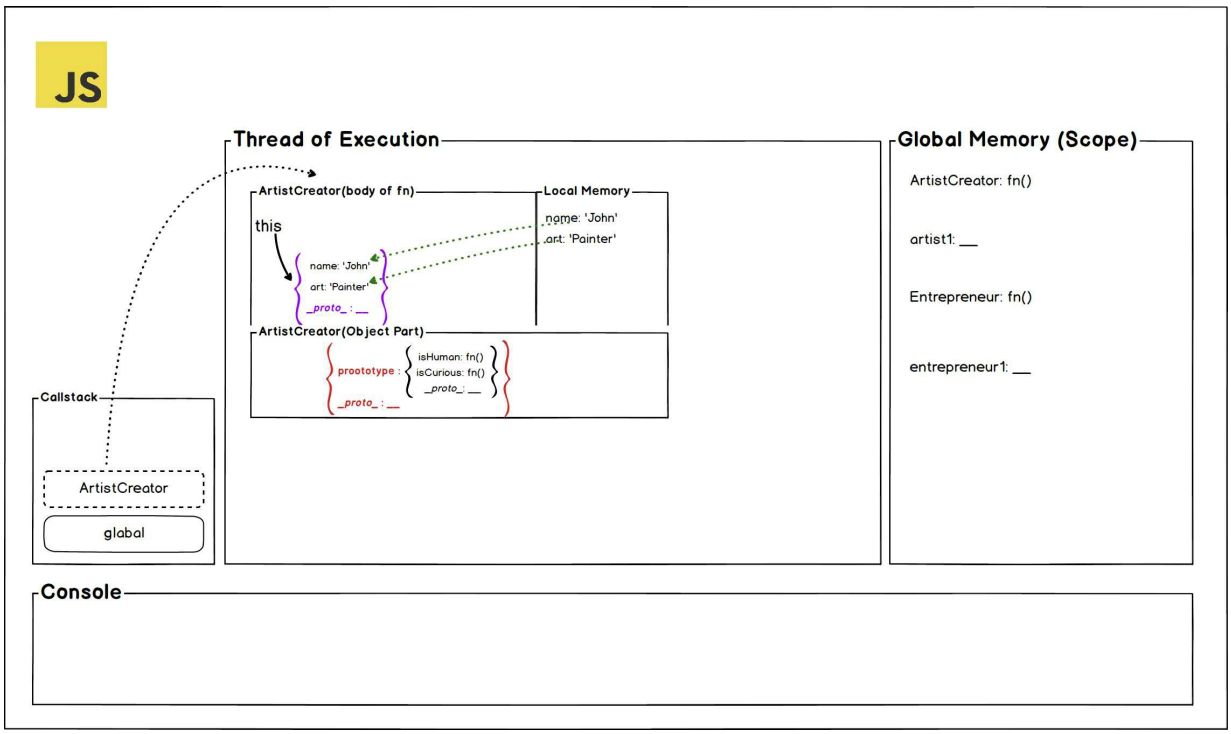
- Don’t worry if you can’t understand the above code, we will go piece by piece how it works. The only thing to note in the above example is that we can inherit **ArtistCreator’s** prototype functions into our newly created **Entrepreneur’s** function constructor. let’s start visualizing😉.

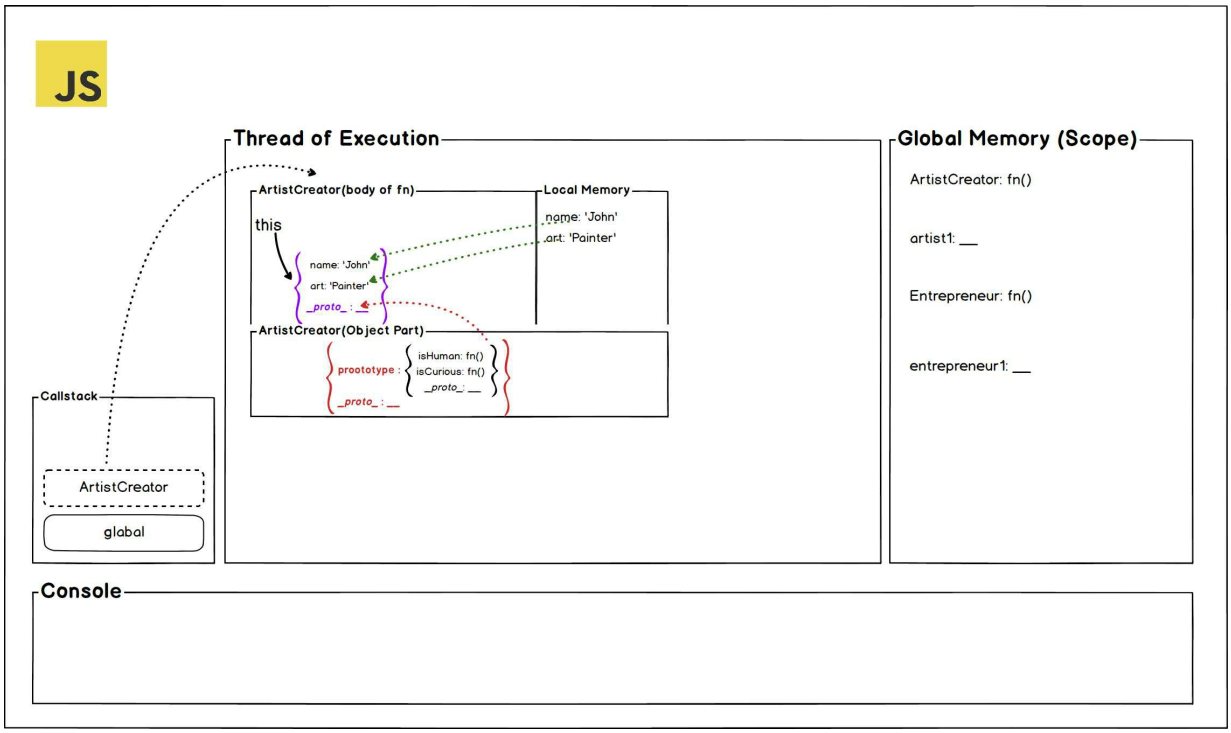
```
274
275   // ArtistCreator contructor
276   function ArtistCreator(name, art) {
277       this.name = name;
278       this.art = art;
279   }
280
281   ArtistCreator.prototype.isHuman = function() {
282       console.log(this.name);
283       return true;
284   }
285   ArtistCreator.prototype.isCuriuos = function() {
286       console.log(this.name);
287       return true;
288   }
289
290   let artist1 = new ArtistCreator('John', 'Painter');
291
```

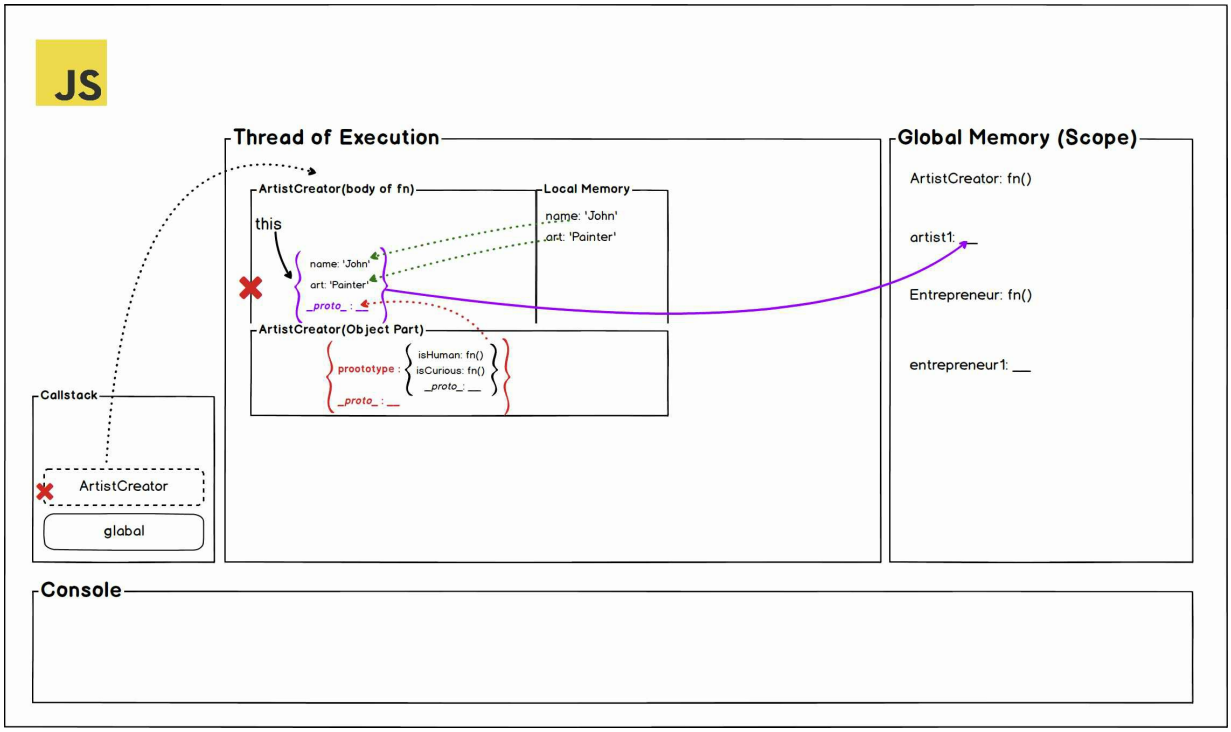
- We are first going to visualize the above line. This will be similar to the previous example we just show.

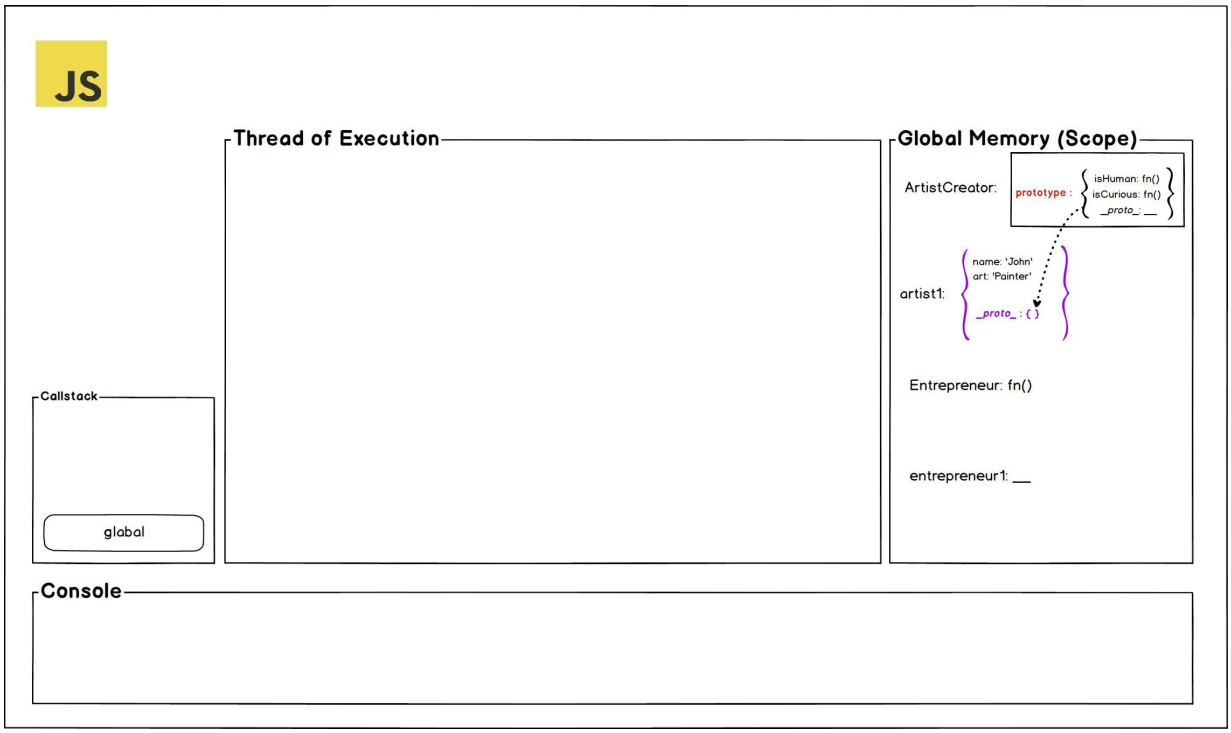

JS
Thread of Execution
Global Memory (Scope)
ArtistCreator: fn()
artist1: __
Entrepreneur: fn()
entrepreneur1: __
Callstack
global
Console


JS
Thread of Execution
ArtistCreator(body of fn)
Local Memory
this
name: 'John'
art: 'Painter'
proto : __
name: 'John'
art: 'Painter'
Global Memory (Scope)
ArtistCreator: fn()
artist1: __
Entrepreneur: fn()
entrepreneur1: __
Callstack
ArtistCreator
global
Console


JS
Thread of Execution
Global Memory (Scope)
ArtistCreator(body of fn)
Local Memory
this
name: 'John'
art: 'Painter'
proto : __
ArtistCreator(Object Part)
proototype :
isHuman: fn()
isCurious: fn()
proto: __
proto : __
Callstack
ArtistCreator
global
ArtistCreator: fn()
artist1: __
Entrepreneur: fn()
entrepreneur1: __
Console


JS
Thread of Execution
Global Memory (Scope)
ArtistCreator(body of fn)
Local Memory
this
name: 'John'
art: 'Painter'
proto : __
ArtistCreator(Object Part)
proototype :
isHuman: fn()
isCurious: fn()
proto: __
proto : __
Callstack
ArtistCreator
global
ArtistCreator: fn()
artist1: __
Entrepreneur: fn()
entrepreneur1: __
Console



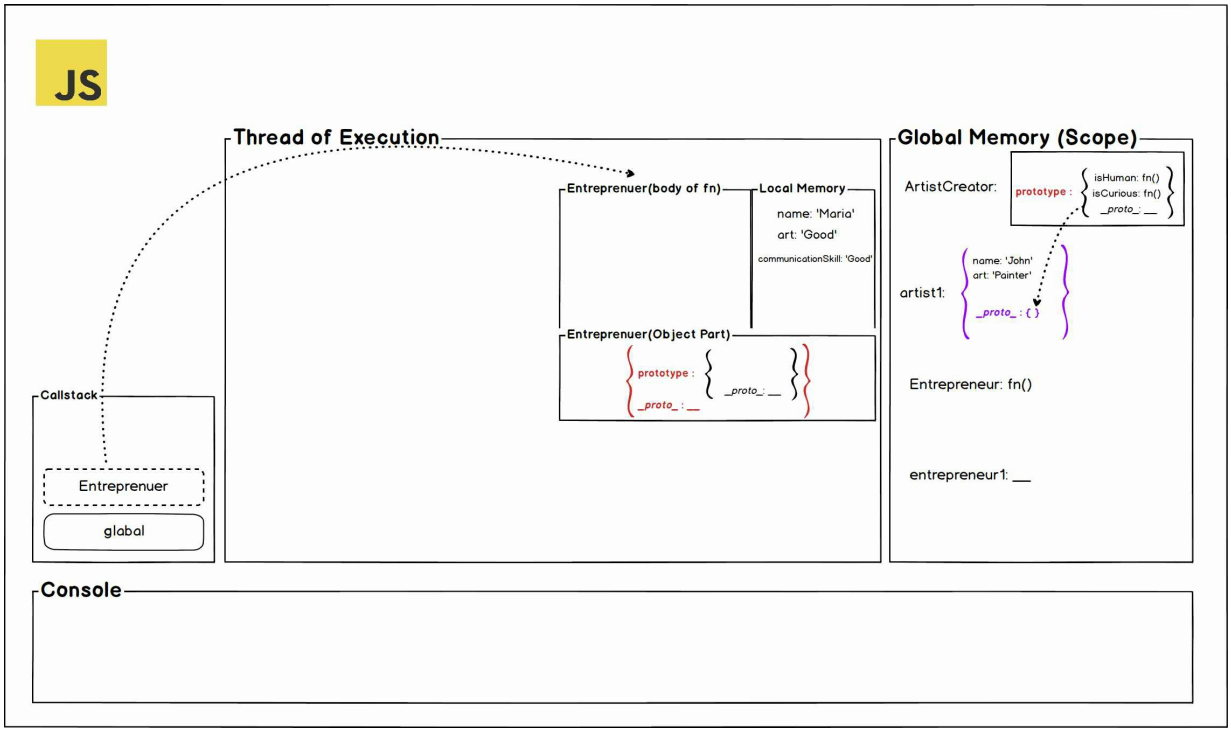


- So far it was the same as the previous example's visualization. now, let's see what happens when we run the below code.

- Also, I am here skipping the execution of line no. 298 to 303 because I don’t want to confuse you with upcoming diagrams. we are actually going to see a diagram of line no. 298 to 303 after when finishing execution on the below-highlighted line.

```
291
292  // Entrepreneur contructor
293  function Entrepreneur(name, art, communicationSkill) {
294      ArtistCreator.call(this, name, art);
295      this.communicationSkill = communicationSkill;
296  }
297
298  Entrepreneur.prototype = Object.create(ArtistCreator.prototype);
299
300  Entrepreneur.prototype.hasPatience = function() {
301      console.log(this.name);
302      return true;
303  }
304
305  let entrepreneur1 = new Entrepreneur('Maria', 'Writer', 'Good');
306
```

- Don’t forget ArtistCreator’s object part will remain in memory even after execution is finished and when we call ArtistCreator again then we will use reference stored from memory instead of creating a copy of it.




```
292  // Entrepreneur contructor
293  function Entrepreneur(name, art, communicationSkill) {
294      ArtistCreator.call(this, name, art);
295      this.communicationSkill = communicationSkill;
296  }
297
298  Entrepreneur.prototype = Object.create(ArtistCreator.prototype);
299
300  Entrepreneur.prototype.hasPatience = function() {
301      console.log(this.name);
302      return true;
303  }
304
305  let entrepreneur1 = new Entrepreneur('Maria', 'Writer', 'Good');
306
```

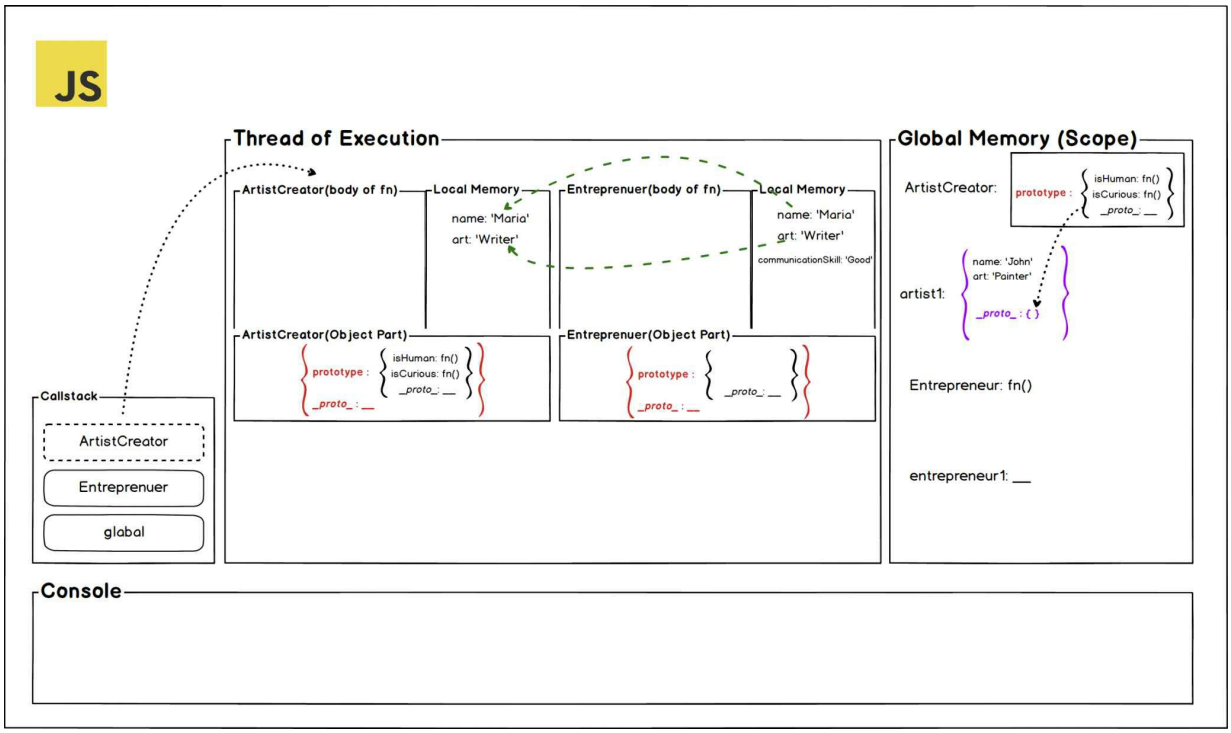
- We are calling a function with the **“call”** keyword and passing **this** is to as the first argument.

```
The call() method calls a function with a given this value and arguments provided individually.
```

- Because we have used a **new** keyword in front of the **Entrepreneur** function, we are going to create a new empty object and **this** keyword will point to that**.** However, Entrepreneur internally calling ArtistCreator with **call()** method and passing the newly created object(this keyword currently pointing to the newly created empty object!).

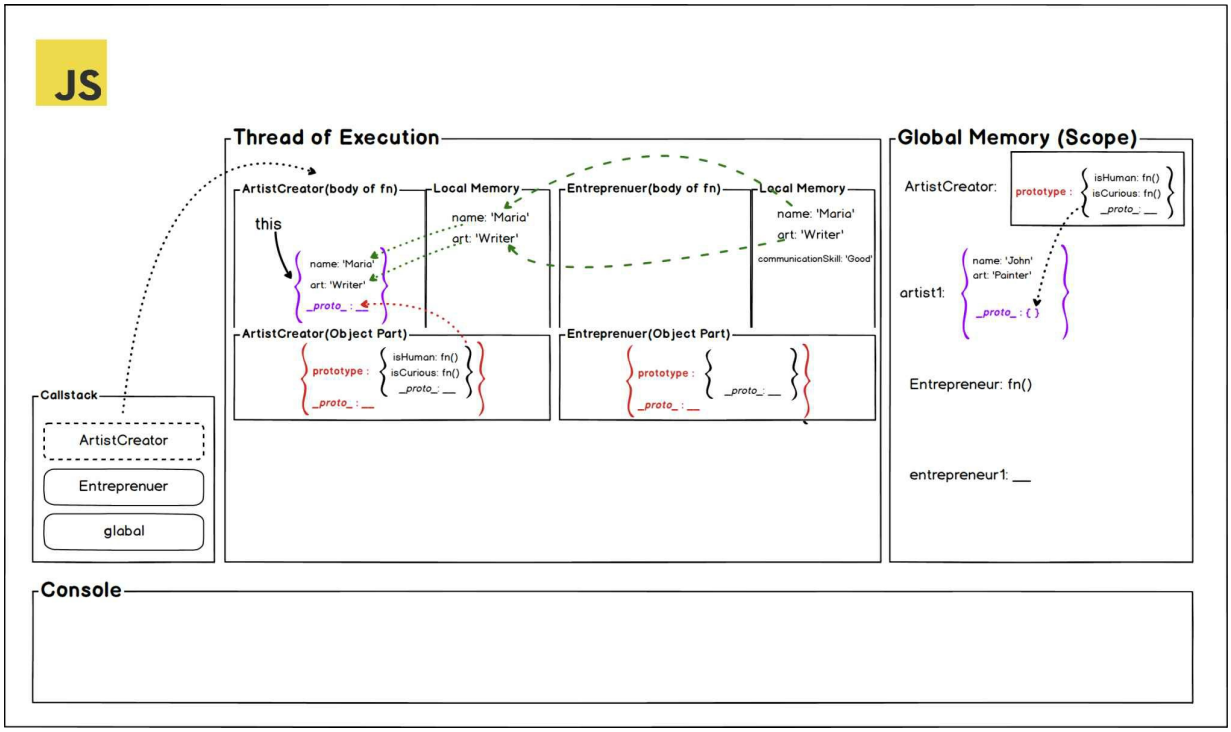
```
274
275   ArtistCreator contructor
276   function ArtistCreator(name, art) {
277       this.name = name;
278       this.art = art;
279   }
280
```

- Here we are just adding properties to a newly created object.

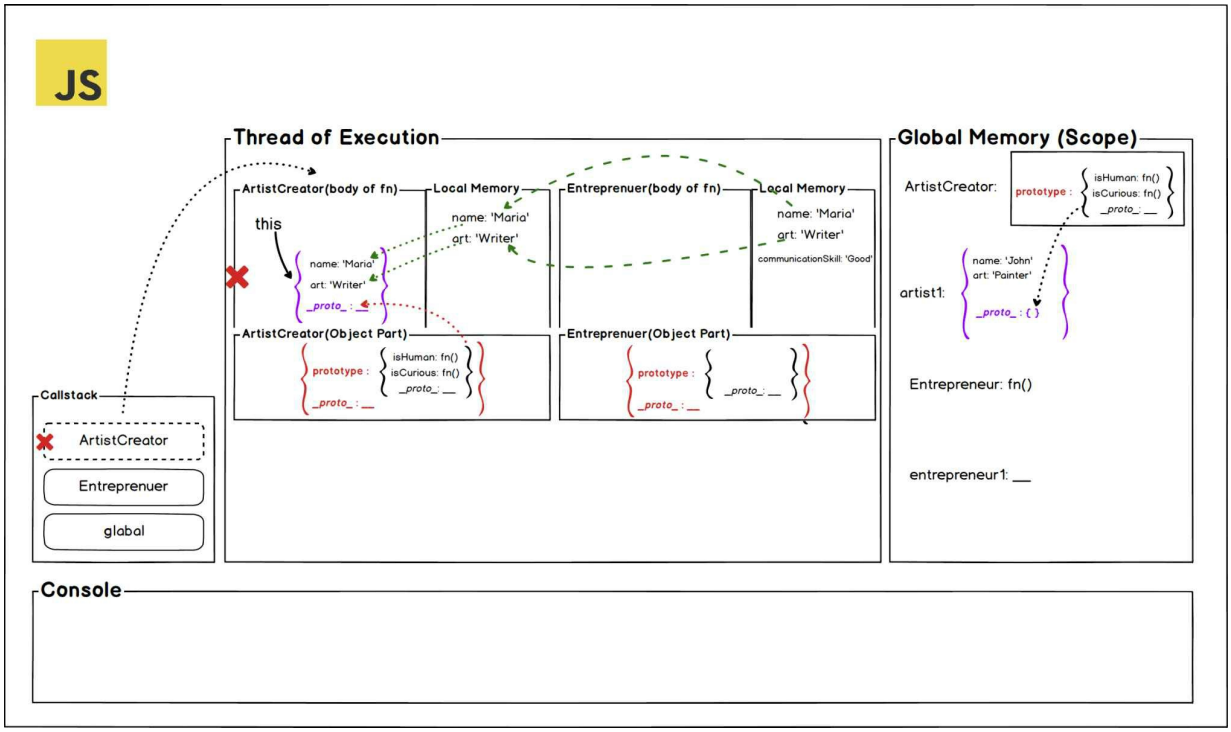


- The arguments are actually made available to the **ArtistCreator**

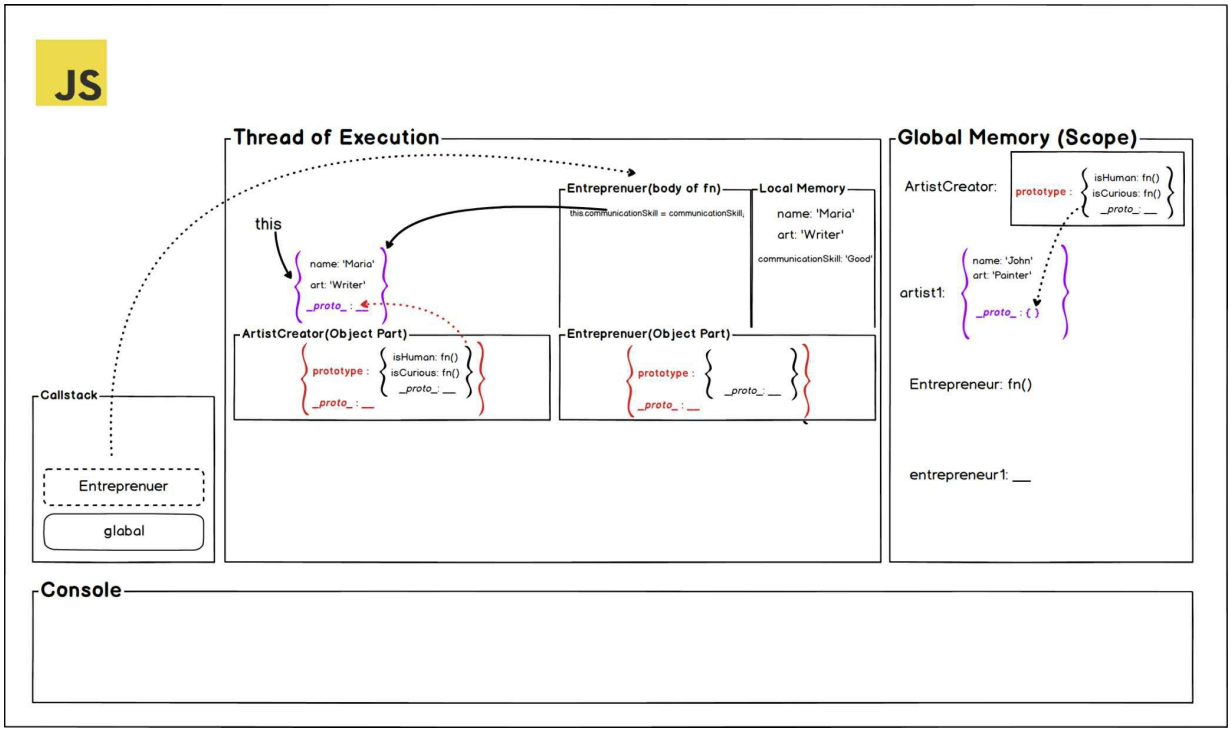
function via reference, where the **Entrepreneur** function is providing those values😊. Just follow the diagram and you will get it.



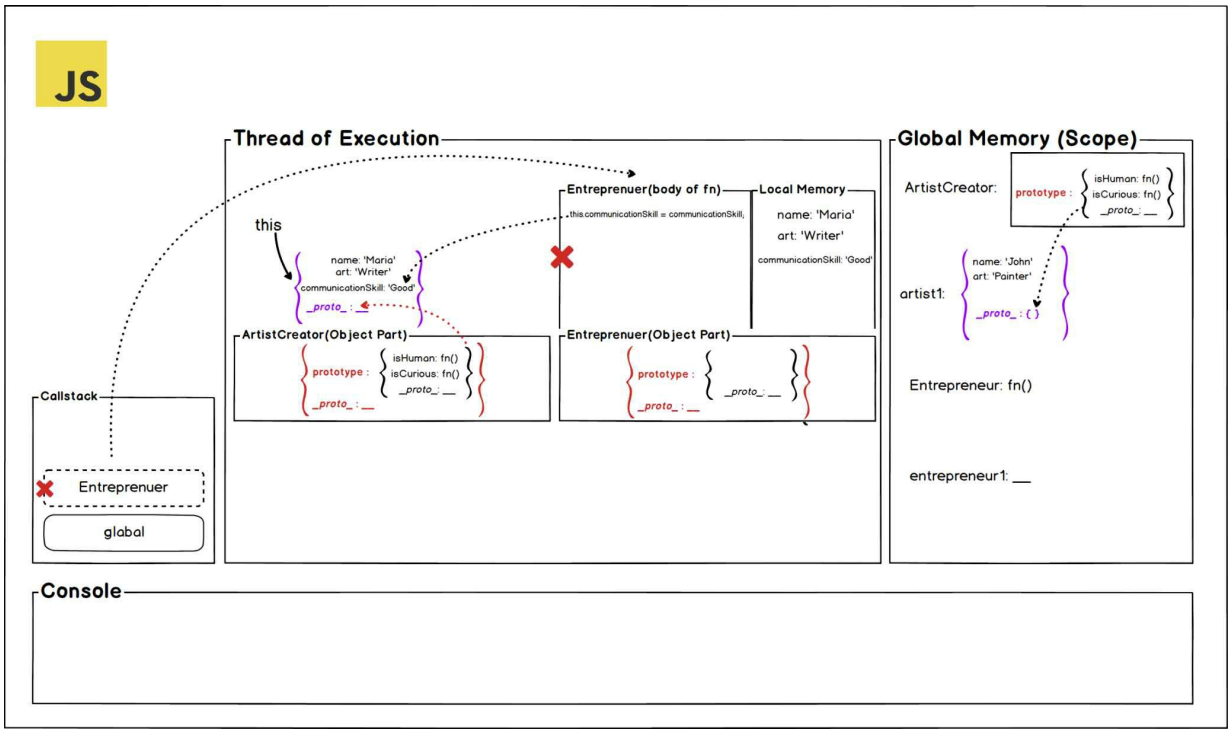
- By default you know, the newly created object(**this**) will have a reference of the prototype object where the object is born.
- In our case, we used ArtistCreator as a mother for the newly created object. so, **the newly created object's __proto__ will point to the prototype object of ArtistCreator**.



- After finishing the execution of the **ArtistCreator** function, we will continue our execution with our **Entrepruer** function execution and here we are adding one more property **“communicationSkill”** to the newly created(this) object.
- I am keeping the object part of functions here because it’ll help us to understand better.

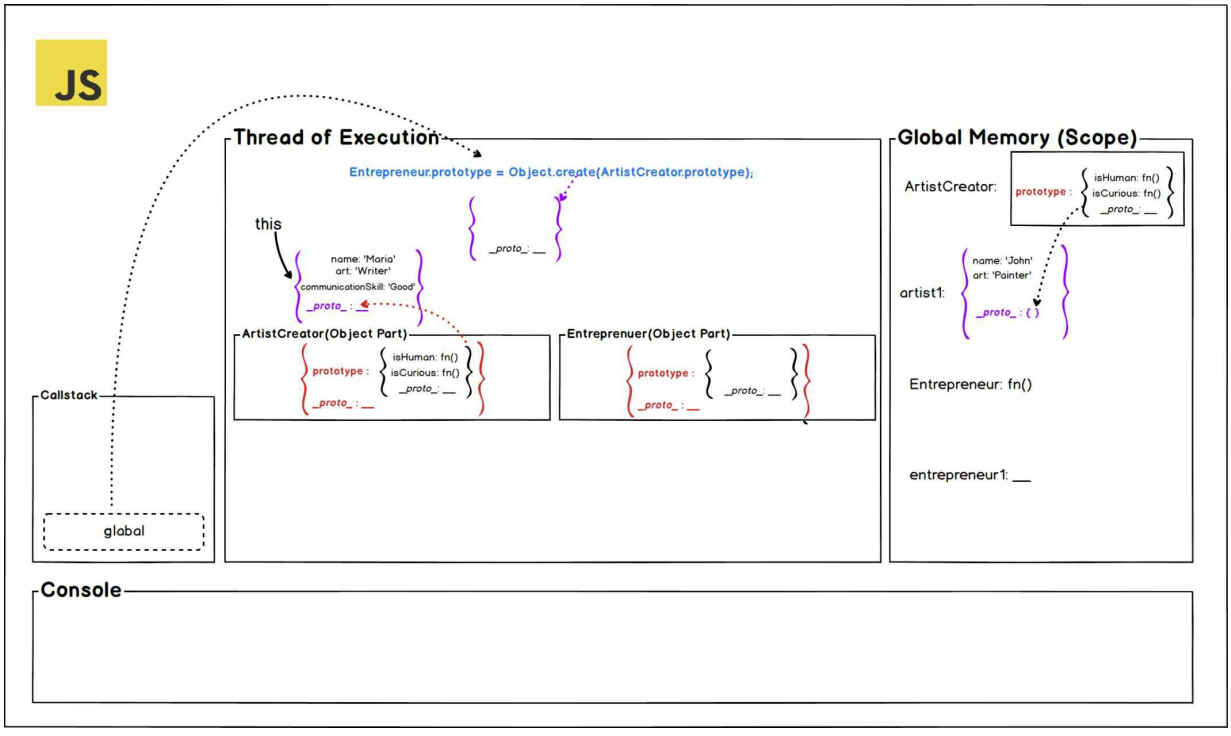


- Also, as you can see we are now done with our Entrepruener function execution as well.

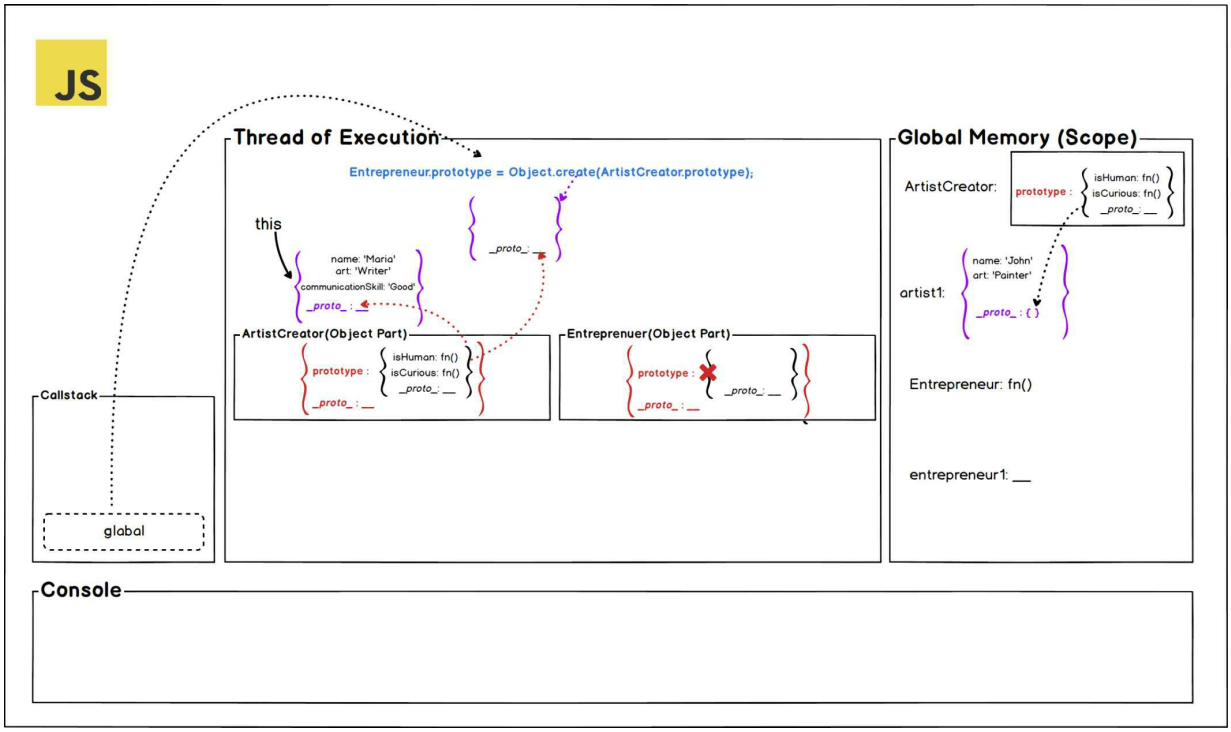


- Now, the important part has arrived. remember I told you to ignore line 298 to 303. let’s see what happens to them behind the scenes when we execute it(actually execution already is finished but I am showing right now because it will be easy to visualize now).

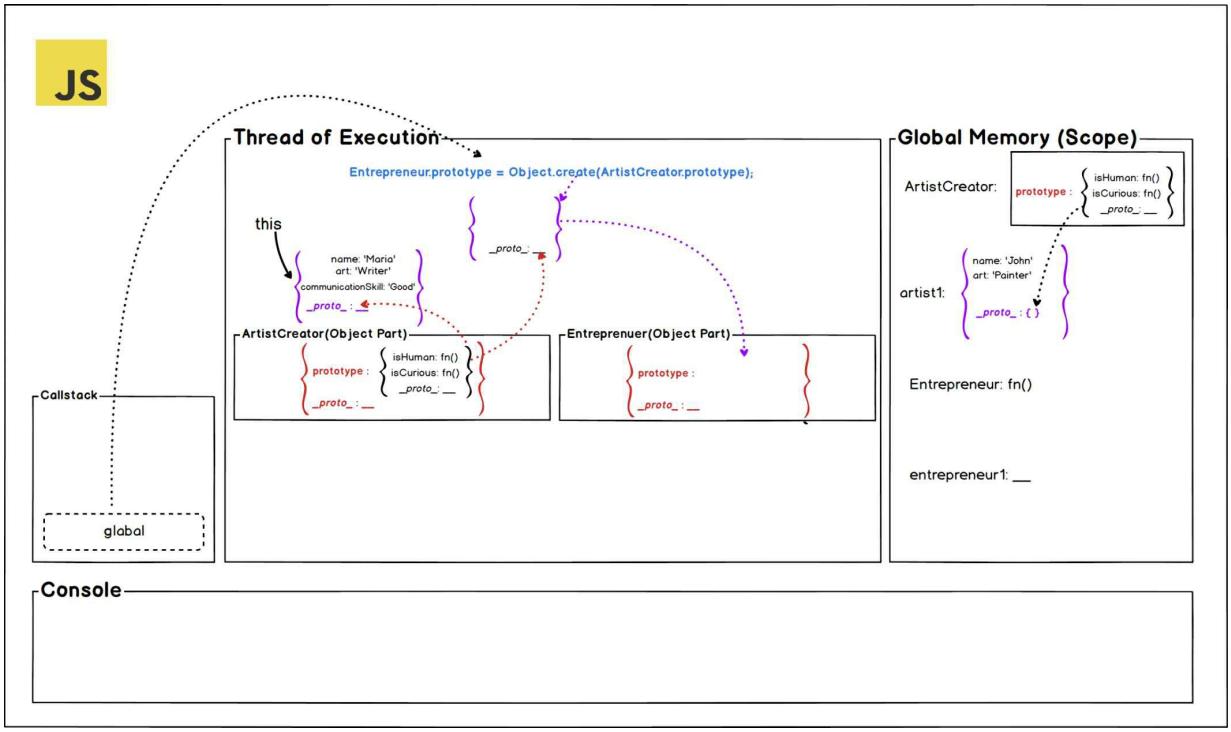
```
297
298  Entrepreneur.prototype = Object.create(ArtistCreator.prototype);
299
300  Entrepreneur.prototype.hasPatience = function() {
301      console.log(this.name);
302      return true;
303  }
304
```



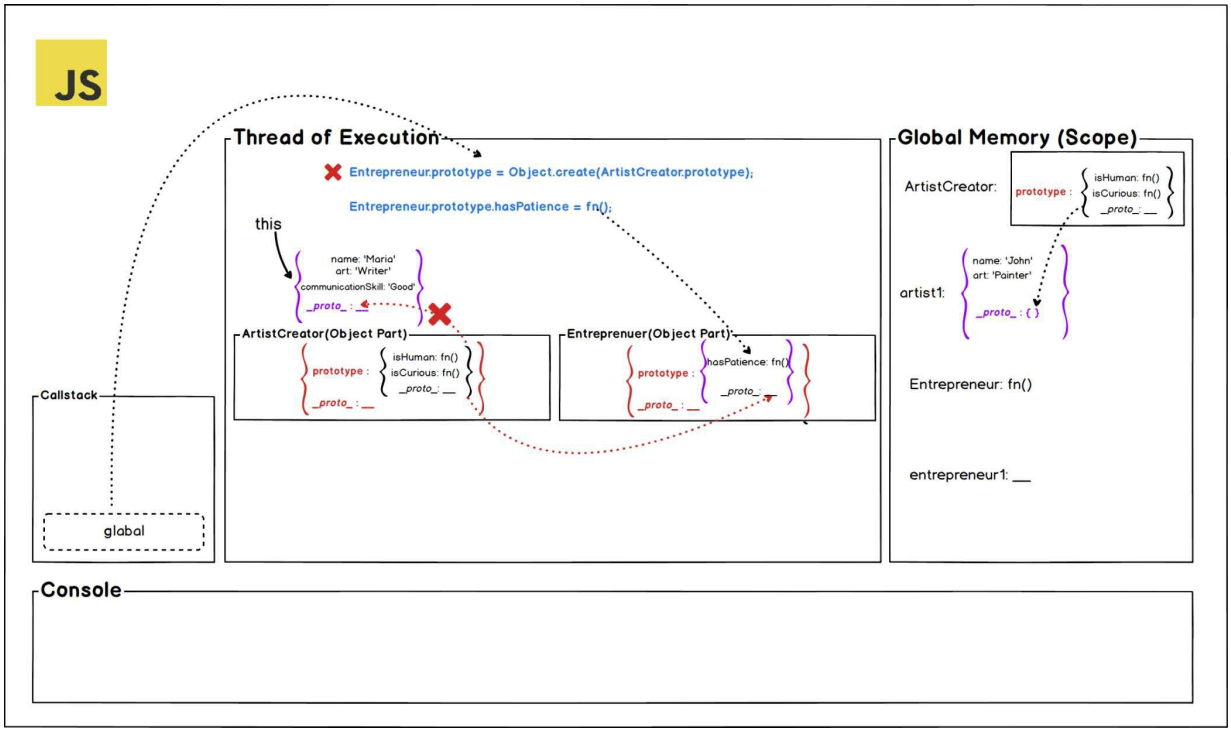
- As we know, **Object.create() will create an empty object**, and __proto__ of just created empty object will point to whatever we pass to the argument of **create() method**.

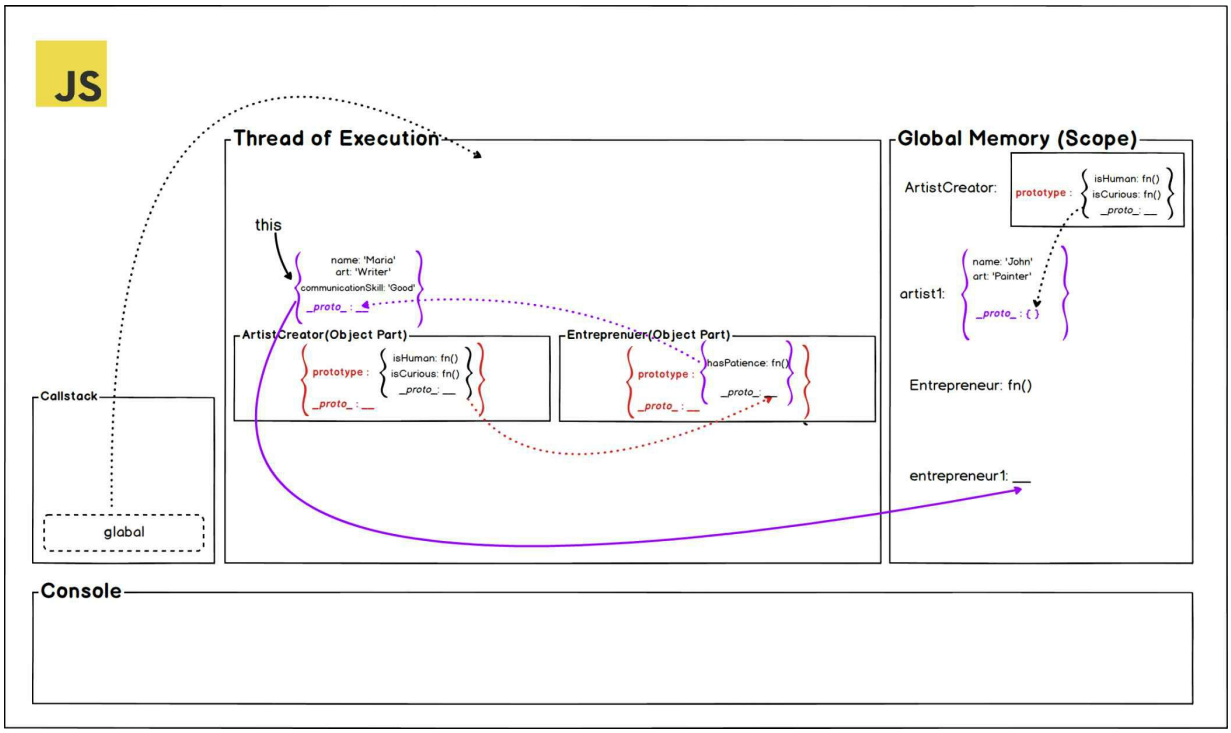


- Because we are reassigning **Entrepreneur.prototype,** the old prototype object of the Entrepreneur will be destroyed from memory. whatever we got back from Object.create() method call will assign to Entrepreneur.prototype😊.



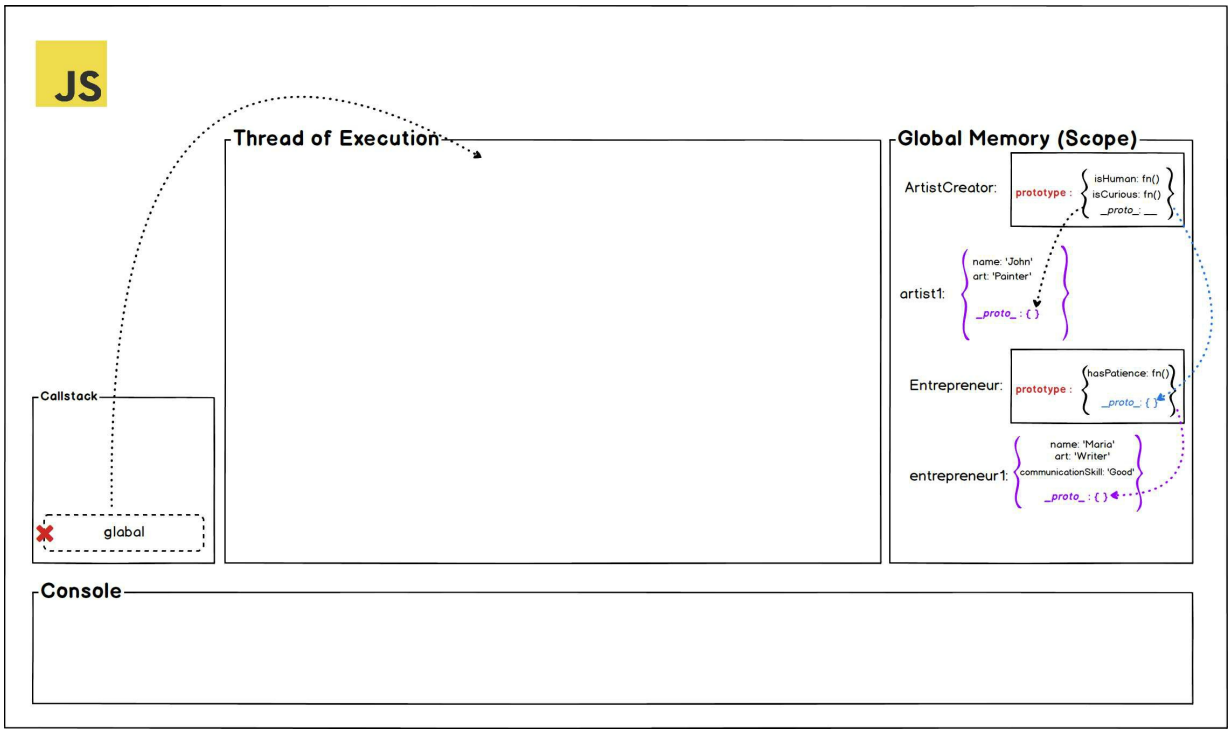
- We are adding one property to **Entrepreneur.prototype** object as well which is the **hasPatience** function.
- we are passing **ArtistCreator.prototype** object to **Object.create()** method. So, the prototype object of ArtistCreator will point to __proto__ of the newly created empty object.

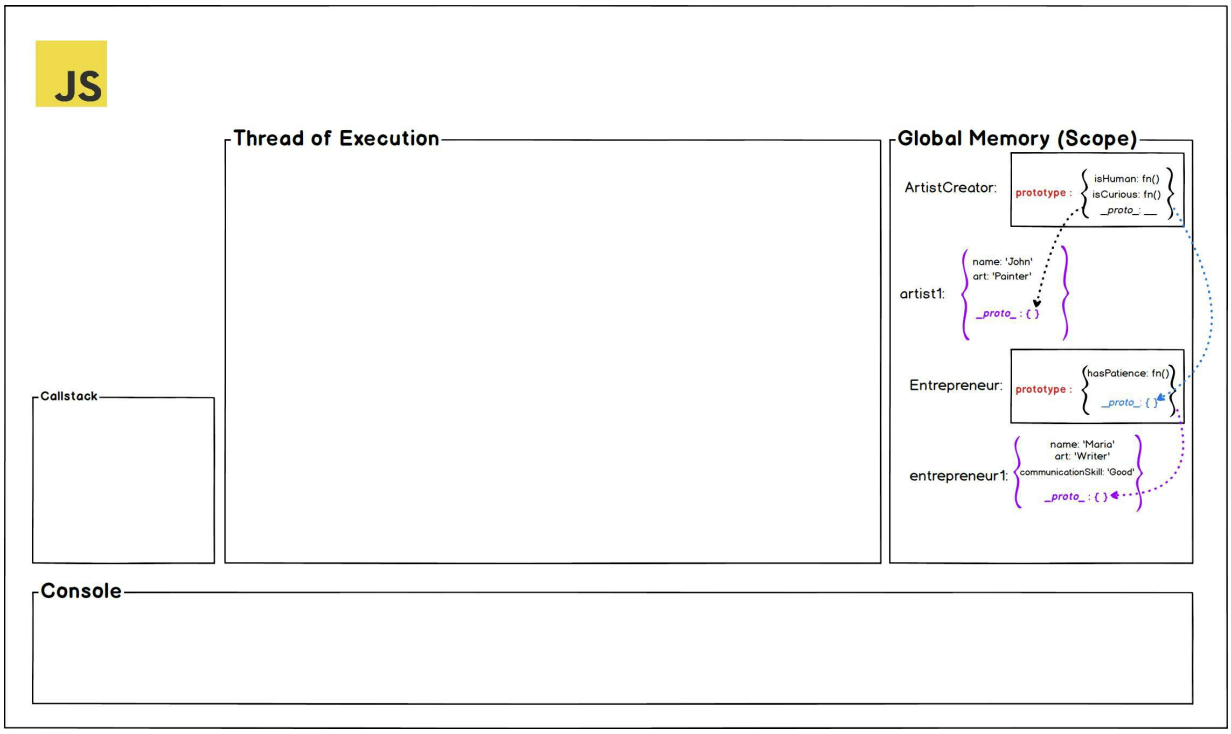




- As we know, prototype objects of functions remain in memory, that's the reason why it's shown in the diagram.

- see how __proto__ and it’s linking works😎.






```
Woooh, we finally completed our visualization(if you find it a little confusing
then just go through it again😊, the first time it always seems a little too much
```

```
to digest.)
```

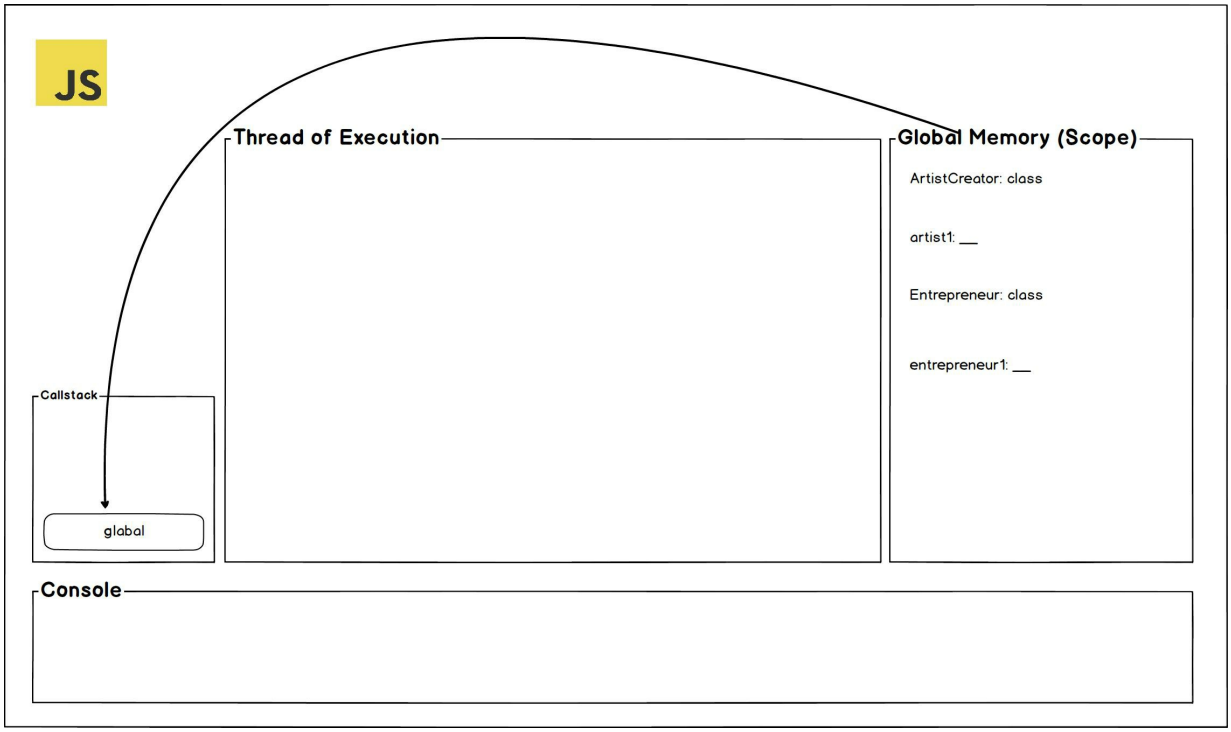
- Let’s just see the output in our console. compare our above diagram with the below output you will get it😊.

What we have achieved then.

We have learned one big important stuff from the previous example. We are literally able to create dynamic objects and reuse the existing functionality of other objects and function(don’t call this inheritance 😂, here we are composing stuff not inheriting).

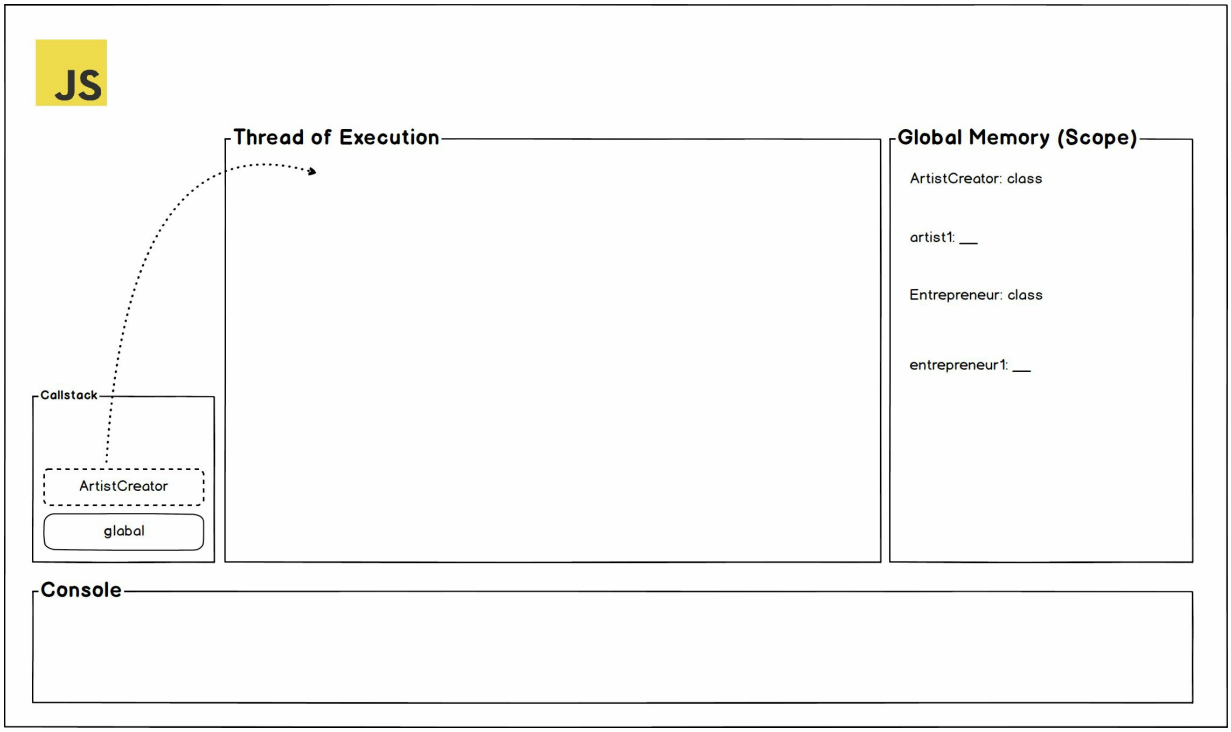
- Now, how about if ES6+ provides syntactic sugar of above. we actually do and have simpler syntax to achieve the same thing as above.
- Here, working is slightly different to achieve the same thing. let’s see.

```
307
308  class ArtistCreator {
309      constructor(name, art) {
310          this.name = name;
311          this.art = art;
312      }
313      isHuman() {
314          console.log(this.name);
315          return true;
316      }
317      isCuriuos() {
318          console.log(this.name);
319          return true;
320      }
321  }
322
323  let artist1 = new ArtistCreator('John', 'Painter');
324
325  class Entrepreneur extends ArtistCreator {
326      constructor(name, art, communicationSkill) {
327          super(name, art);
328          this.communicationSkill = communicationSkill;
329      }
330      hasPatience() {
331          console.log(this.name);
332          return true;
333      }
334  }
335
336  let entrepreneur1 = new Entrepreneur('Maria', 'Writer', 'Good');
337
```



- Let’s start with our simple class execution for the below line and visualize it.

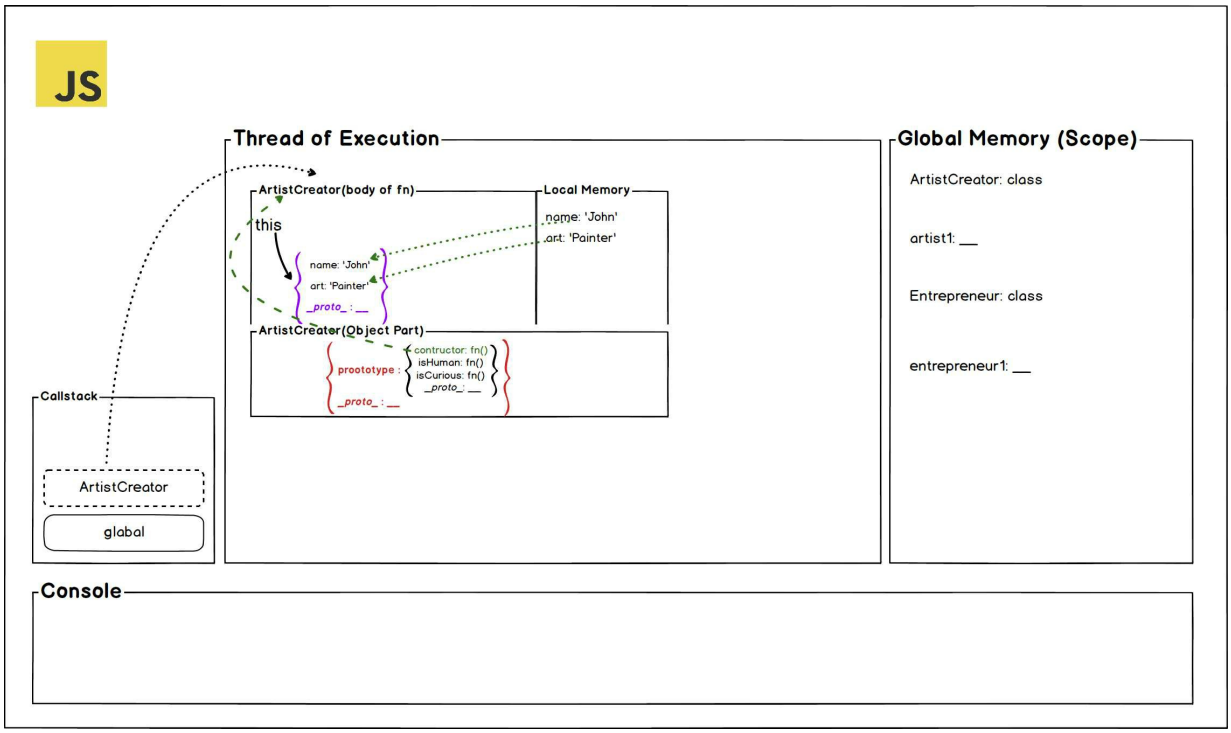
```
307
308  class ArtistCreator {
309      constructor(name, art) {
310          this.name = name;
311          this.art = art;
312      }
313      isHuman() {
314          console.log(this.name);
315          return true;
316      }
317      isCuriuos() {
318          console.log(this.name);
319          return true;
320      }
321  }
322
323  let artist1 = new ArtistCreator('John', 'Painter');
324
```



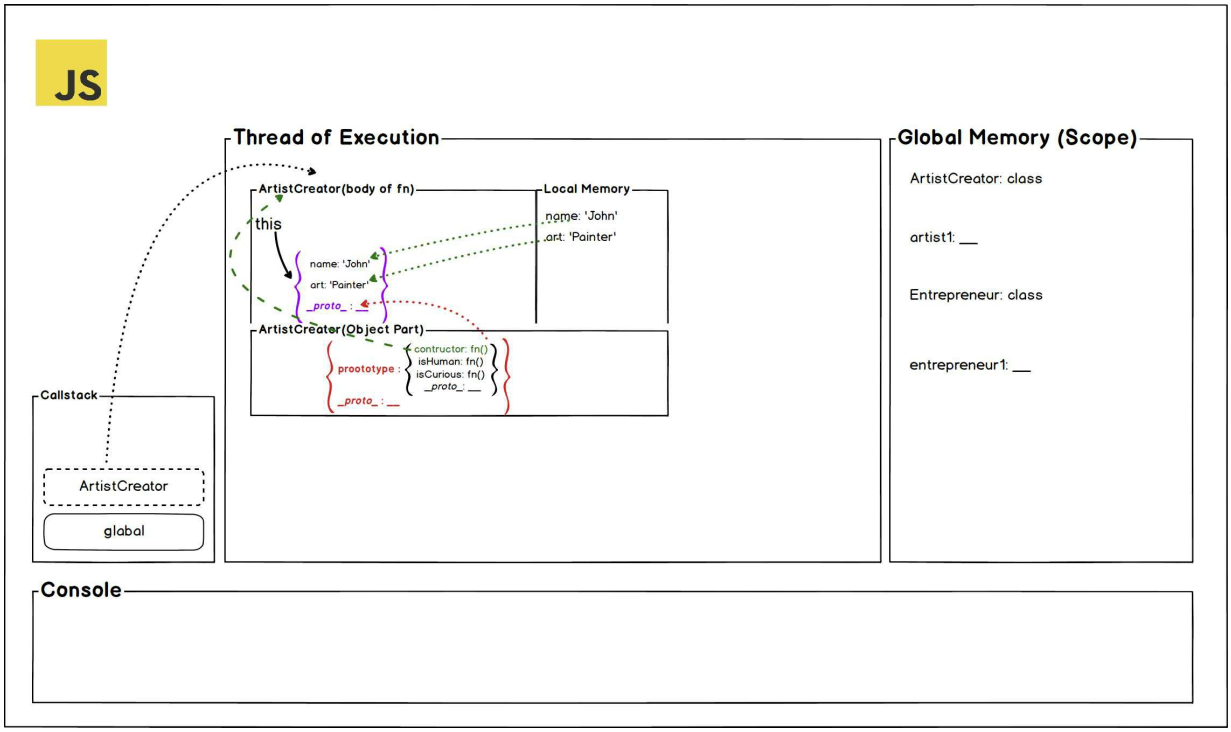
- In **class,** we should remember that the **constructor** method is available

on its prototype object. so whenever we try to execute class with a **new** keyword this constructor method from the prototype object is used (class is syntactic sugar for function and function always has prototype object).

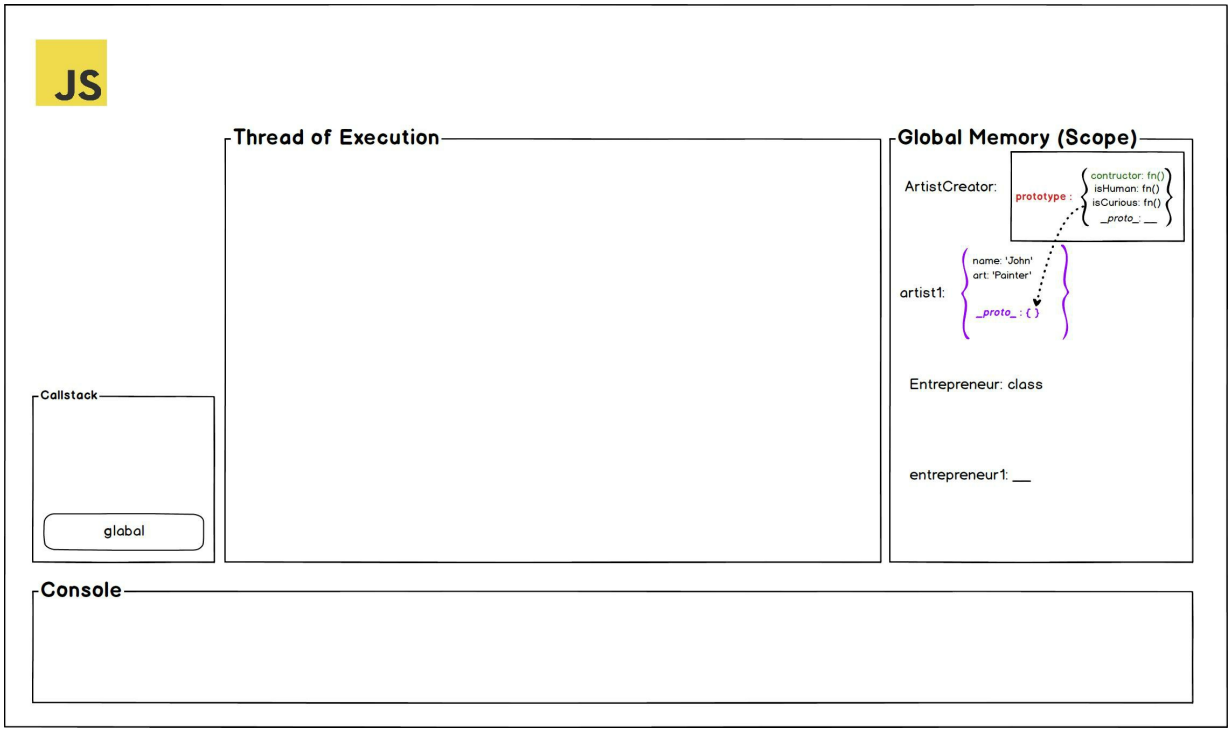
- See diagram we are executing constructor of ArtistCreator class.



Here, **this** keyword will point to the newly created object by the **new** keyword.

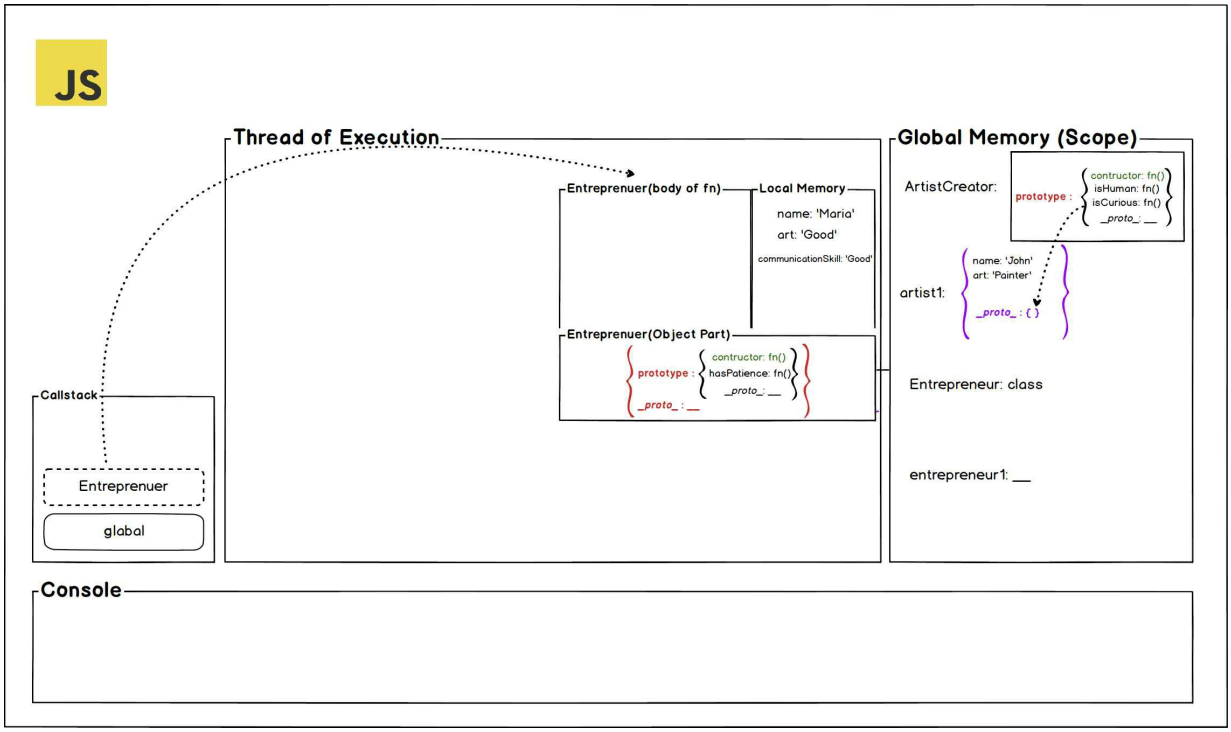


As per the below diagram, we find that using the **class** we are adding a function body for creating a new object and we are calling it constructor and putting it on the class's prototype object.



I think execution for ArtistCreator class was easy to grasp. now let's see what happens when we execute the below line.

```
335
336 let entrepreneur1 = new Entrepreneur('Maria', 'Writer', 'Good');
337
```



Note here, we have 2 major things.

1. **extends** keyword
2. **super** keyword inside the constructor.

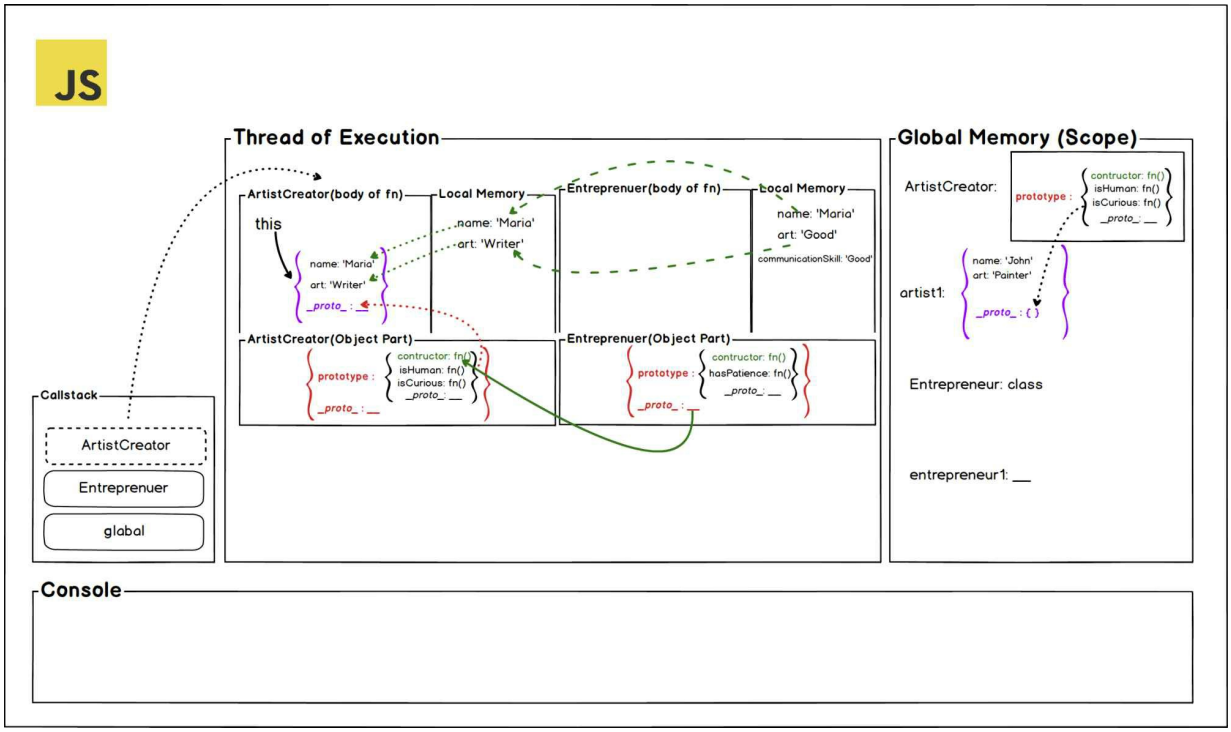
extends

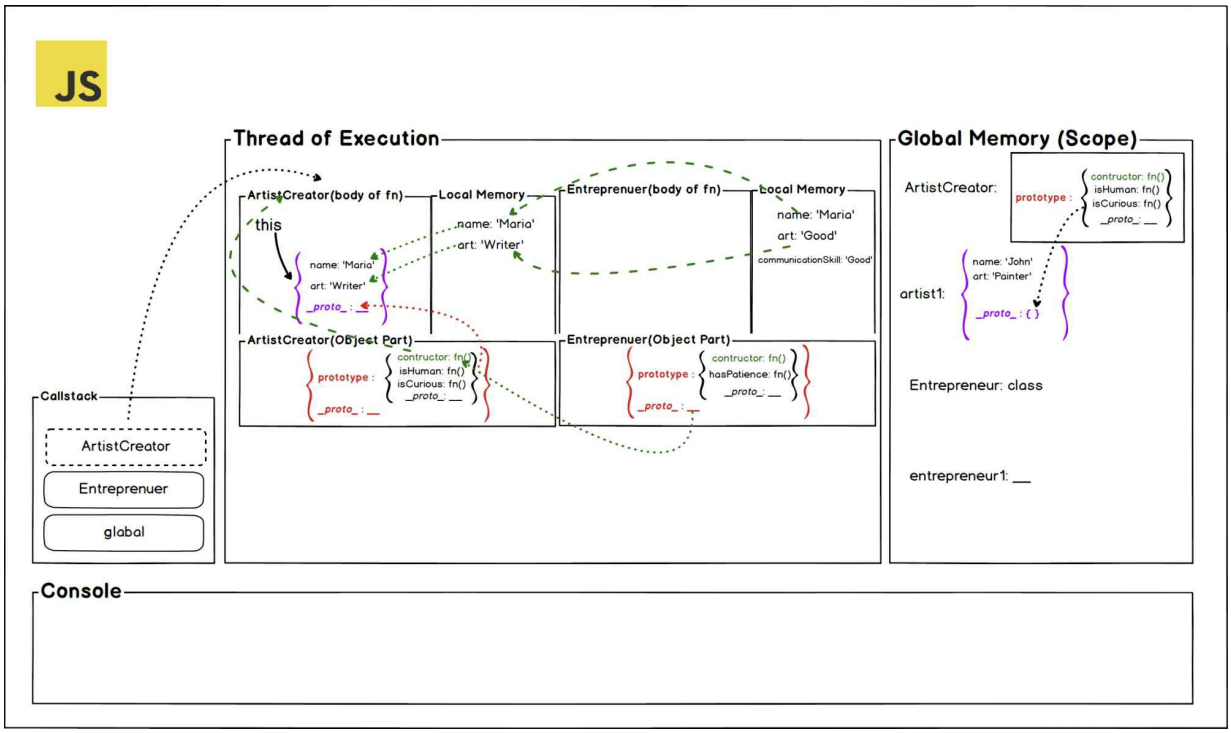
- the **extends** keyword behind the scene, set the reference of __proto__ of the current class (here in our case **Entrepreneur**) to the parent class's constructor(here in our case **ArtistCreator.prototype.constructor**)
- As per the diagram, we can see **Entrepreneur.__proto__ = ArtistCreator.prototype.constructor.**

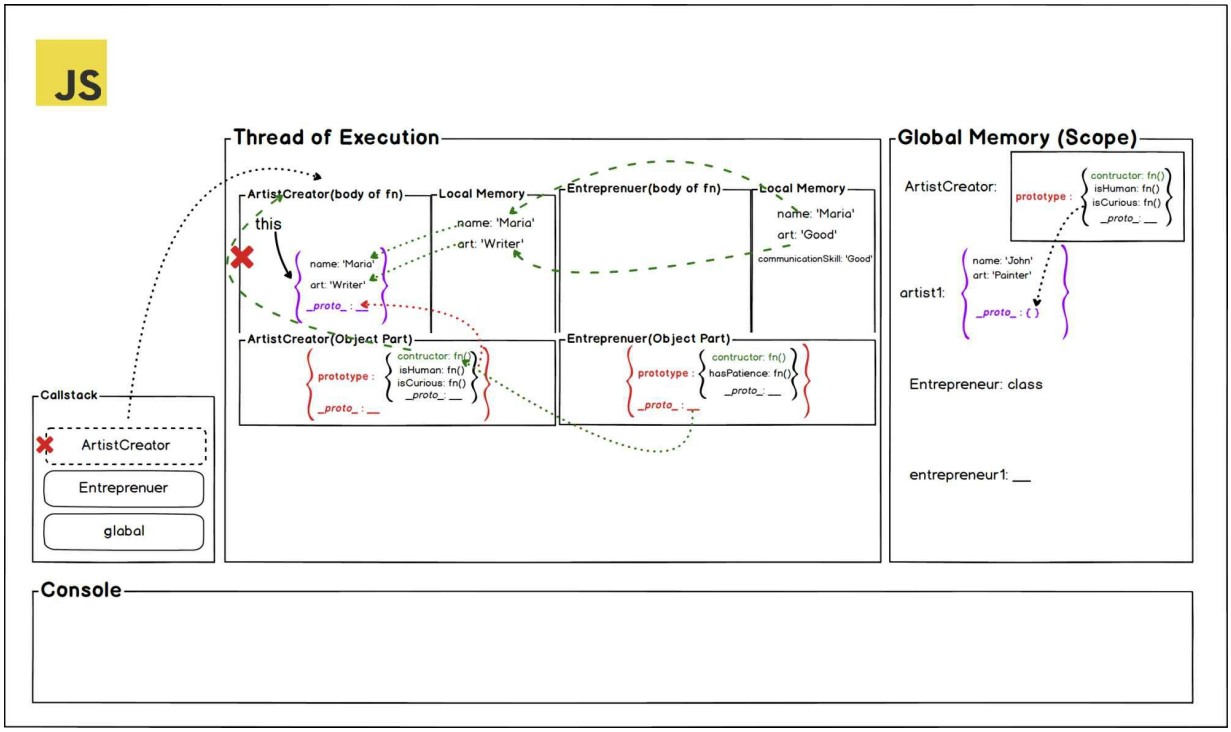
super

- The super keyword sends a command to call the method available on **Entrepreneur.__proto__.** Here it's pointing to the parent class's(**ArtistCreator.prototype.constructor**) constructor.
- Also, **this** keyword inside the super() call will create an empty object as well. Let's continue our visualization.

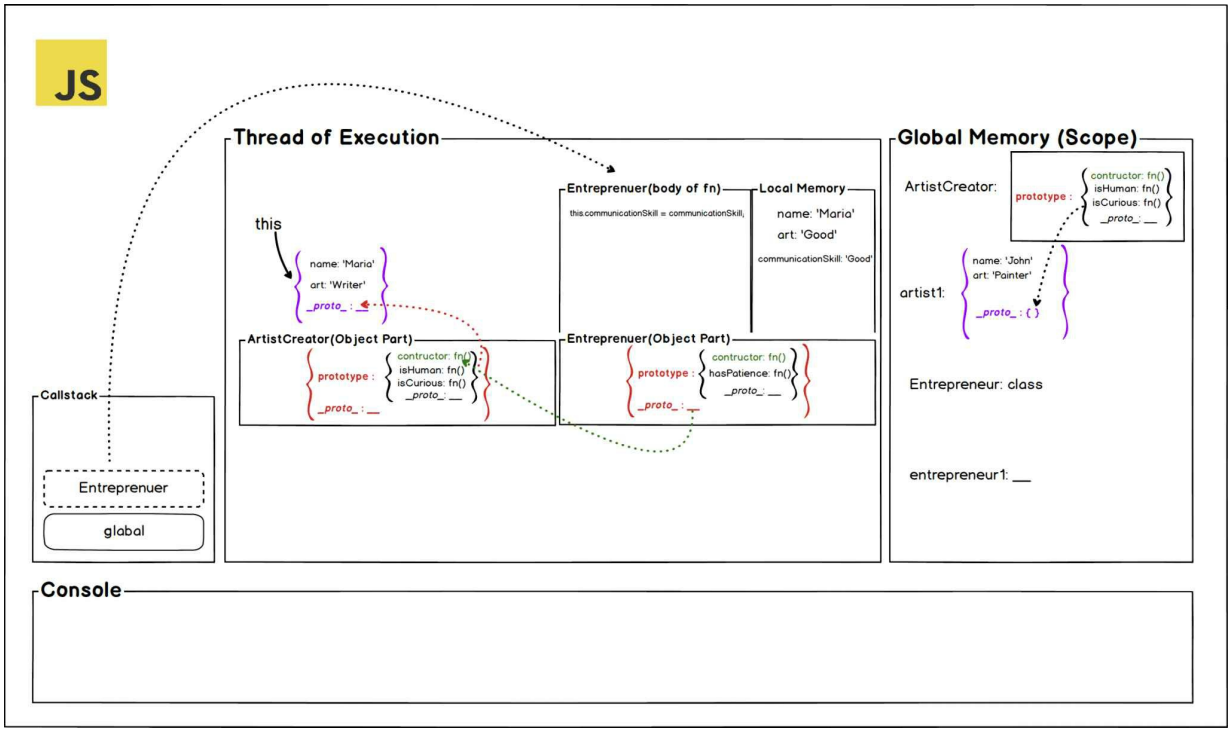
```
324
325   class Entrepreneur extends ArtistCreator {
326       constructor(name, art, communicationSkill) {
327           super(name, art);
328           this.communicationSkill = communicationSkill;
329       }
330       hasPatience() {
331           console.log(this.name);
332           return true;
333       }
334   }
335
```


JS
Thread of Execution
Global Memory (Scope)
ArtistCreator(body of fn)
Local Memory
this
name: 'Maria'
art: 'Writer'
proto : __
Entreprenuer(body of fn)
Local Memory
name: 'Maria'
art: 'Good'
communicationSkill: 'Good'
ArtistCreator(Object Part)
prototype :
contructor: fn()
isHuman: fn()
isCurious: fn()
proto: __
proto : __
Entreprenuer(Object Part)
prototype :
contructor: fn()
hasPatience: fn()
proto: __
proto : __
ArtistCreator:
prototype :
contructor: fn()
isHuman: fn()
isCurious: fn()
proto: __
artist1:
name: 'John'
art: 'Painter'
proto : { }
Entrepreneur: class
entrepreneur1: __
Callstack
ArtistCreator
Entreprenuer
global
Console

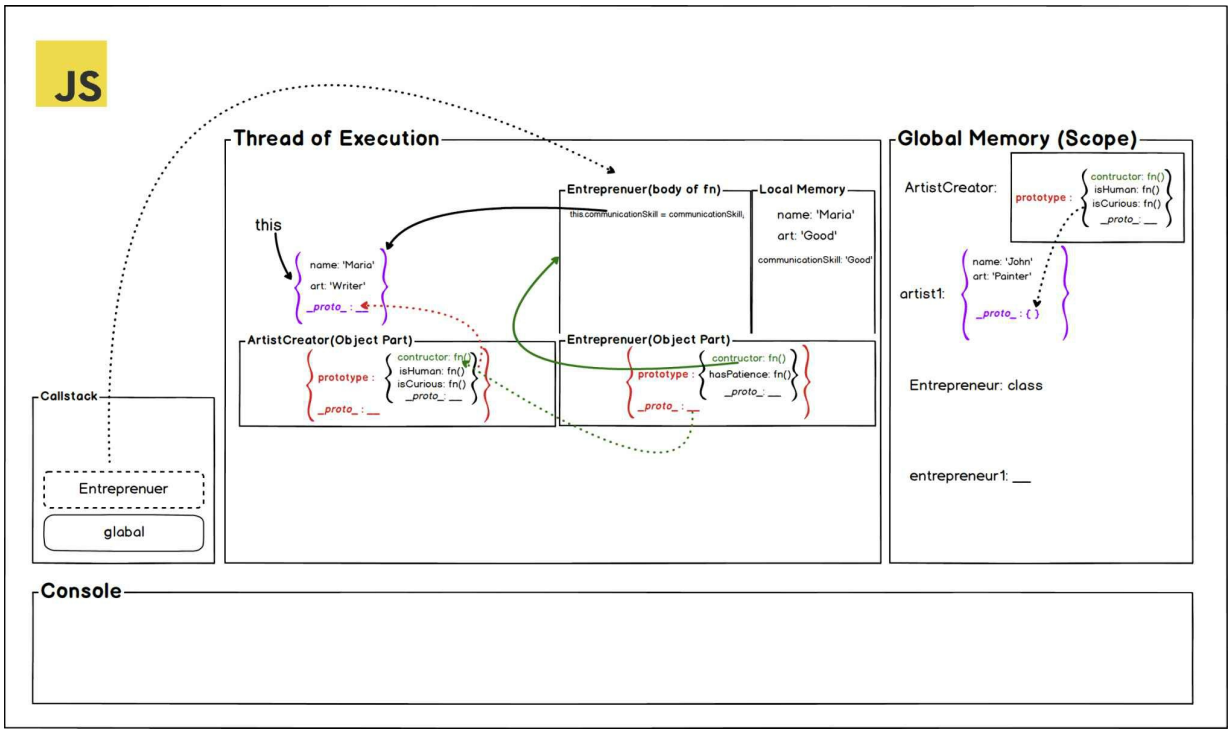

JS
Thread of Execution
Global Memory (Scope)
ArtistCreator(body of fn)
Local Memory
this
name: 'Maria'
art: 'Writer'
proto : __
Entreprenuer(body of fn)
Local Memory
name: 'Maria'
art: 'Good'
communicationSkill: 'Good'
ArtistCreator(Object Part)
prototype :
contructor: fn()
isHuman: fn()
isCurious: fn()
proto: __
proto : __
Entreprenuer(Object Part)
prototype :
contructor: fn()
hasPatience: fn()
proto: __
proto : __
ArtistCreator:
prototype :
contructor: fn()
isHuman: fn()
isCurious: fn()
proto: __
artist1:
name: 'John'
art: 'Painter'
proto : { }
Entrepreneur: class
entrepreneur1: __
Callstack
ArtistCreator
Entreprenuer
global
Console

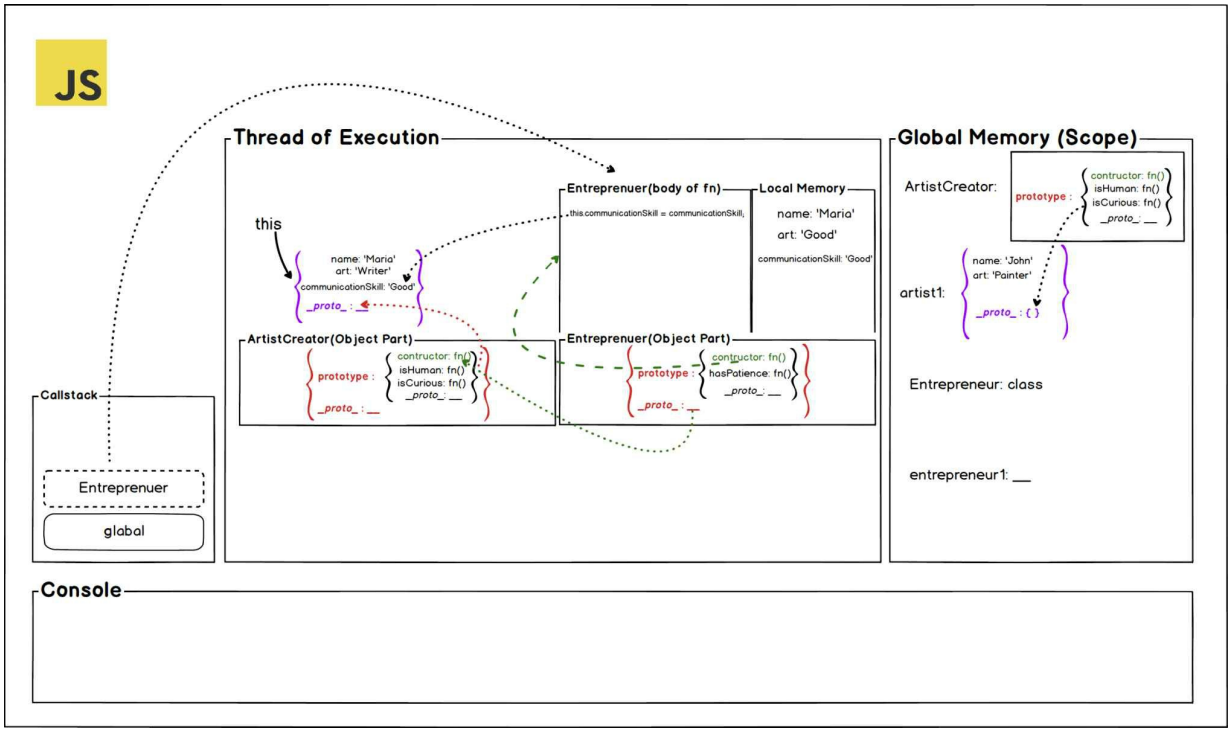


- After finish executing the super() call, we are now back in call of the constructor of the current class(Entrepreneur)
- In Entrepreneur's prototype object, we have reference to its own constructor which we are going to execute next.

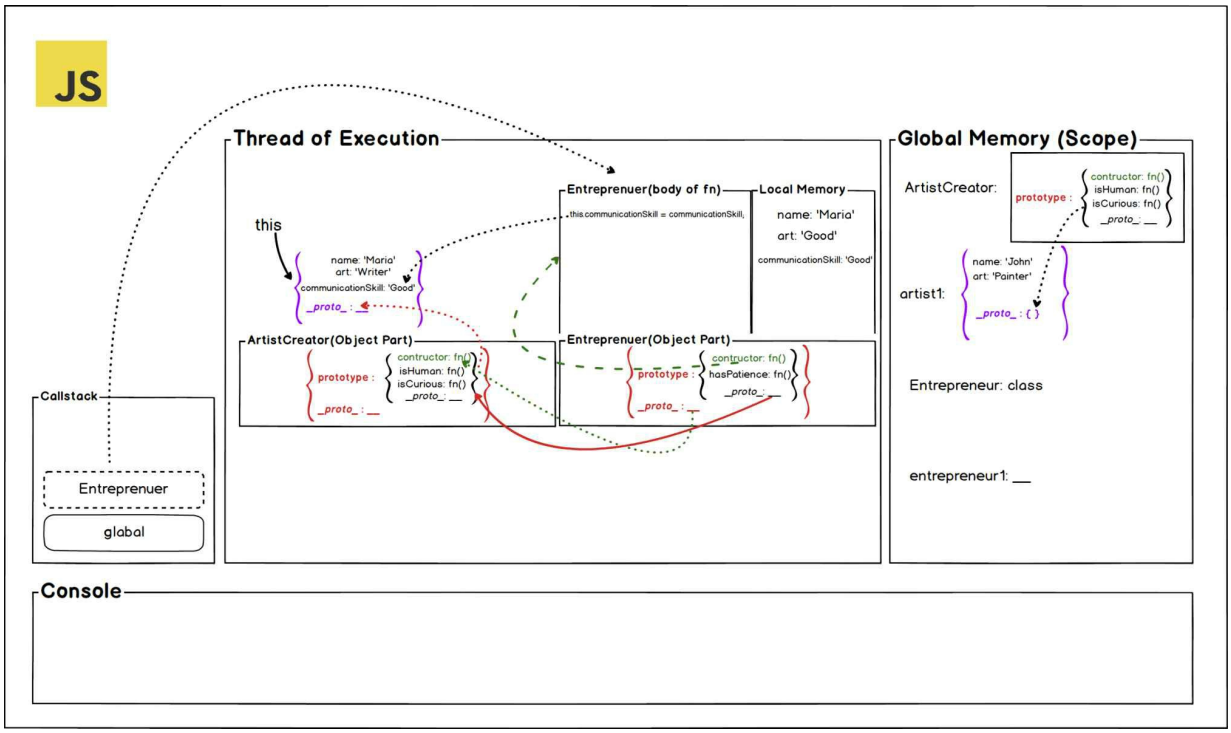


- Inside the body of an Entrepreneur's constructor, we are adding one more property to the newly created object.



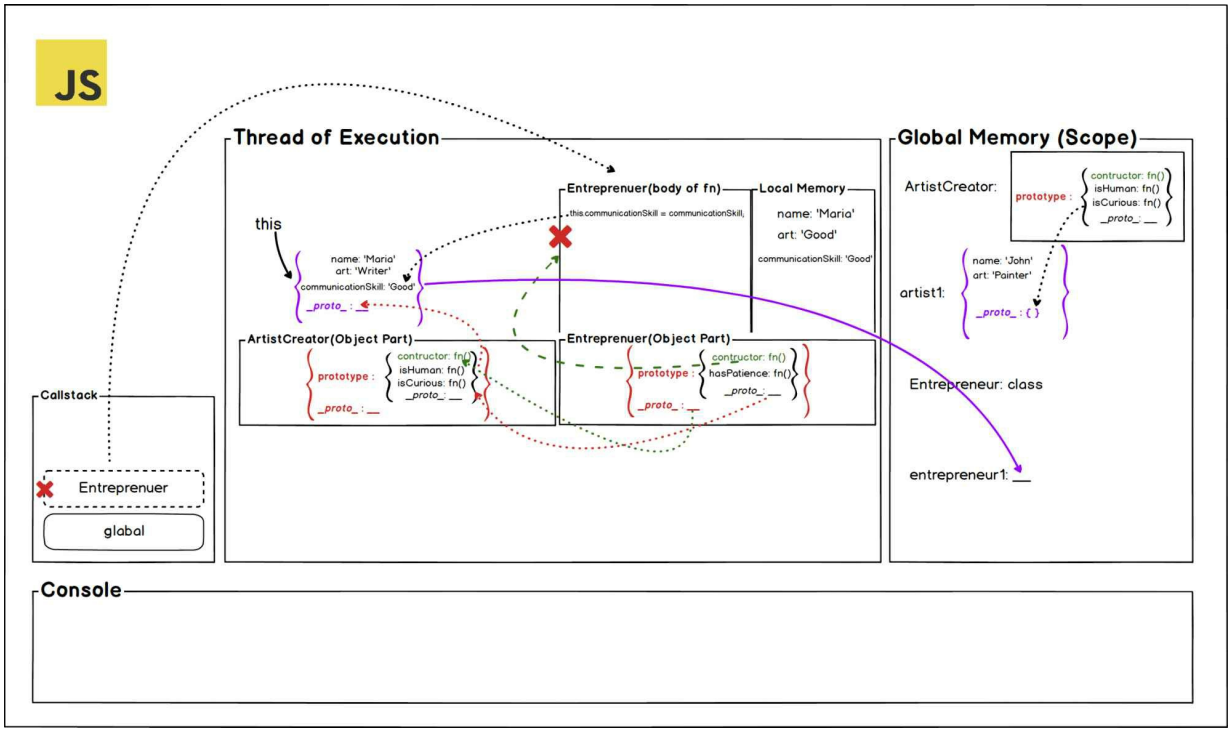


Now, the last thing is the newly created object's __proto__ will have reference to **Entrepreneur.prototype** object. see below diagram.

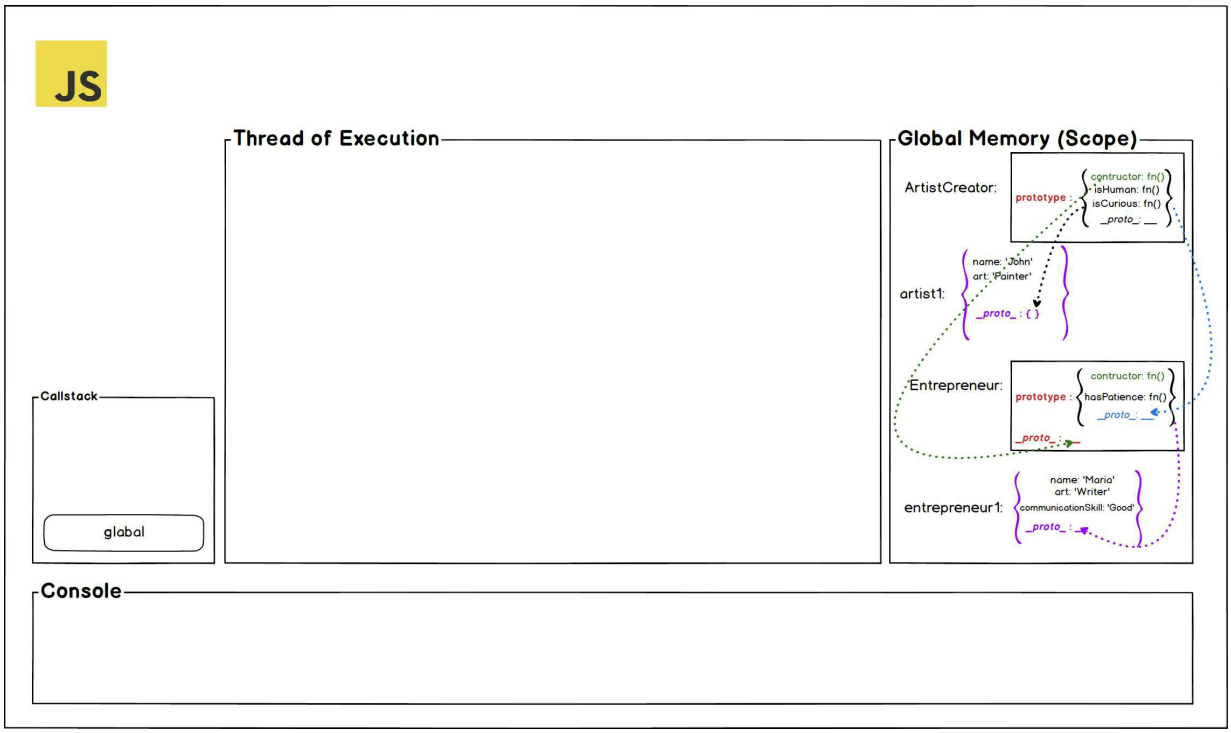


So, long story short. here is what happened by calling **Entrepreneur** class with **new** keyword step by step.

1. An empty object created from the super() call and **this** keyword will point to our newly created object.
2. All the properties were added from the super() call(see digram/code).
3. The newly created object's **__proto__** is pointing to the parent's prototype object(here **ArtistCreator.prototype**)
4. After re-entering the constructor of Entrepreneur, we added one more property to the newly created object.
5. Now, because we are inside Entrepreneur class, we actually updated __proto__ of newly created object to the Entrepreneur.prototype object.
6. But here we lose the functionalities of parent's class functions. So, the **extends** keyword helped us to set **__proto__** property of the newly created object's **__proto__** to the ArtistCreator.prototype object. (*think of it like* ***this.prototype.__proto__ = ArtistCreator.prototype***)
7. And last we finally returned our object to global memory and stored it in a variable called **entrepreneur1**.
8. If you still confused then please see the diagrams slowly and read the above 7 steps again😊.



- After all execution, you can see our final diagram😎.



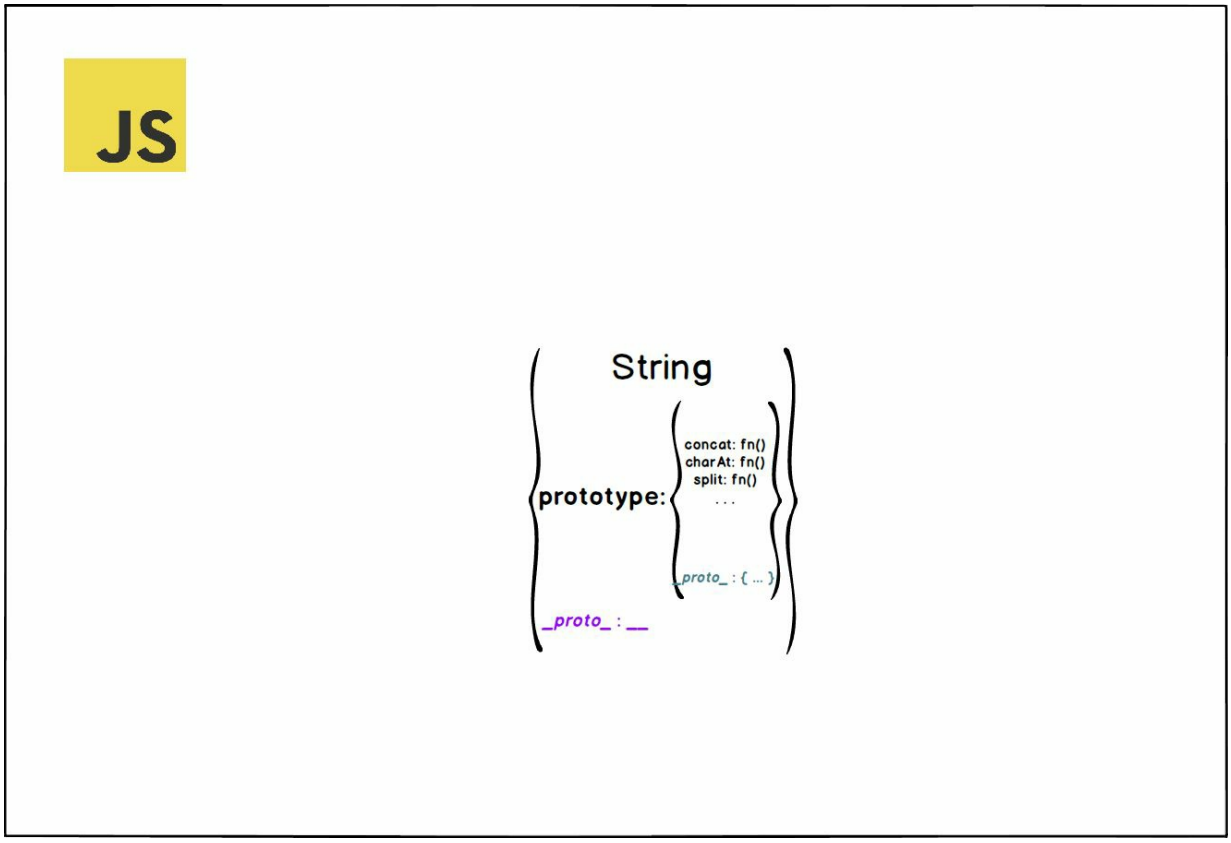
- Let’s not forget to see the output in our console as well. I hope I don’t

have to explain it😊.

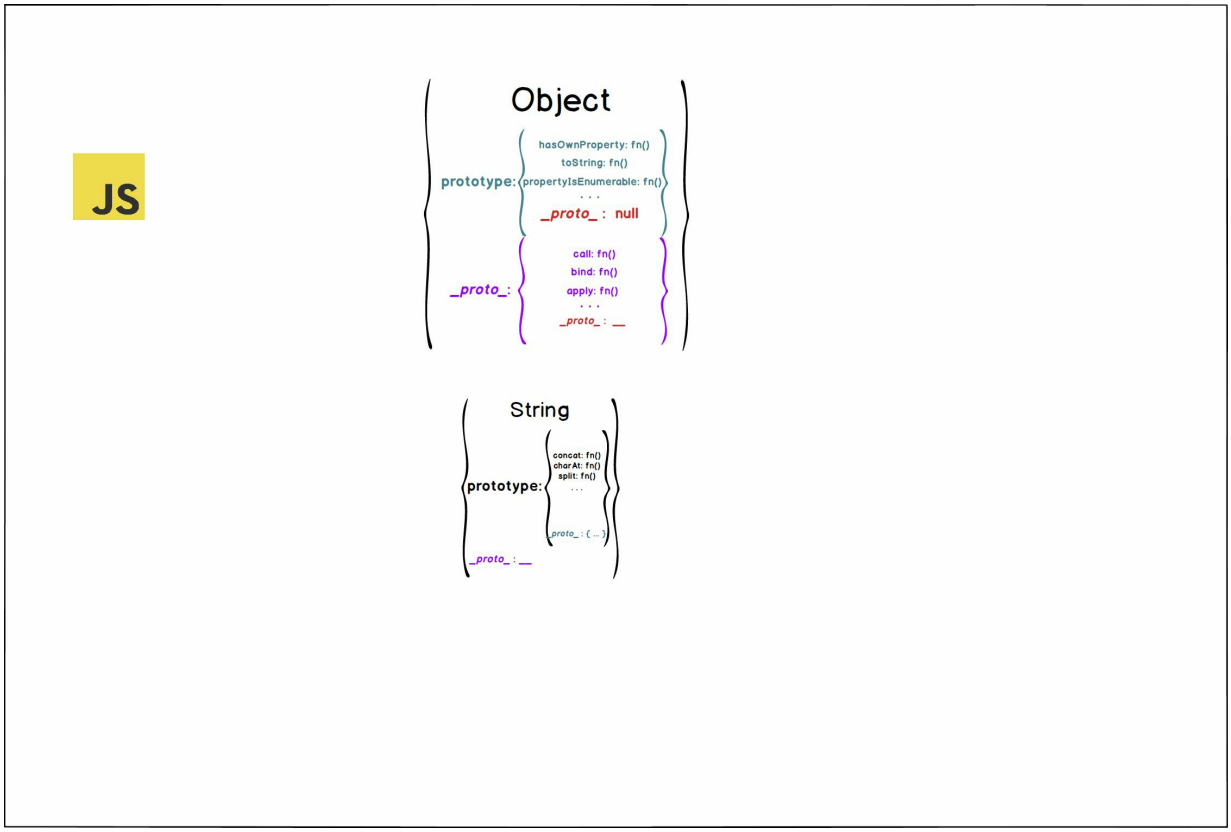
- We call it **prototype chaining** because we are literally extending __proto__ with prototype properties of others(mostly prototype object of function constructor or class).


```
That was freaking too much under the hood stuff I know, but this is all you need to know.
```

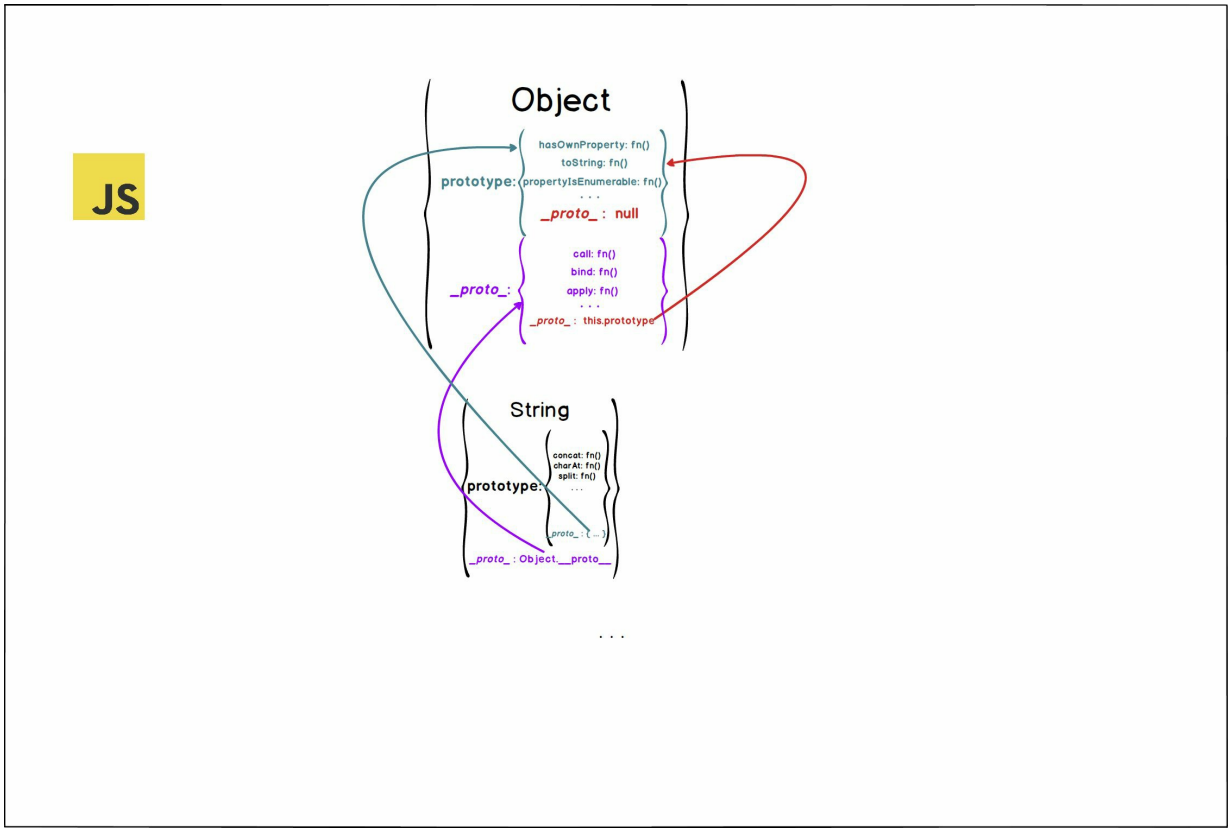
Before we close this chapter we will see how JavaScript itself uses the above mechanism.



- As we can see, **String** is a function constructor and it has a prototype and __proto__ object available to us. The prototype object itself has __proto__ property.
- Every string that we define in JavaScript is actually made from this String constructor. That is the reason why you can access all string methods of it via __proto__ reference which points to this prototype object(String.prototype).

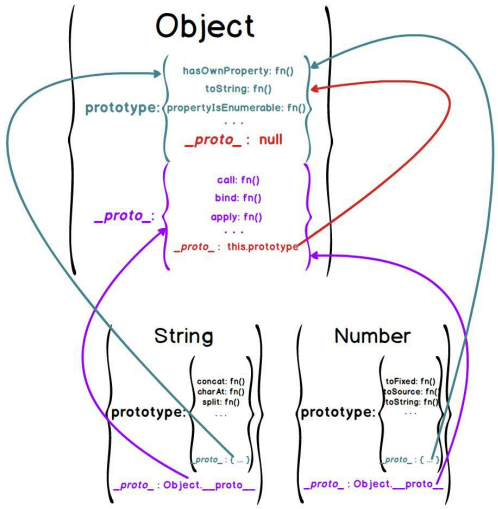


- Every JavaScript object literally borns from an **Object(**notice capital ‘O’**). This is the end of our chaining(will see soon).**
- The **Object** itself has a prototype and __proto__ property available to it.
- The prototype object has methods like hasOwnProperty(), toString() available which you might have used in your day to day code.
- Let’s see how these **String** and **Object** has a connection with one another.



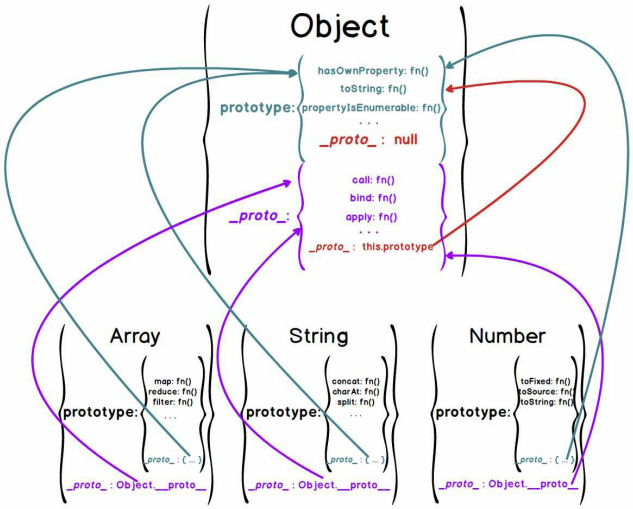
- As you can see, **String's** __proto__ point to our global **Object's** prototype. One more thing our global **Object's** __proto__ object has call(), bind(), apply() and other function but __proto__ of **Object's** __proto__ is **this.prototype.** And the last **Object is also a function constructor so the prototype object is available to it.** Here, __proto__ of the prototype is **null.** The main reason is **null** because here our __proto__ chain ends😎.
- Let's go through other similar objects as well.

JS

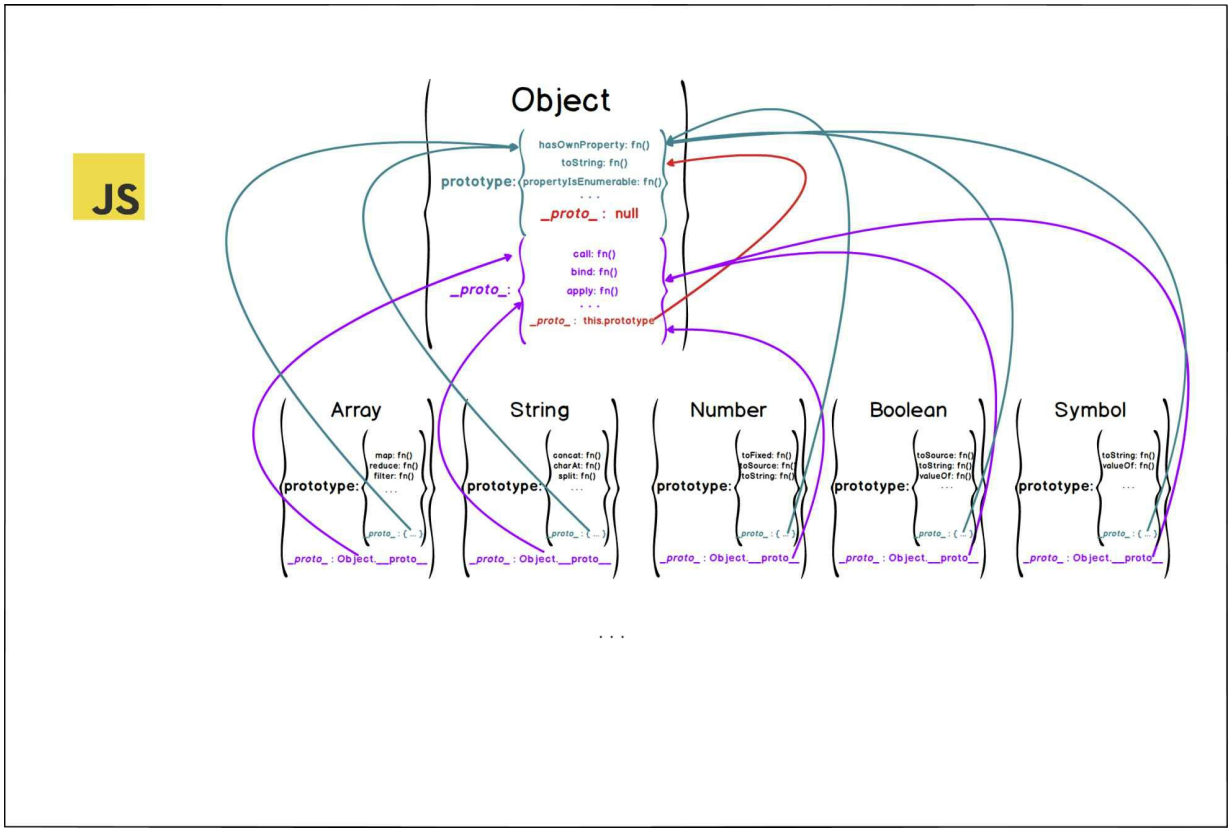

Object
hasOwnProperty: fn()
toString: fn()
prototype:
propertyIsEnumerable: fn()
. . .
proto : null
call: fn()
bind: fn()
proto:
apply: fn()
. . .
proto : this.prototype
String
concat: fn()
charAt: fn()
prototype:
split: fn()
. . .
proto : { . . . }
proto : Object.__proto__
Number
toFixed: fn()
toSource: fn()
prototype:
toString: fn()
. . .
proto : { . . . }
proto : Object.__proto__

. . .

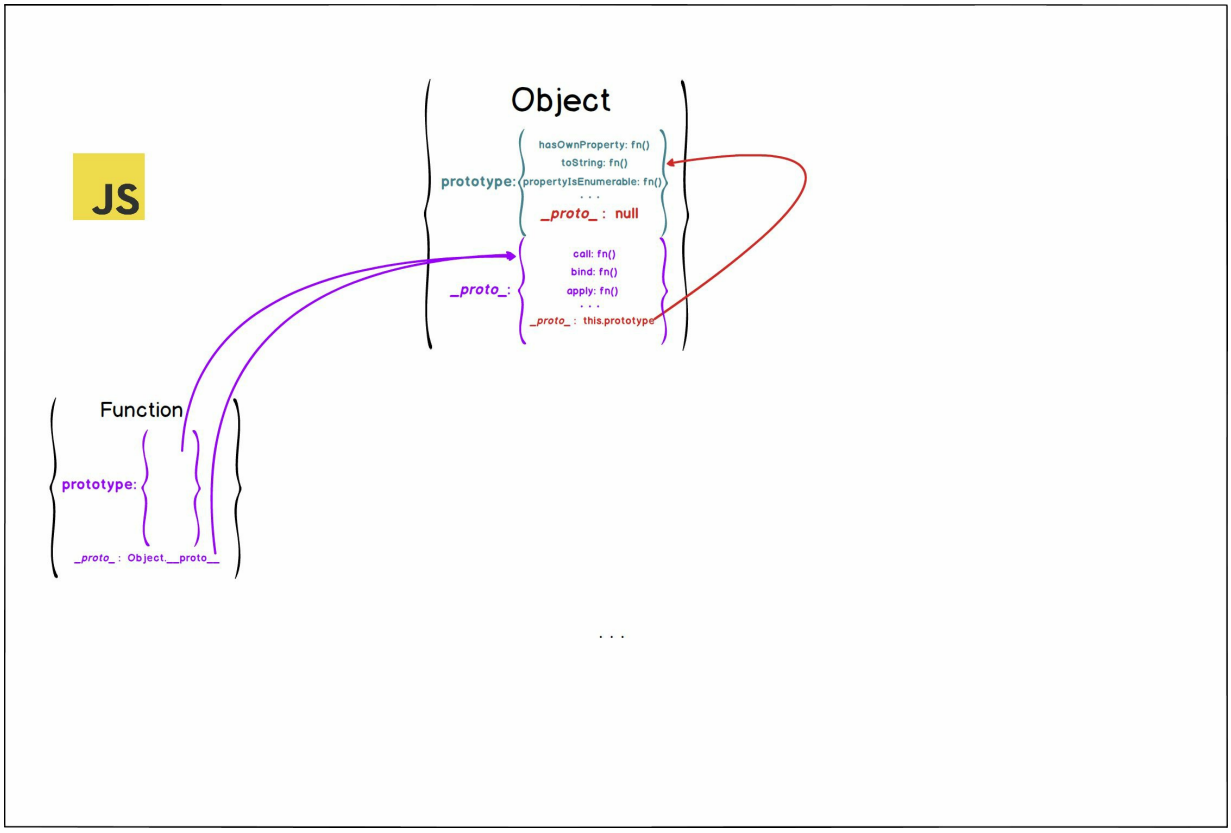
JS


Object
hasOwnProperty: fn()
toString: fn()
prototype:
propertyIsEnumerable: fn()
. . .
proto : null
call: fn()
bind: fn()
proto:
apply: fn()
. . .
proto : this.prototype
Array
map: fn()
reduce: fn()
prototype:
filter: fn()
. . .
proto : { . . . }
proto : Object.__proto__
String
concat: fn()
charAt: fn()
prototype:
split: fn()
. . .
proto : { . . . }
proto : Object.__proto__
Number
toFixed: fn()
toSource: fn()
prototype:
toString: fn()
. . .
proto : { . . . }
proto : Object.__proto__

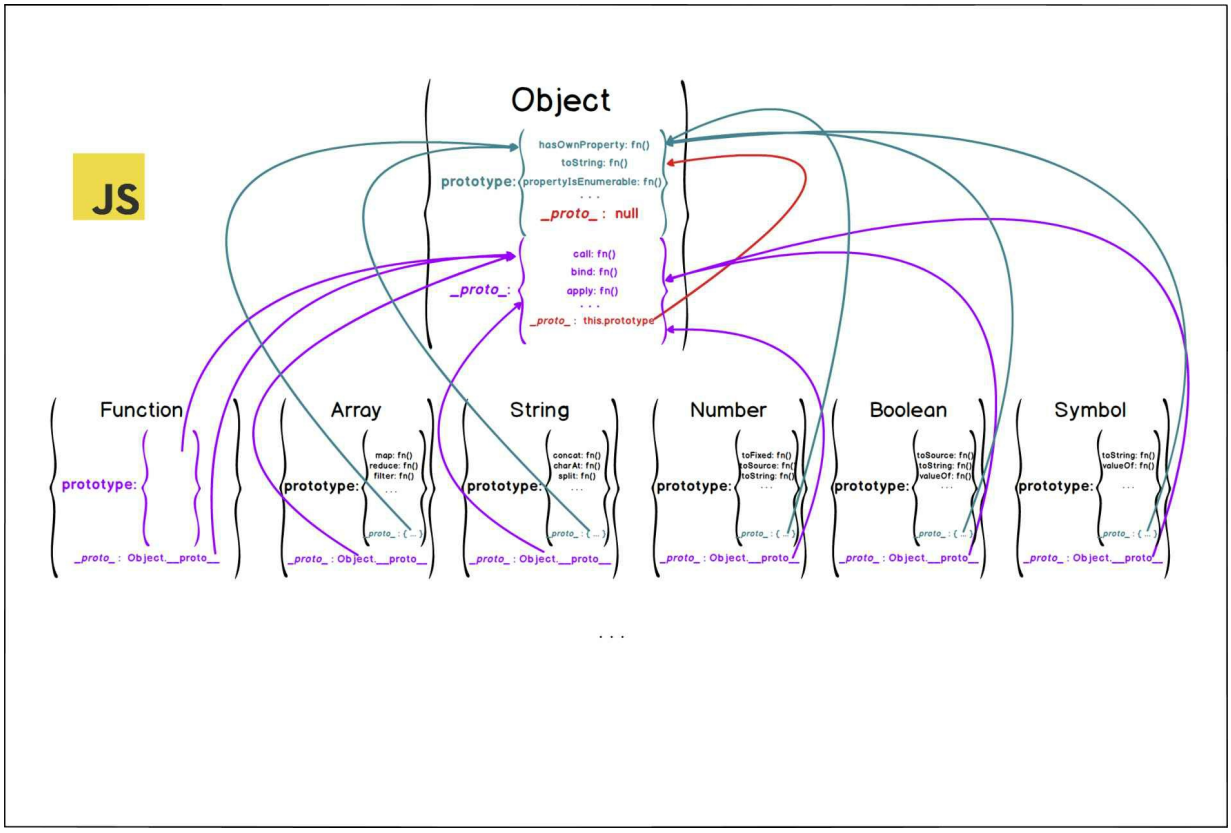
. . .



The Function is a special type of constructor. it’s a prototype and __proto__ both points to Object.__proto__. See the below diagram😁.



- Finally combined all diagram's native constructors. (there are a lot more than these constructors but no need to show them in the below image as we understood the nuance of it).



There you have it, my friends, the most complete guide to __proto__, prototype, and object linking😁. Enjoy the further journey with the next chapters.

* * *

4

Asynchronous Execution

JavaScript (without environment) by default is single-threaded (one task at a time) & synchronous (Line by line & top to bottom execution). If you think this is pretty bad because we can't do multiple tasks at a time and we are blocking our execution until the current line finishes its execution. So, JavaScript needs a way to tackle this. Let's see how.

JavaScript alone is not enough!

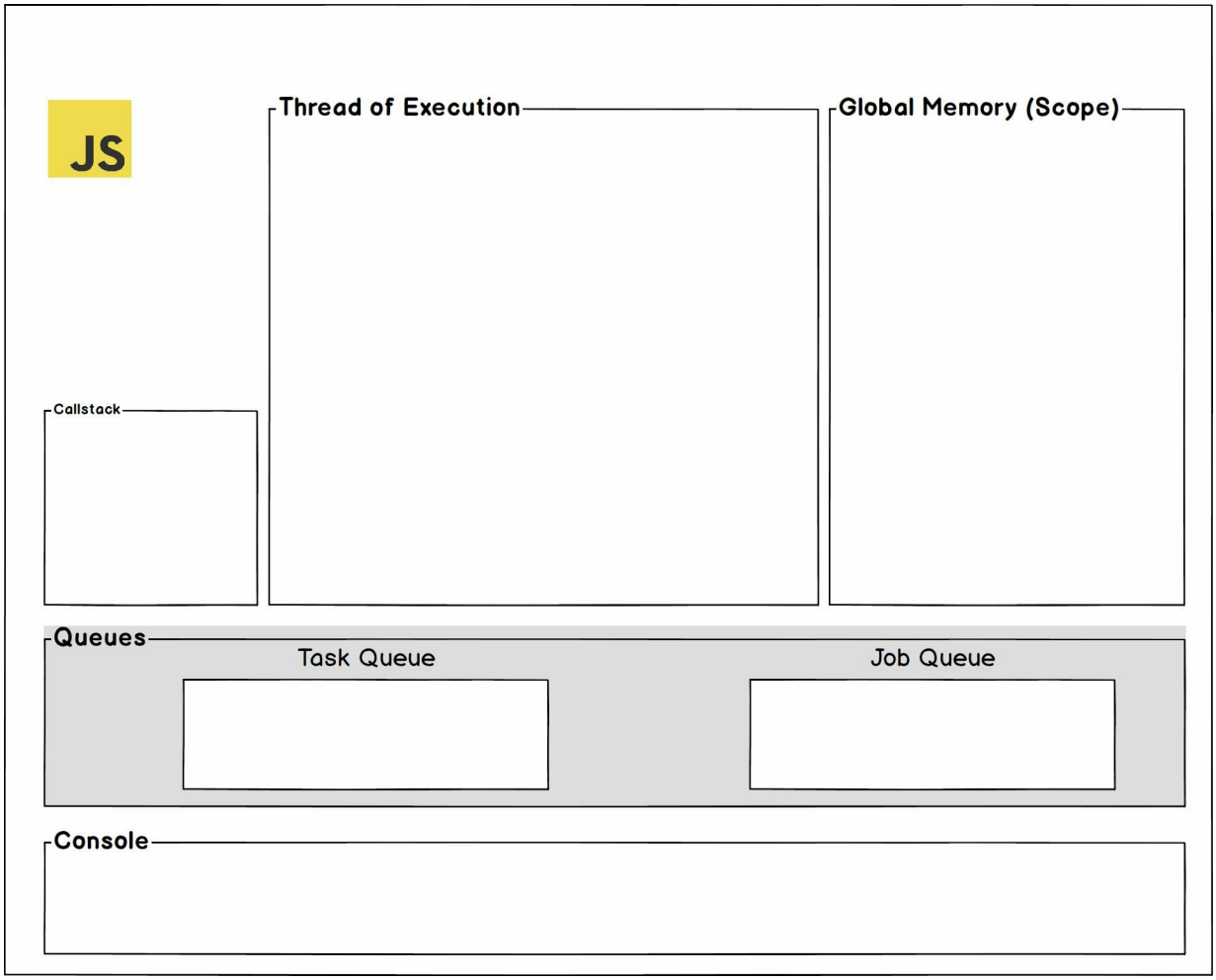
- When you play cricket/football, a single most powerful player is not enough to win the entire match. you need to play as a team and each player's contribution matters.
- Same way JavaScript alone is not enough, we need extra players or rather a team.
- That is where the extra "environment" comes and helps JavaScript.
- There is 2 famous JavaScript environment we use a lot.

1) Browser
2) NodeJS

- In this chapter I will explain only the browser environment, we will understand the nodejs environment in a later chapter.
- We have seen our "mental model" in chapter 1, but I didn't tell you that it was incomplete because I didn't want you to confuse there. It's time to show the big "mental model" with the environment surrounding.
- Brower environment is so big which include HTML/CSS parsers, console, network monitoring, storages(local, Indexeddb, web SQL, etc), performance tool, etc. we only going to focus on how asynchronous part handle through something called "Queues" & "Event Loop".

JavaScript is synchronous! (Don’t Forget)

- Timing and resource utilization are always important in every single app or the real world.
- JavaScript is synchronous and it’s so important that a lot of people forget that JavaScript is just a scripting language that is meant for just making webpage dynamic in its early days.
- Ultimately JavaScript never builds for servers or complex rendering that we do today.
- But some people wanted to make javascript more flexible and powerful without breaking the origin of the language mindset.
- In 2004, Google released Gmap and it was built with fully JavaScript on the browser and stunned the world what potential JavaScript language has.
- Without doing history talk more, let’s come to the actual implementation of an asynchronous environment.
- You will see the new **Queues** part will be drawn, that is where our journey of asynchronous starts😁.
- Remember, **‘Queues’** are implemented by the browser so sometimes you see different output/results than the actual desired one from different browsers, but we will discuss what is right and what the desired approach should be. let’s have a sneak peek at our new mental modal!
- Don’t try to understand deeply, we will go through it in details soon.



Current Extended Mental Model

Callbacks: functions that are called back!

- Before we move further it's important to talk about callback functions a little bit.
- Callbacks are simple functions that are invoked whenever they are called back(AKA whenever a particular task has been finished after that).
- There are 2 types of callbacks.

1) Sync Callback/s
2) Async Callback/s

1) Sync Callbacks

- Sync callbacks are called on iterators which is easy and quick to execute (fast operations)
- An example of a sync callback is given below.

```
51
52  let arrIterator = [1, 2, 3];
53
54  arrIterator.forEach(function (element, index) {
55      console.log({ element, index });
56  });
57
```

Sync callback

2) Async Callbacks

- Async callbacks are time-consuming tasks or error pruning

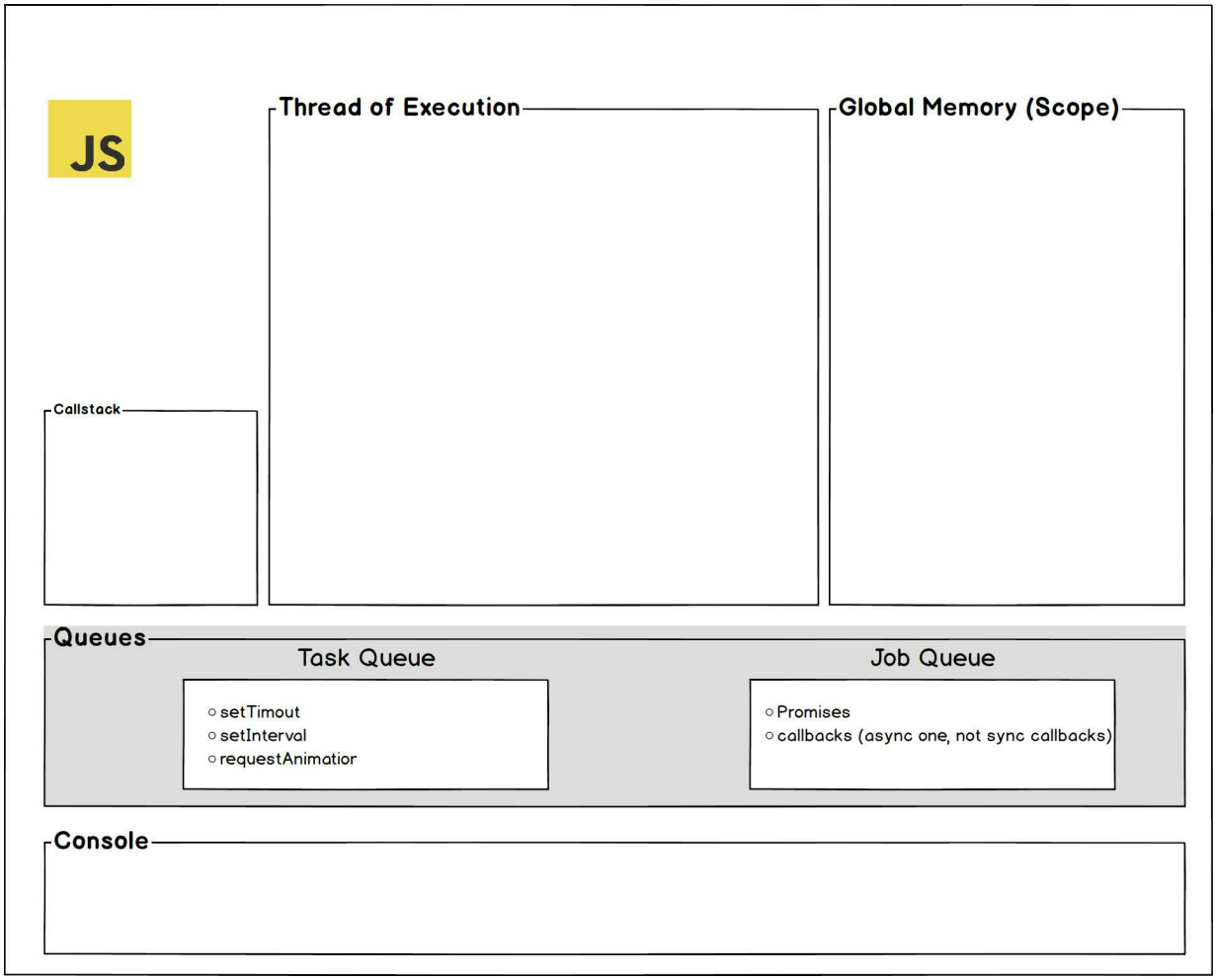
functions(might finish a task or won't finish it at all kind of).

- An example of async callbacks is given below!
- Don't try to understand deeply, just an example😂. will understand shortly.

```
58
59  let asyncForEach = function (array, callback) {
60      array.forEach(function (element, index) {
61          setTimeout(() => callback(element, index), 0);
62      });
63  };
64
65  asyncForEach(arrIterator, function (element, index) {
66      console.log({element, index});
67  });
68
```

Async callback

- We have extended our mental model of course, but one thing to note that our extended mental model is not just JavaScript. It's JavaScript + browser APIs(Those queues' function are part of the browser environment, aka setTimeout,setInterval, etc).
- But there is one twist here, promises are more complicated more than that, they are part of javascript and browser both. will see that in the mental model.
- Currently, let's see a simpler version mental model with queues.



Mental Model With Queues

```
Queue: It's a simple data structure where the first in first out rule is
followed(aka, whoever comes first will go first). Just like a movie ticket purchase
line.
```

Task Queue

- first of all weird thing is, task queue is actually not a queue, it is a **set** (in Cpp world it’s struct).
- Tasks are added in task queue are picked and removed it, not like a queue that it’s dequeued. To keep it simple let’s treat it as a normal

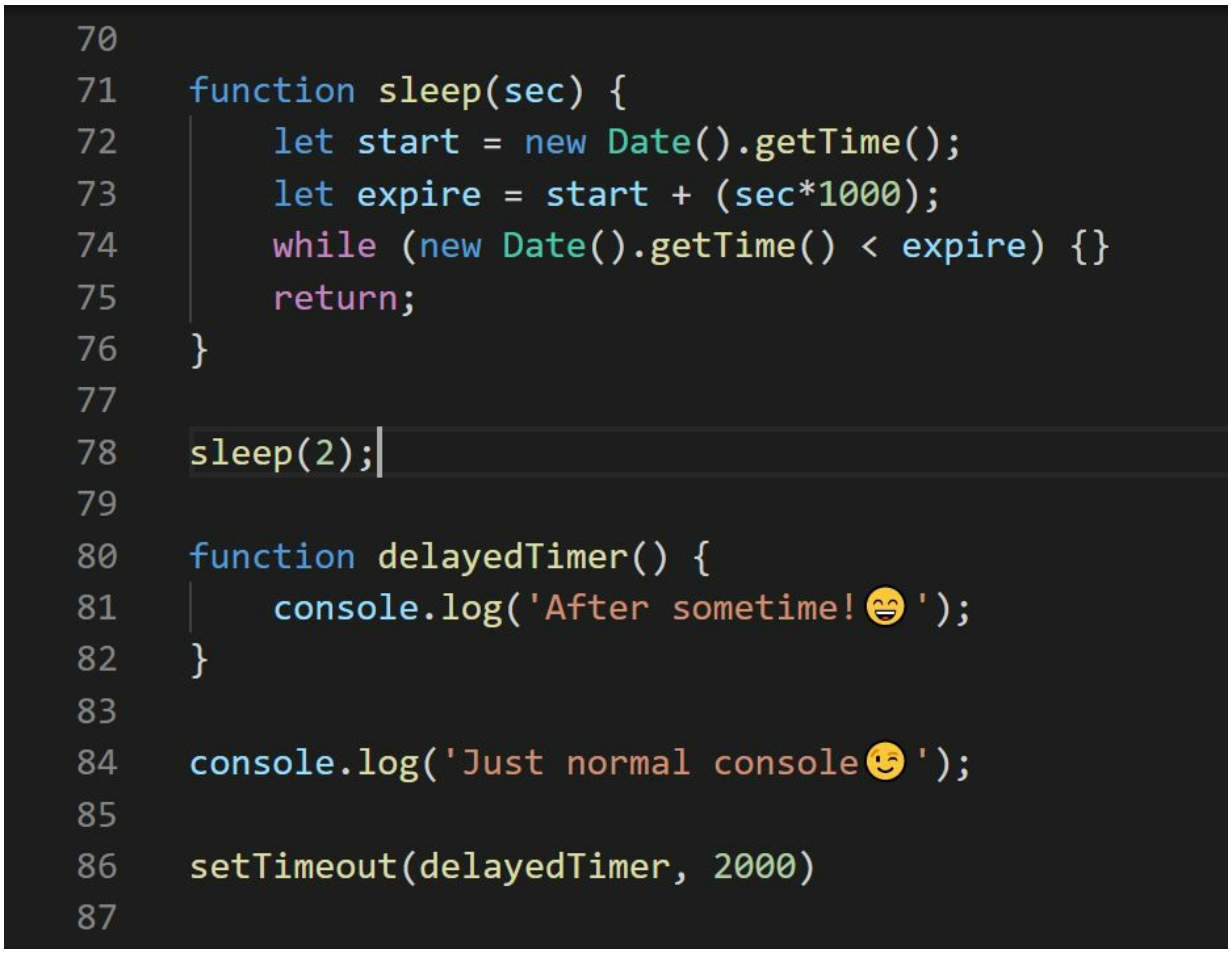
queue(most javascript developers do that and will do the same)😁.

- **Macro Task**, is a term used sometimes for task queue.

Job Queue

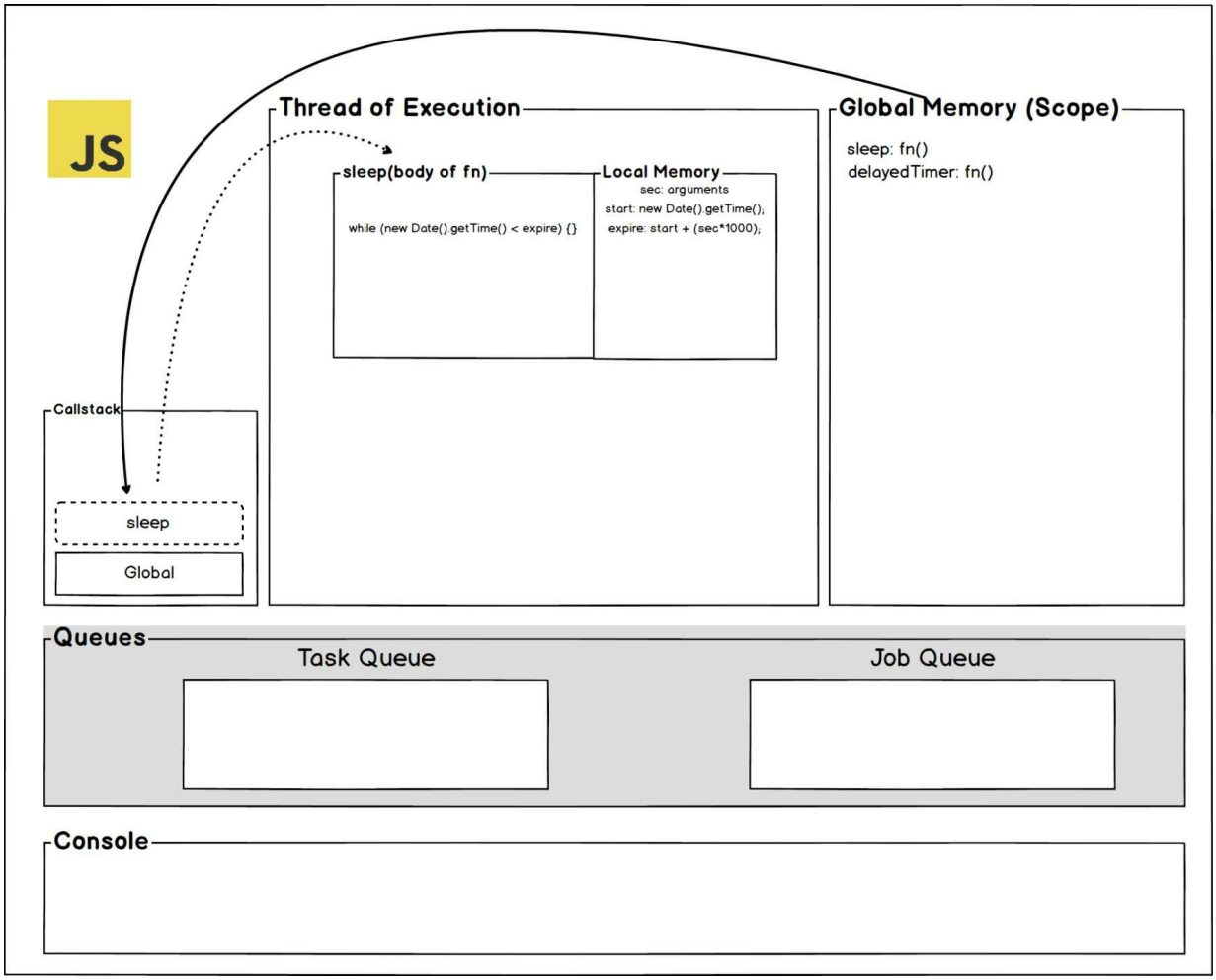
- The job queue is an actual queue where tasks are queued and dequeued. it’s a real queue.
- **Micro Task**, is a term used sometimes for job queue and but it’s called Job queue in ES6+ specifications.

Time to understand stuff with an example!

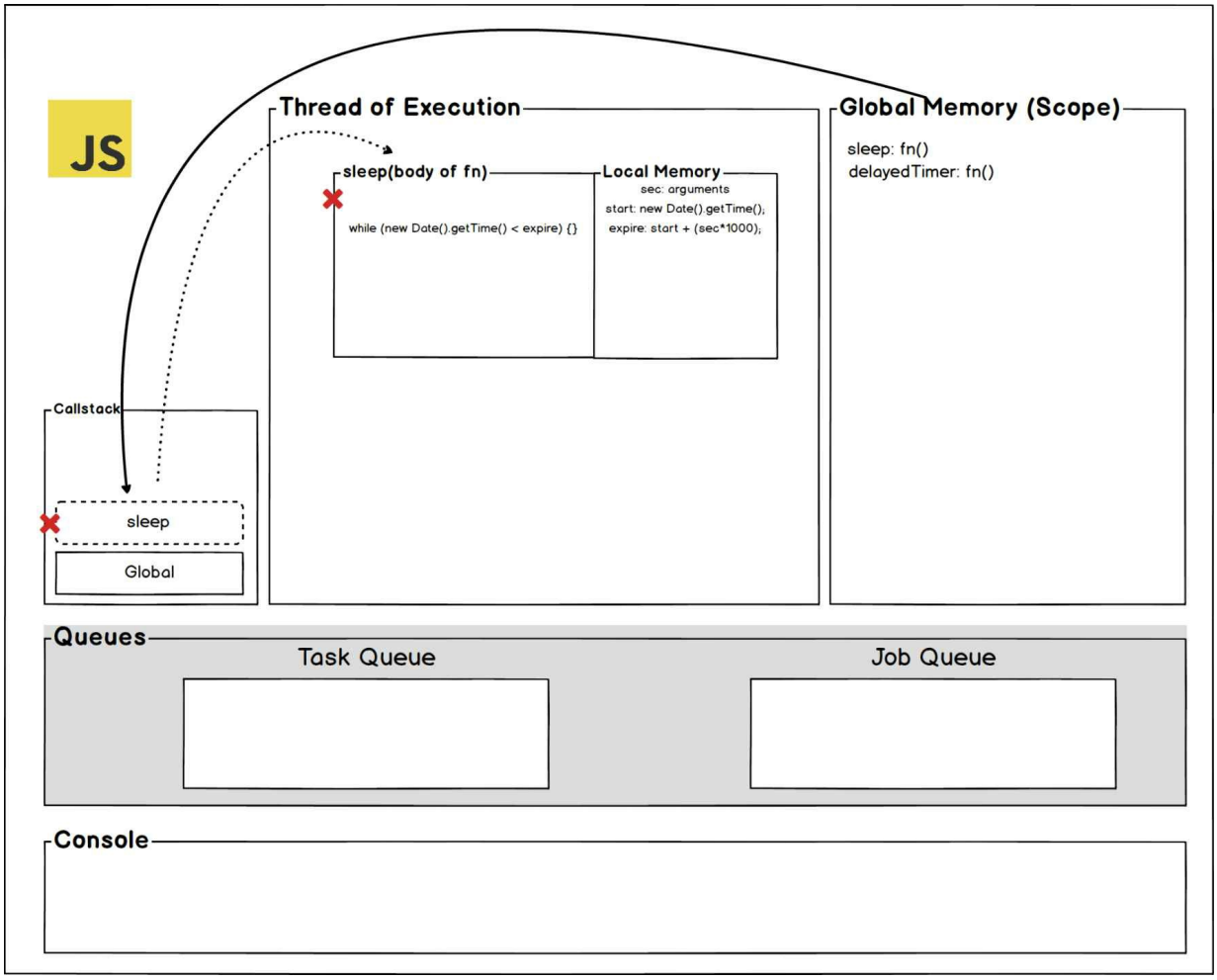


```
70
71  function sleep(sec) {
72      let start = new Date().getTime();
73      let expire = start + (sec*1000);
74      while (new Date().getTime() < expire) {}
75      return;
76  }
77
78  sleep(2);
79
80  function delayedTimer() {
81      console.log('After sometime!😄');
82  }
83
84  console.log('Just normal console😉');
85
86  setTimeout(delayedTimer, 2000)
87
```

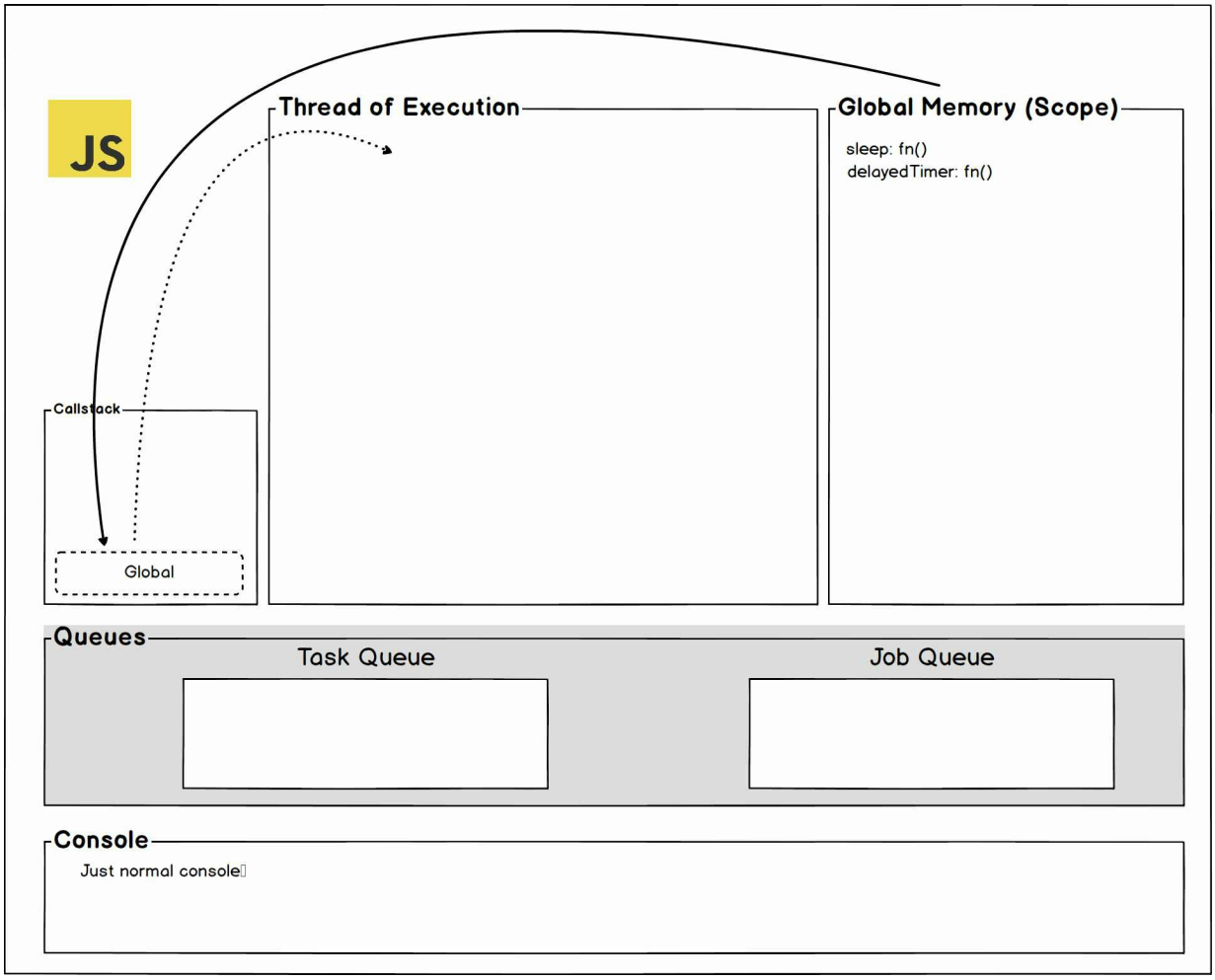
- There is a sleep function which I simulated that block the function for given seconds. Let's see our mental model of it, how it's going to be executed!
- **setTimeout is a browser feature and it's not a part of regular javascript. Waiting for the given time period and then giving the callback function back to the task queue is done by the browser.**

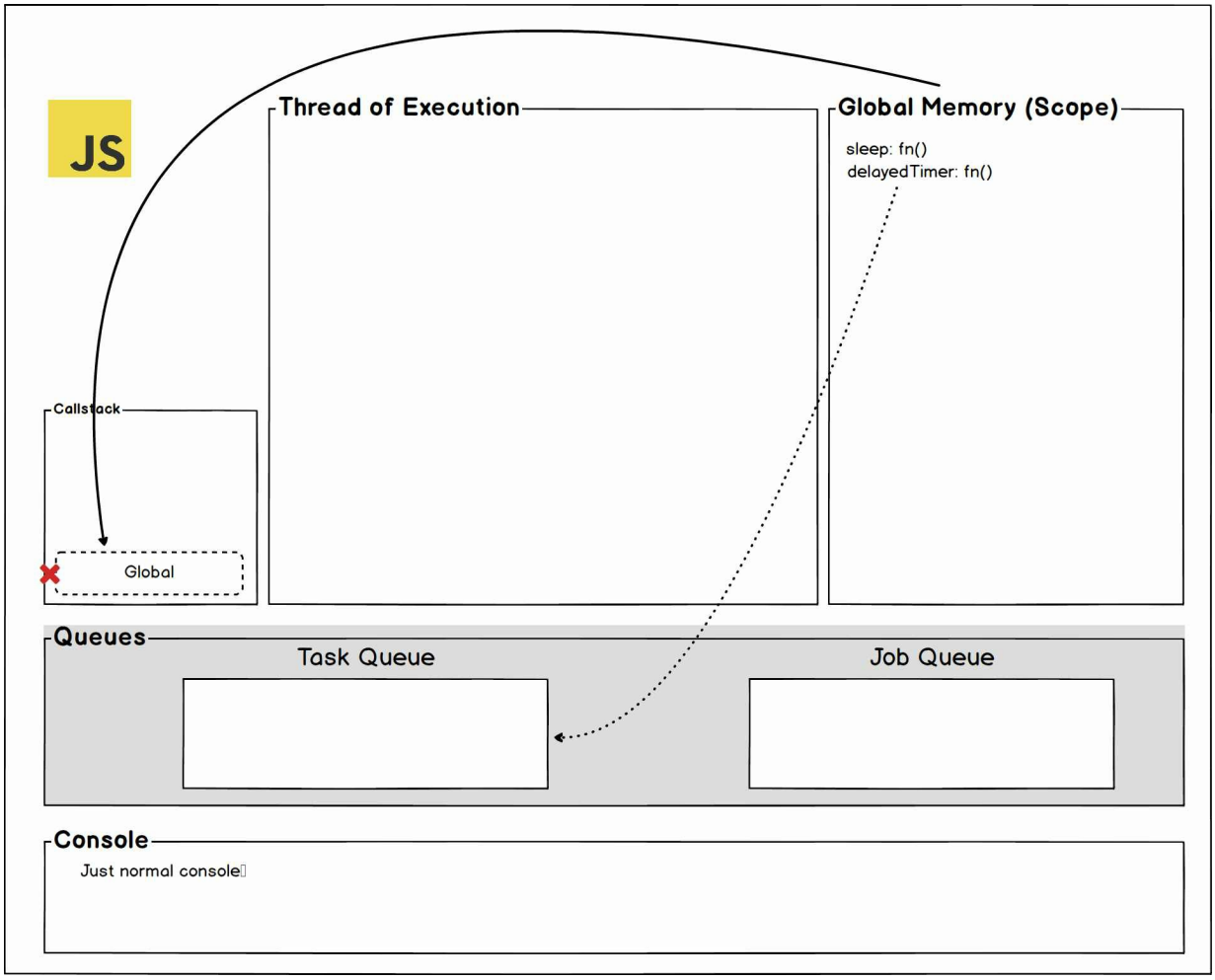


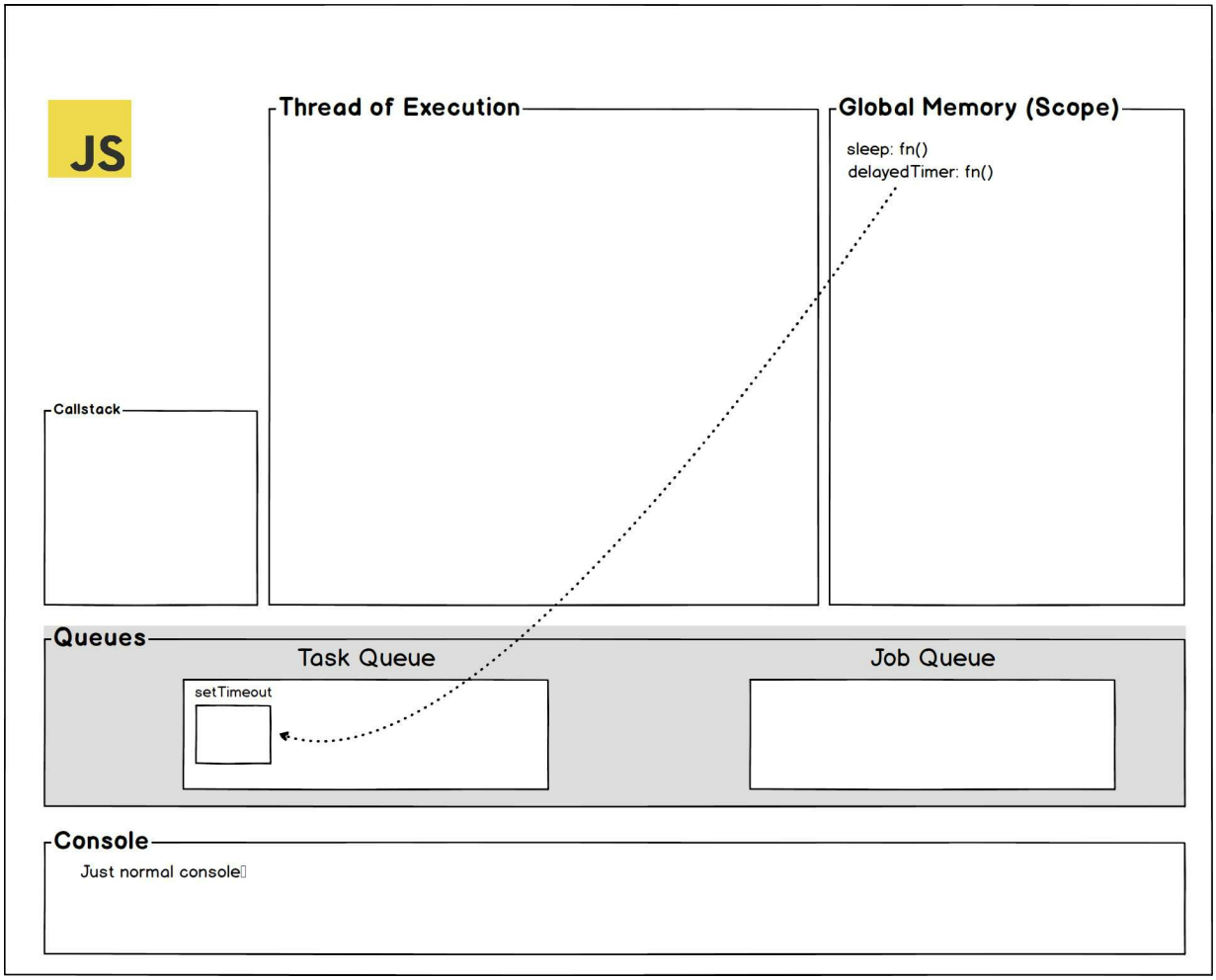
- Now let’s see what will happen when we finished executing the sleep function and try to execute it further.



- Also, note that we have **delayedTimer** function defined but we never called it. so, it’s still not part of callstack and also it’s passed as an argument to setTimeout.


JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
delayedTimer: fn()
Callstack
Global
Queues
Task Queue
Job Queue
Console
Just normal console

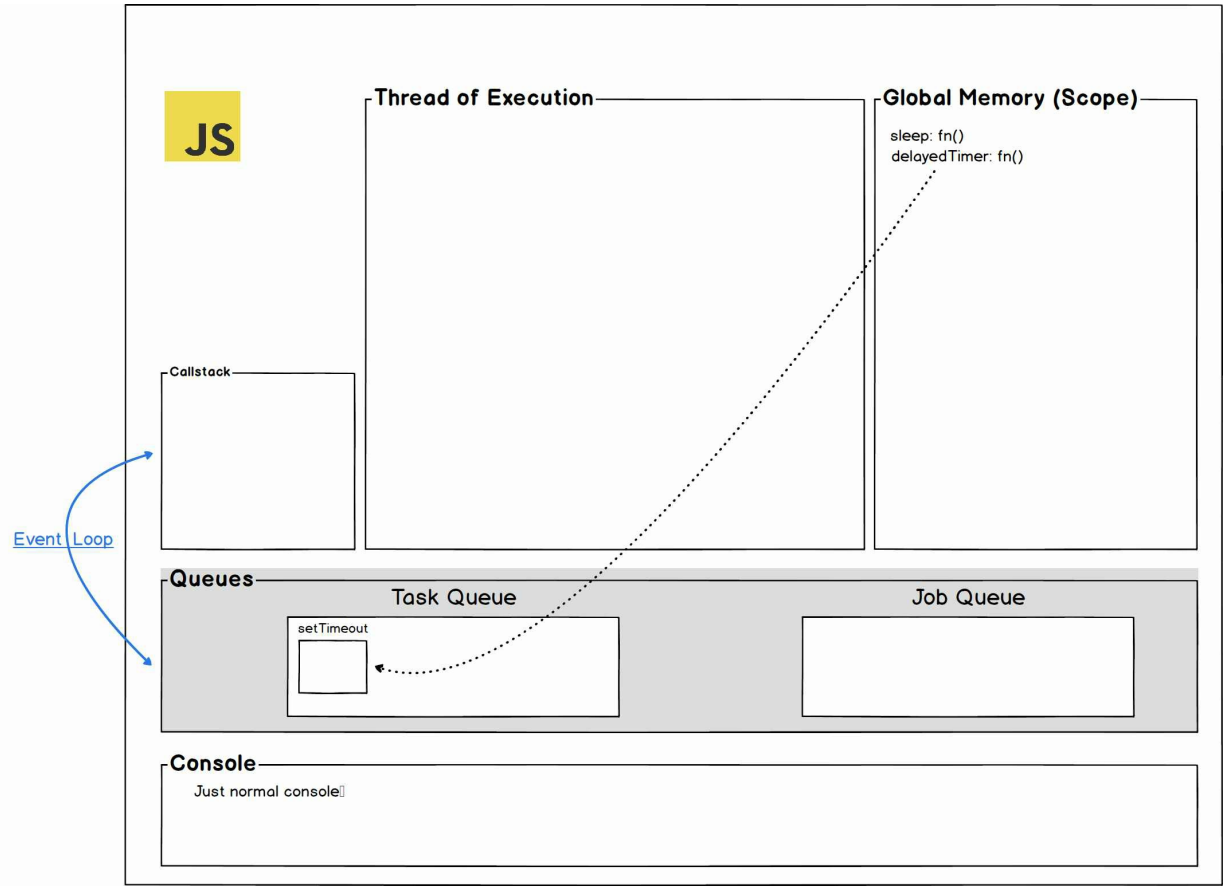

JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
delayedTimer: fn()
Callstack
Global
Queues
Task Queue
Job Queue
Console
Just normal console



- As per the above images, we can see that all the execution of callstack is done including global expressions and statements code.
- So in short, JavaScript will execute all synchronous code straight forward and all async operations are just delegated for later execution. However, All function references are still stored on global memory space, they are not copied to the task/job queue.

Time for introduction “Event Loop”!

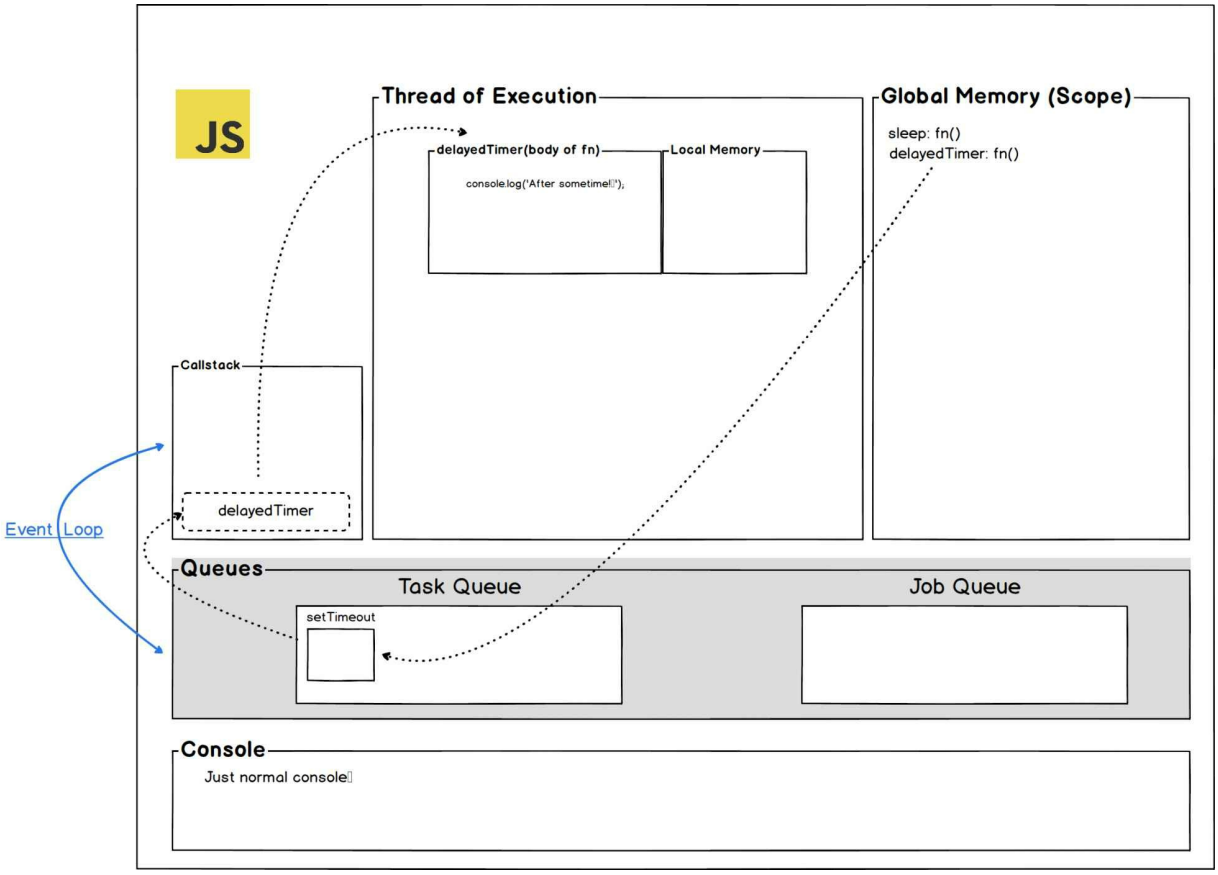
```
The event loop is a simple while loop, which keeps tracks that where any work is pending in queue/s to be finished or not with continuously checking that whether callstack is ready for taking the work from the queue/s or not.
```

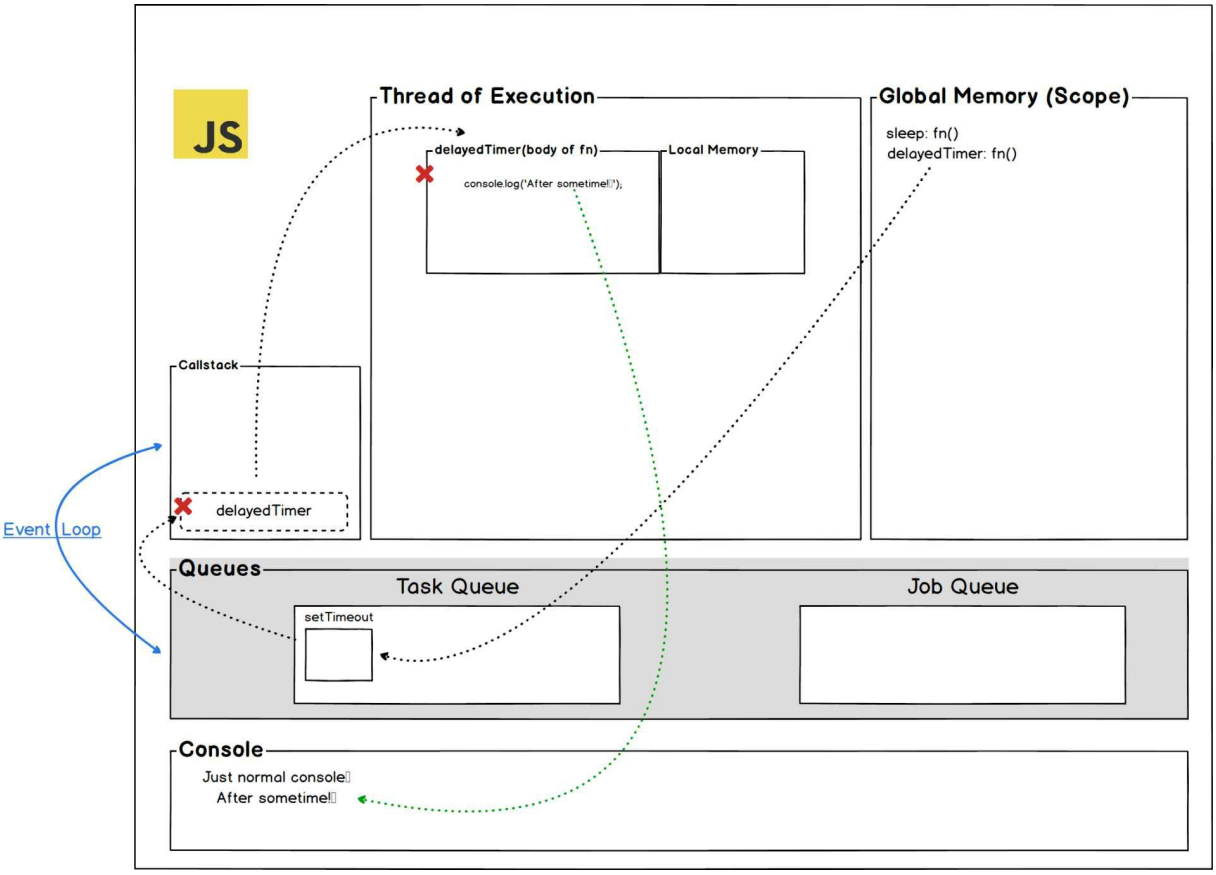


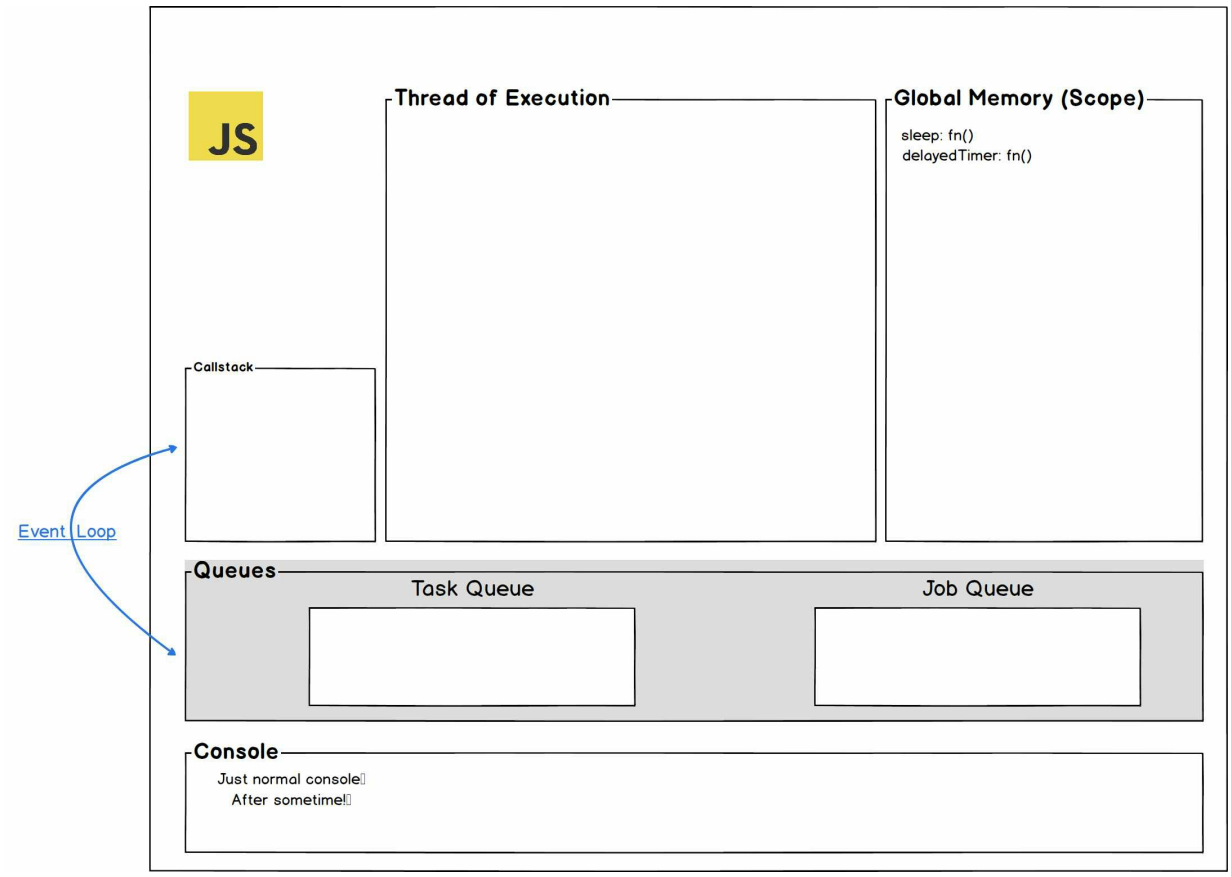
- Now because currently our callstack is empty and the queue has pending tasks to be finished, the event loop will find it. Note that **delayedTimer** actually made available to **task queue** after 2000 milliseconds(which we provided time, after how much time it’ll become part of task queue

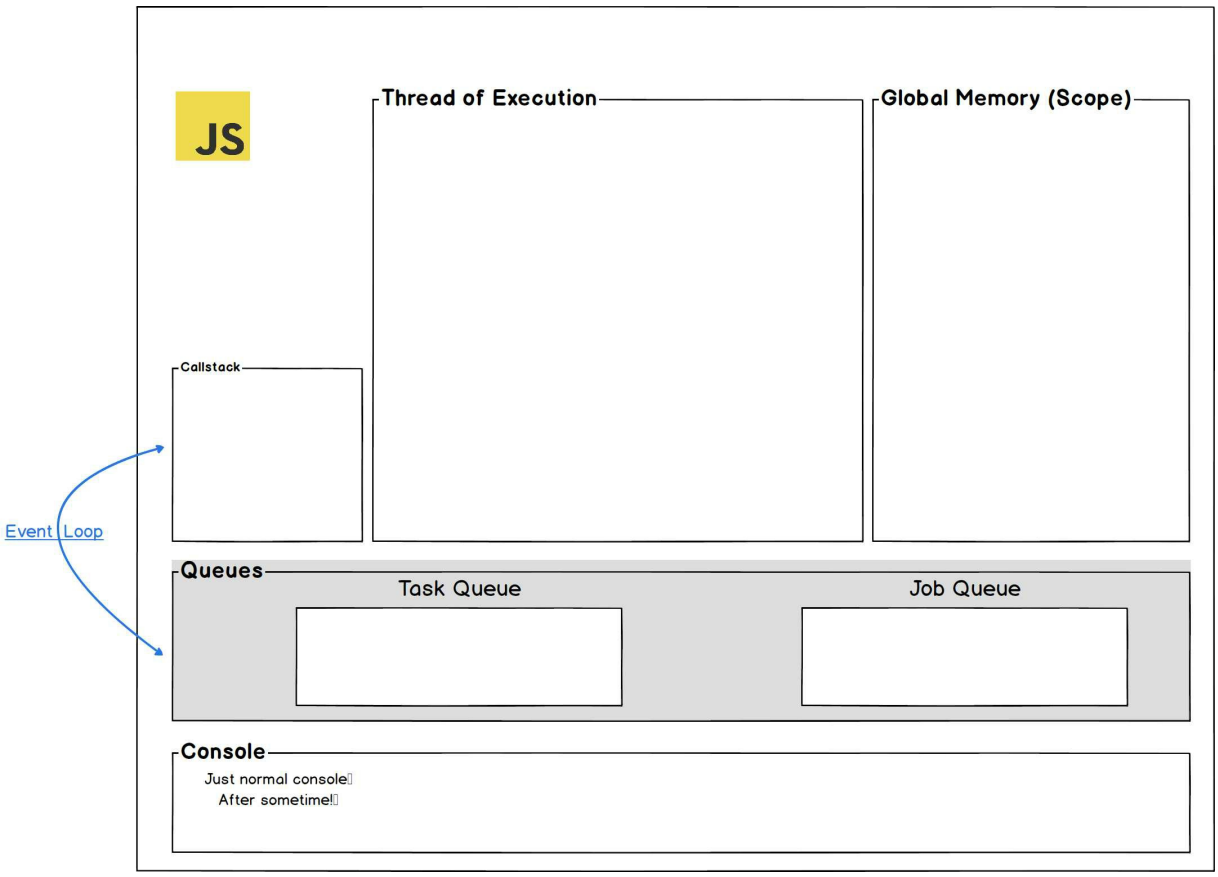
actually!)

- The event loop will have task only checking whether the work on queues are pending or not and which are those tasks.
- The function **delayedTimer** will be pushed to the **callstack** by the environment and become part of normal execution😀.




JS
Thread of Execution
delayedTimer(body of fn)
Local Memory
console.log('After sometime!');
Global Memory (Scope)
sleep: fn()
delayedTimer: fn()
Callstack
delayedTimer
Event Loop
Queues
Task Queue
setTimeout
Job Queue
Console
Just normal console
After sometime!


JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
delayedTimer: fn()
Callstack
Event Loop
Queues
Task Queue
Job Queue
Console
Just normal console
After sometime!



- There you have my friends, this is how async code is executed in JavaScript. There are little more to explore still. Promises and Async Await.
- In this chapter, I will only explain promises, because async-await requires little more knowledge that we will discuss in upcoming chapters.

Promise - An Aesthetic Async Task Resolver

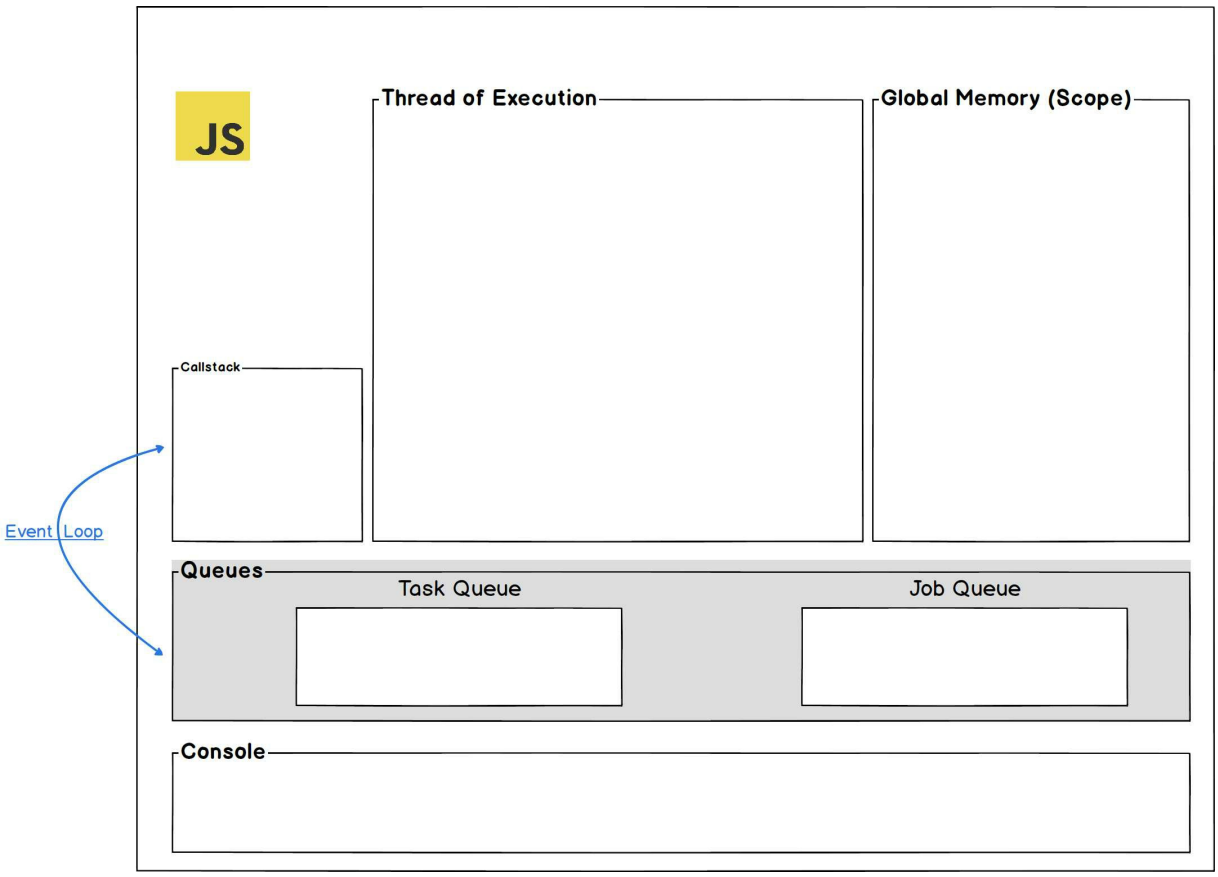
MDN

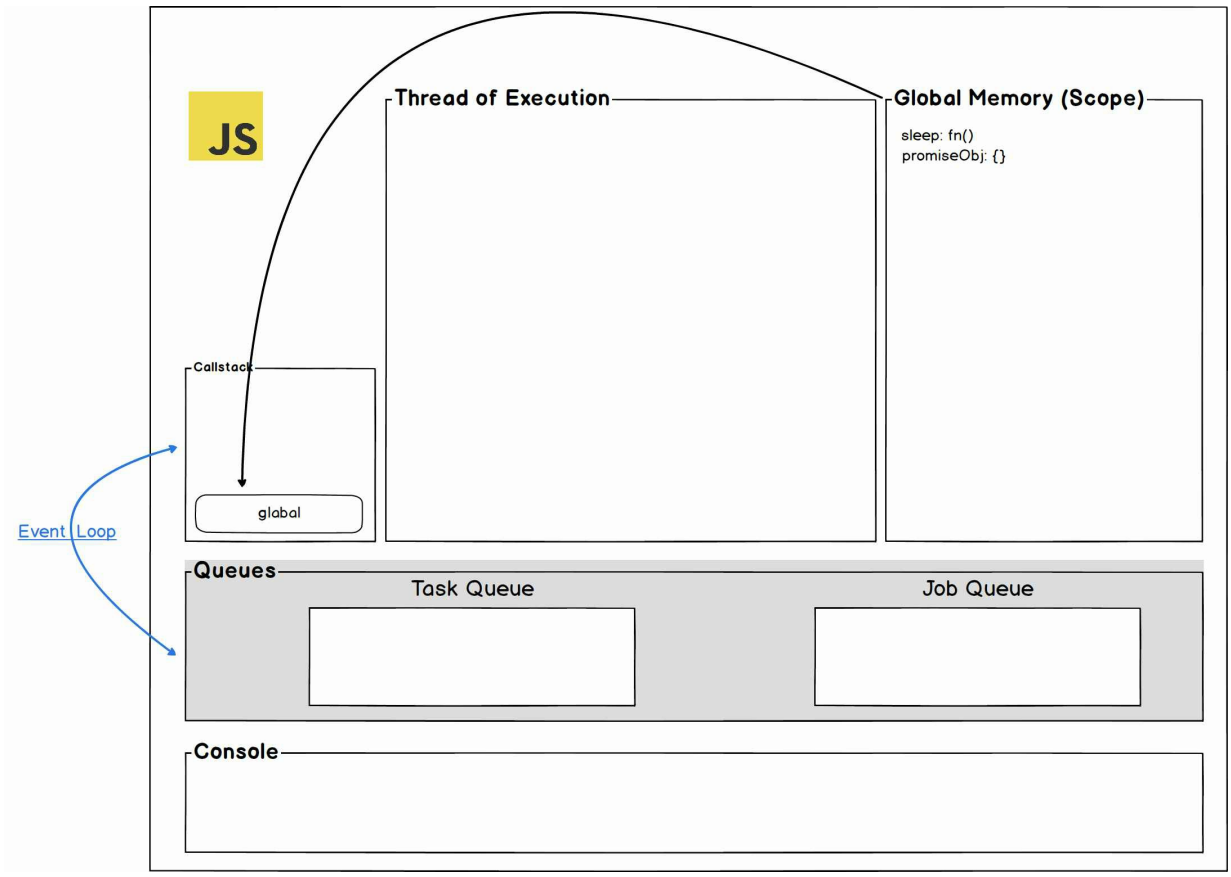
```
The Promise object represents the eventual completion (or failure) of an
asynchronous operation and its resulting value.
```

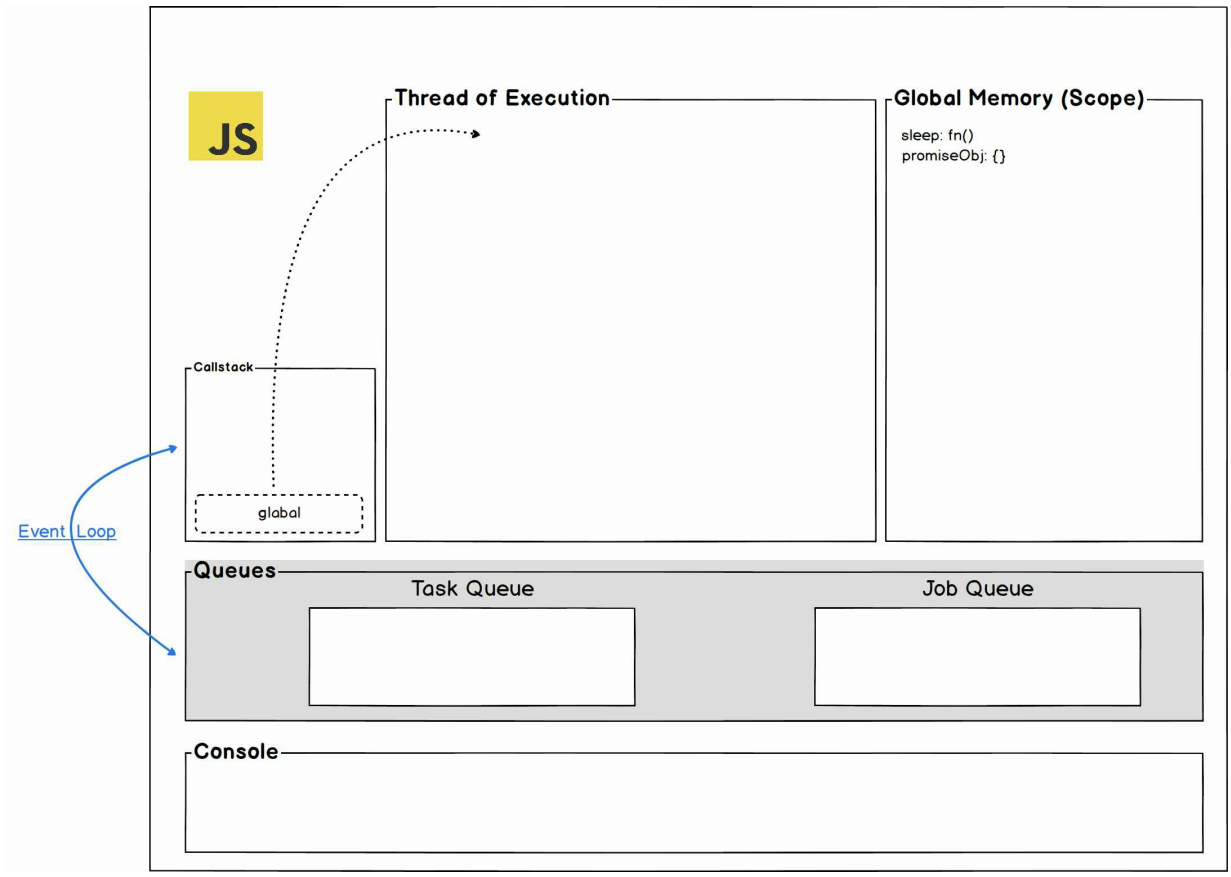
- Of course, it’s a short and simple definition but also not explaining what the hell actually happening under the hood!
- Let me tell unwill the promise more simply for you with a visual example.
- In the above example, I made a simple **sleep** function for simulating a synchronous blocking mechanism for given second/s. we will use this for our next example too.
- There are 2 things different things available to us, we have a **Promise** function constructor(notice capital letter) and **promise object.**
- **The Promise is a function constructor** that will create a **promise object** when we use a **new** keyword in front of the Promise constructor.
- let’s start with an example.

```
90
91  function sleep(sec) {
92      let start = new Date().getTime();
93      let expire = start + (sec*1000);
94      while (new Date().getTime() < expire) {}
95      return;
96  }
97
98  const promiseObj = new Promise((resolve, reject) => {
99      let num = Math.floor(Math.random()*10) + 1;
100
101     sleep(2);
102
103     if(num > 5) {
104         resolve(num);
105     } else {
106         reject(num);
107     }
108
109 });
110
```

- Now in the above code, as you can see we will be going to store the returned object of the **Promise function constructor** in the **promiseObj** variable**.**
- Note here we have made **conditional resolve and reject** calls based on whether **num > 5 or not**.
- **It's visualization time😊.**


JS
Thread of Execution
Global Memory (Scope)
Callstack
Event Loop
Queues
Task Queue
Job Queue
Console

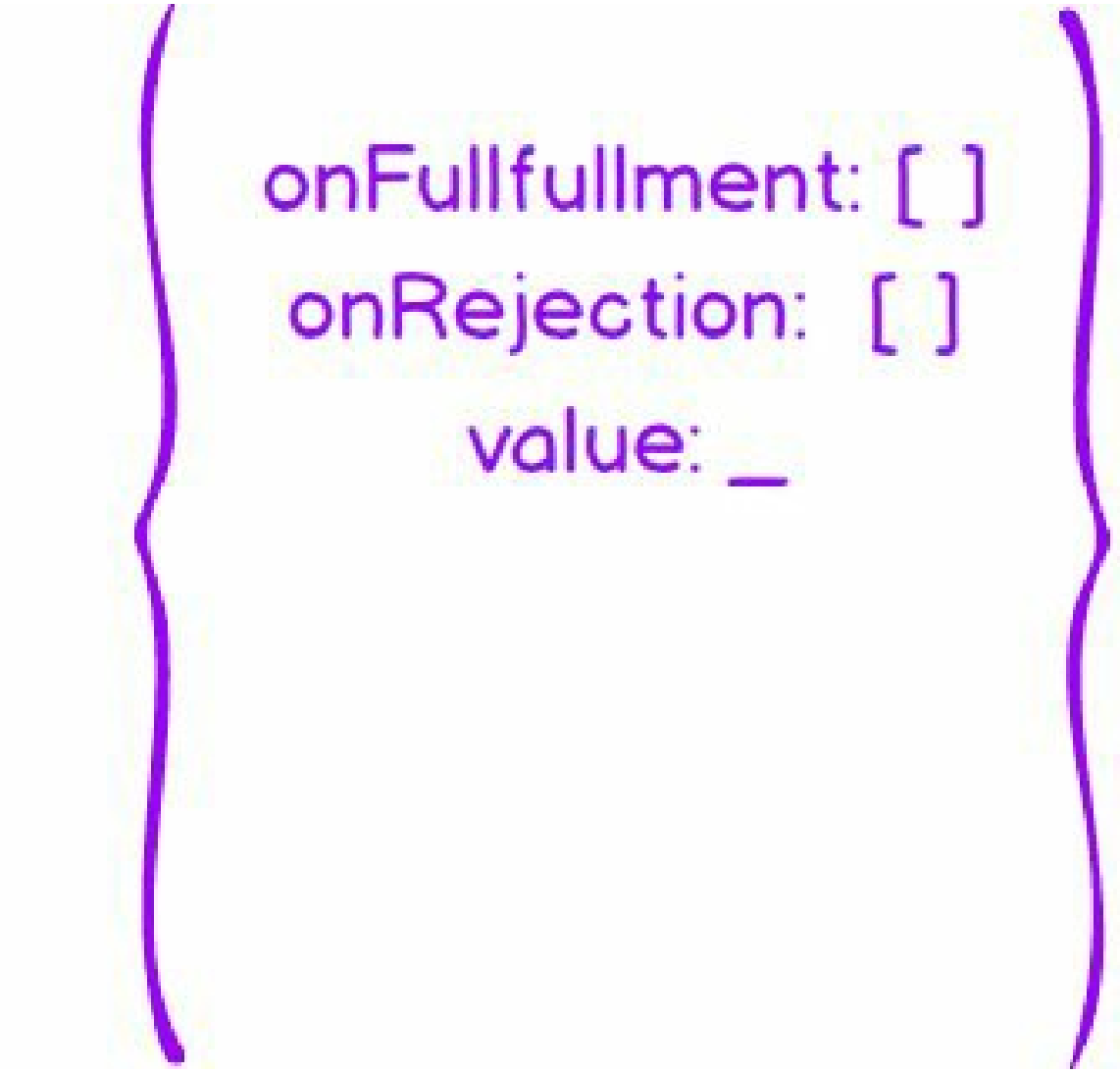

JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
promiseObj: {}
Callstack
global
Event Loop
Queues
Task Queue
Job Queue
Console



- Let’s see our code again so we don’t lose track of that code in our minds.

```
90
91  function sleep(sec) {
92      let start = new Date().getTime();
93      let expire = start + (sec*1000);
94      while (new Date().getTime() < expire) {}
95      return;
96  }
97  const promiseObj = new Promise((resolve, reject) => {
98      let num = Math.floor(Math.random()*10) + 1;
99      sleep(2);
100     if(num > 5) {
101         resolve(num);
102     } else {
103         reject(num);
104     }
105 });
106
107 promiseObj.then((val) => {
108     console.log(val)
109 }).catch((err) => {
110     console.error(err)
111 })
112
```

- As we can see, we have the **Promise** function constructor which will be used to create a promise object. but before we visualize everything, let me tell you what we will get back after executing that Promise function constructor.
- Let’s see which are properties and internal properties are given to us for the **promise object**.



- As per the diagram, we can see that the promise object has the main 3 key-value pairs(internal and hidden properties actually).
- Also, remember we have passed **resolved** and **reject** inside the original **Promise function constructor's first argument.**
- The return **promiseObj** has the main 3 important key-value pairs(hidden/internal)**.**

onFullfillment

- It’s an array, which is used as a placeholder for all **.then call’s function arguments.** All **then** methods are in order pushed into this array. (will visualize soon how😊).

onRejection

- It’s an array, which is used as a placeholder for **.catch call’s function arguments. catch** method is pushed into this array. (will visualize soon how😊).

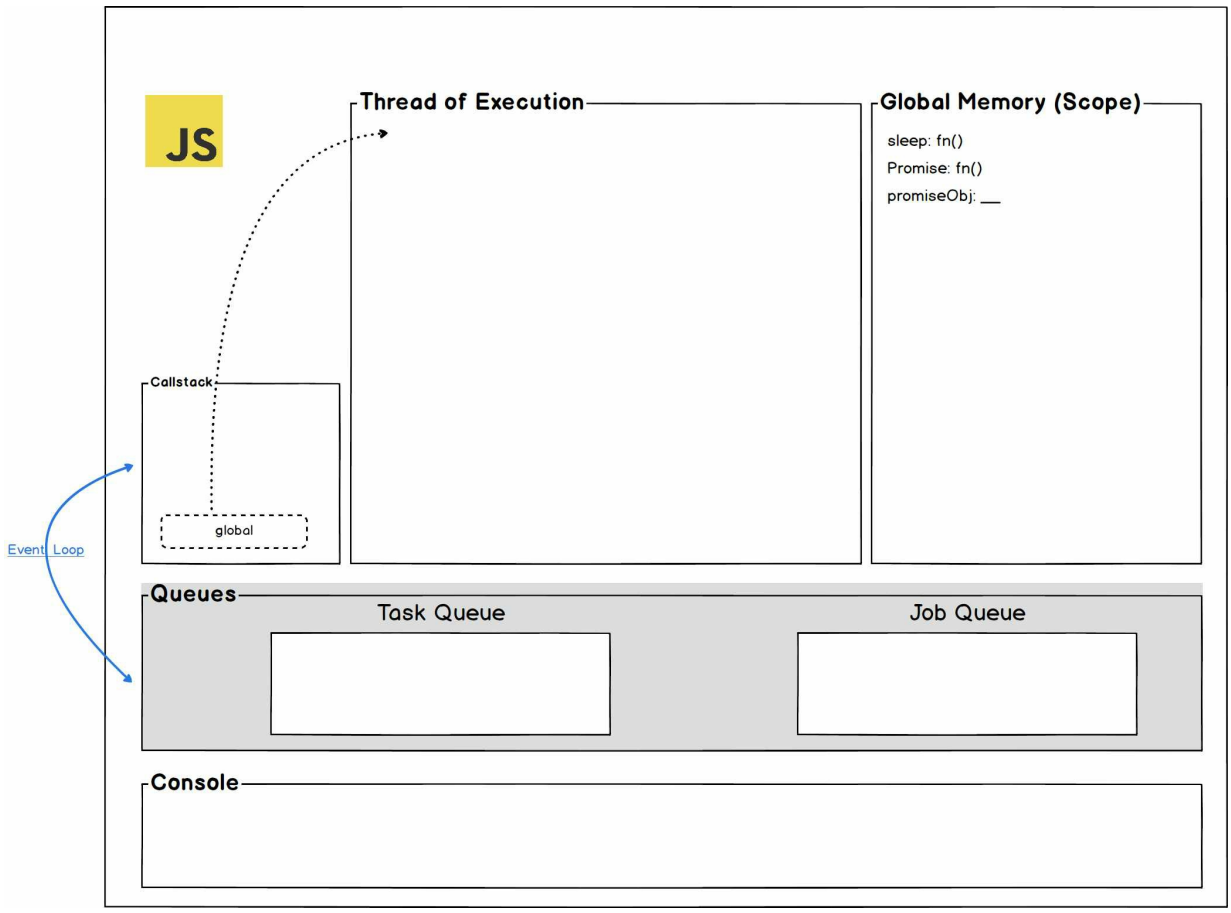
value

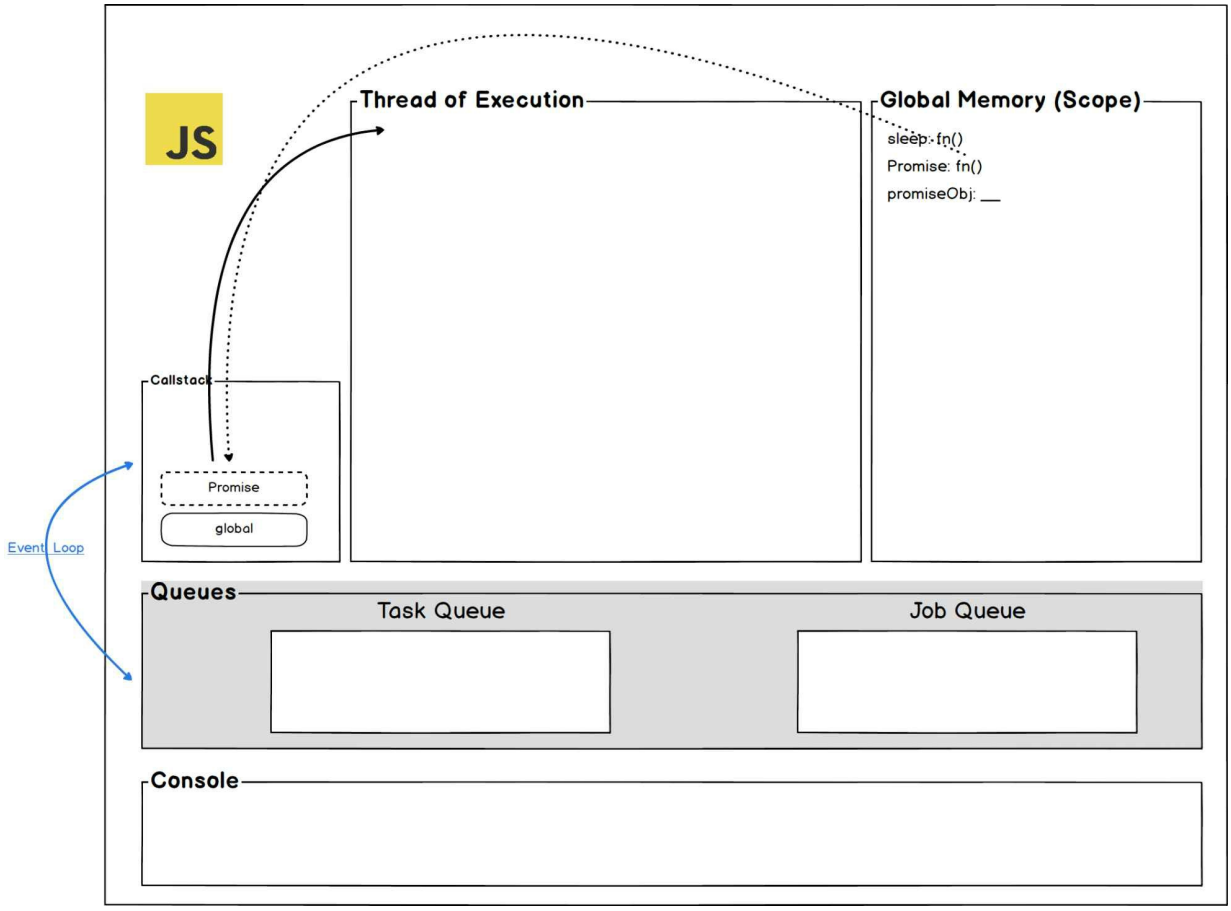
- It’s a value holder that will have the **resolved value or rejected value**.
- The importance of this resolve and **reject** functions are the mapping of value to the **value** key of the returned object(we will visualize it).

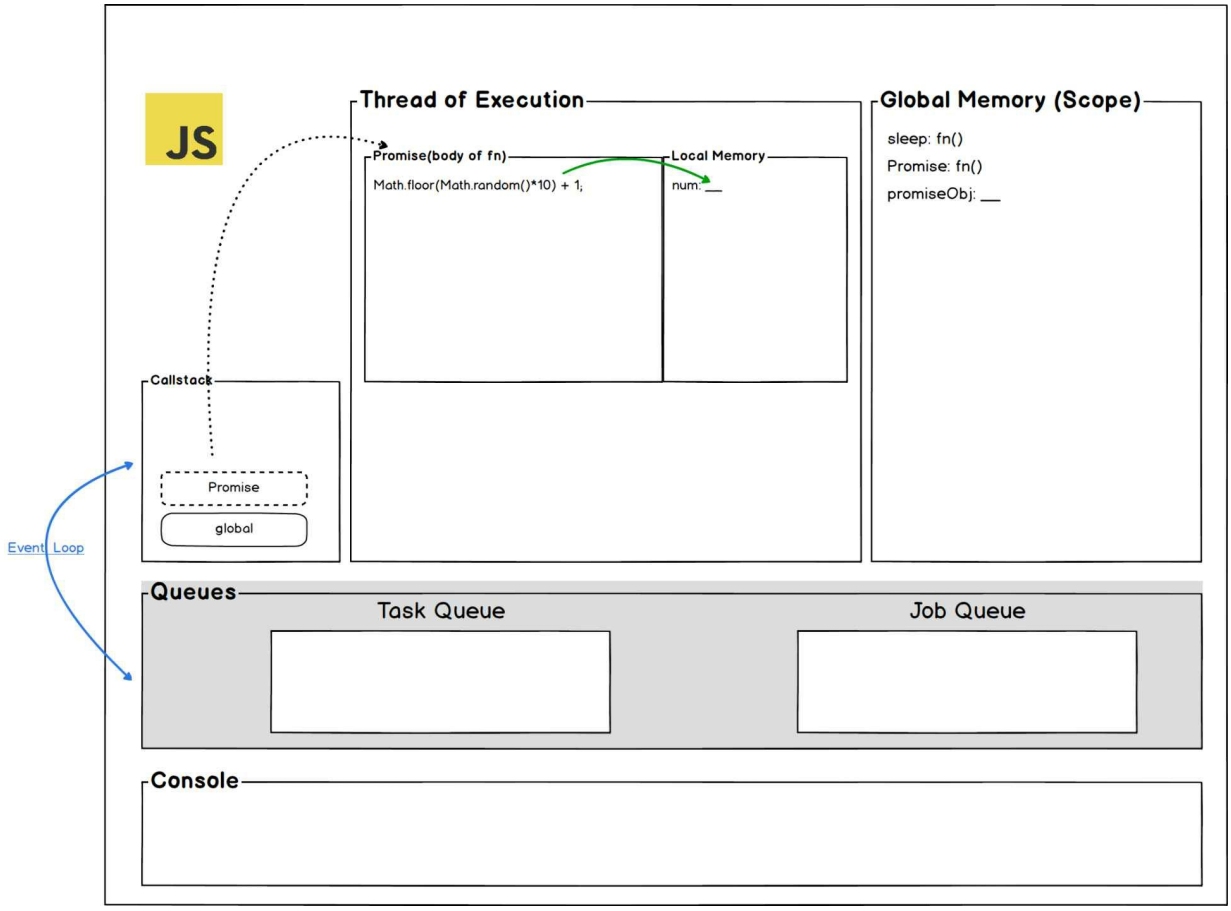
Let’s see how the below code snippet is working and how its(promiseObj’s) value is read.

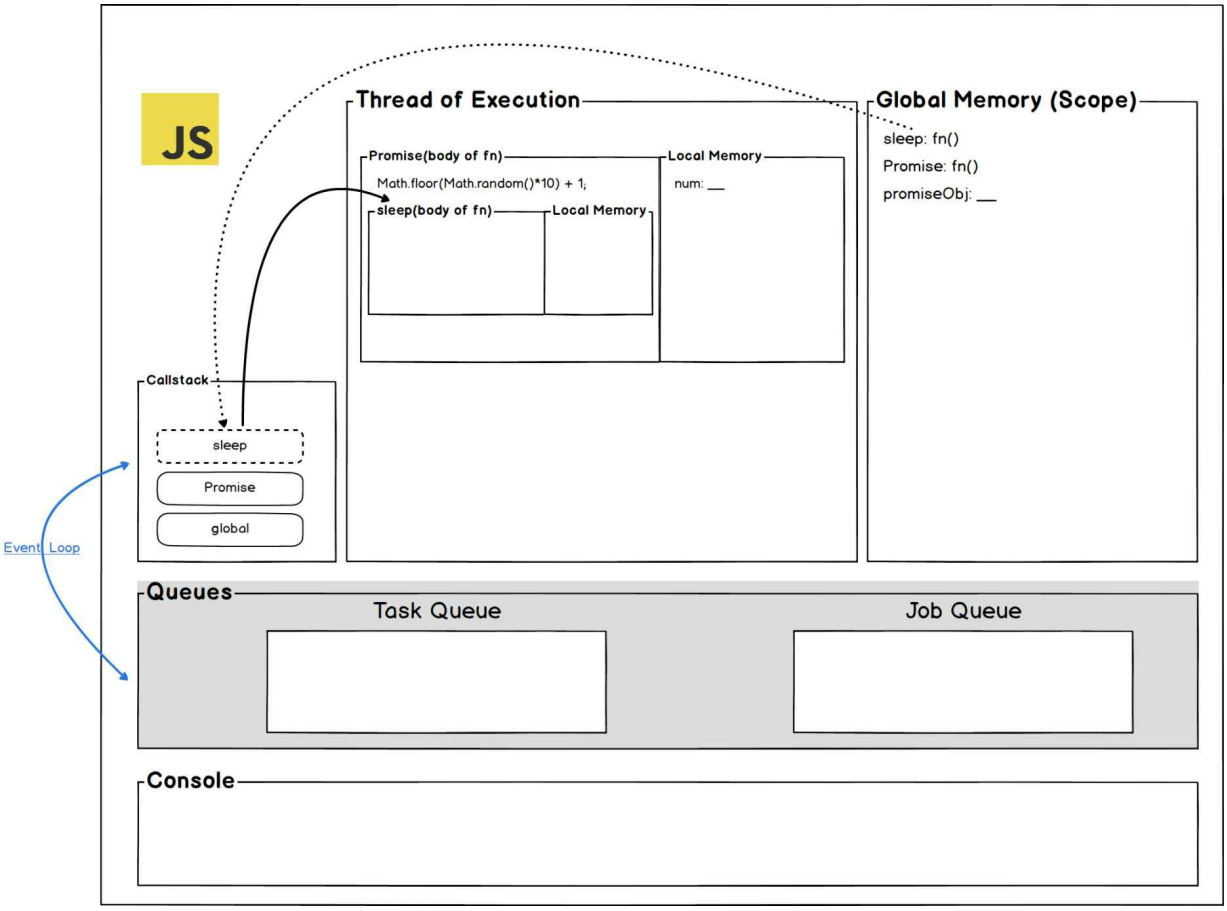
```
90
91  function sleep(sec) {
92      let start = new Date().getTime();
93      let expire = start + (sec*1000);
94      while (new Date().getTime() < expire) {}
95      return;
96  }
97  const promiseObj = new Promise((resolve, reject) => {
98      let num = Math.floor(Math.random()*10) + 1;
99      sleep(2);
100     if(num > 5) {
101         resolve(num);
102     } else {
103         reject(num);
104     }
105 });
106
107 promiseObj.then((val) => {
108     console.log(val)
109 }).catch((err) => {
110     console.error(err)
111 })
```

- Promise(It’s a function) constructor evaluation is actually a **synchronous** task.

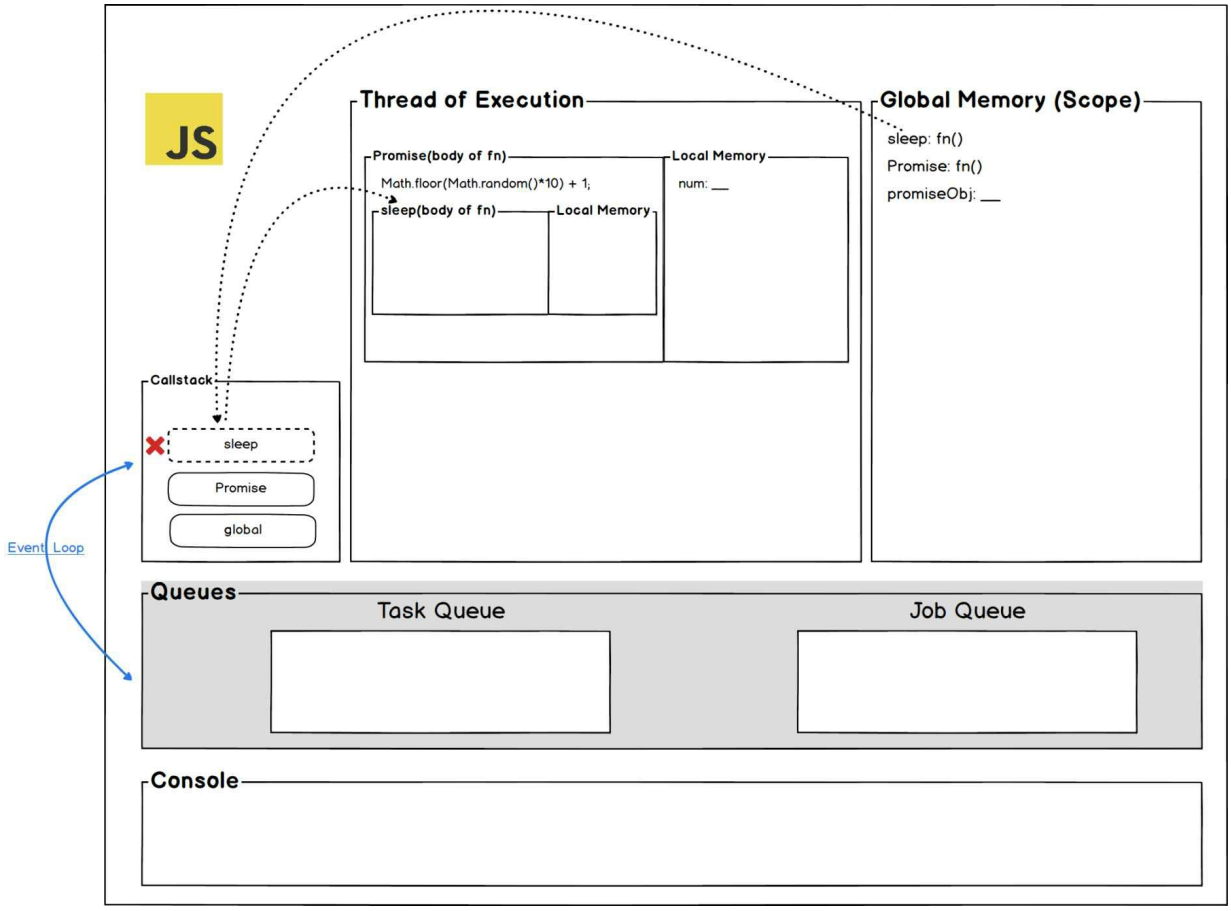

JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj: __
Callstack
global
Event Loop
Queues
Task Queue
Job Queue
Console

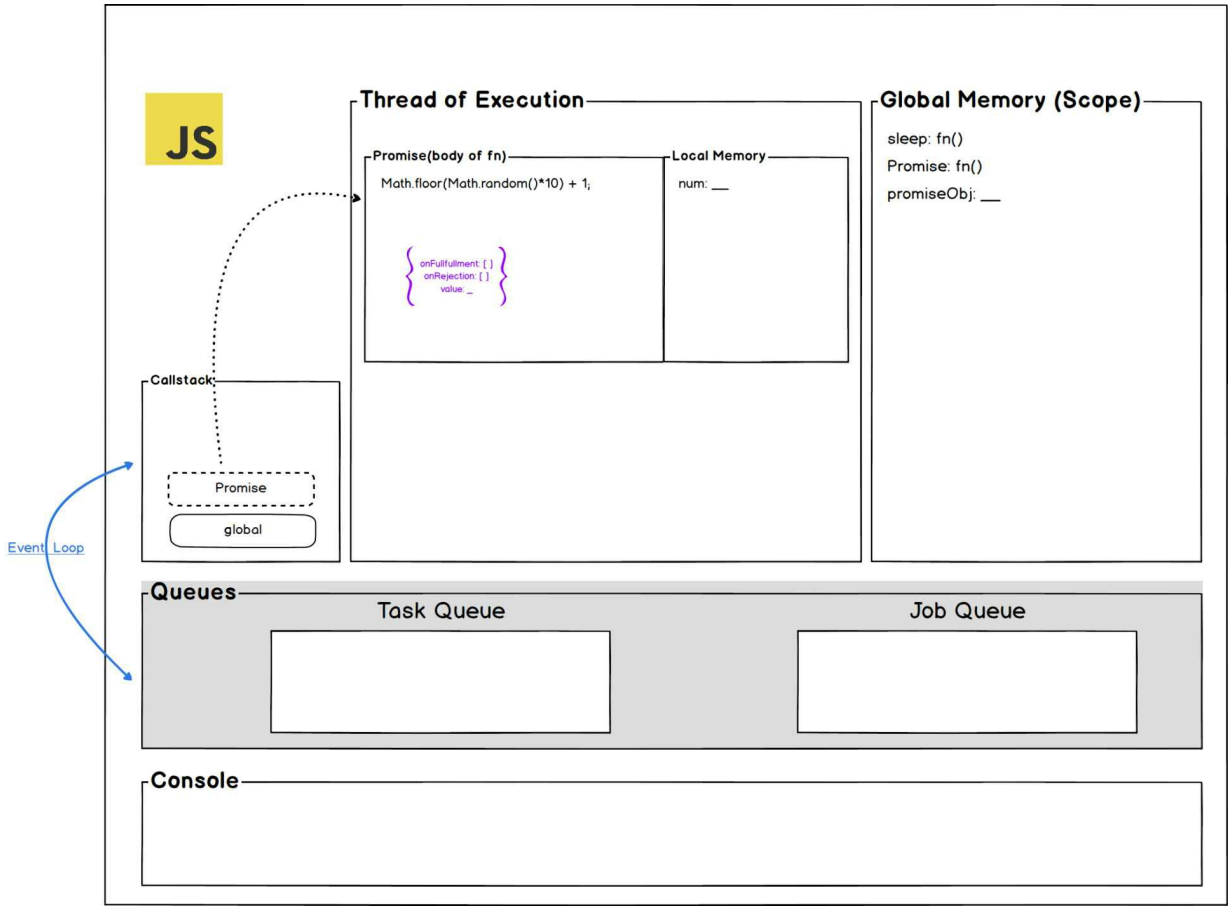

JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj: __
Callstack
Promise
global
Event Loop
Queues
Task Queue
Job Queue
Console


JS
Thread of Execution
Promise(body of fn)
Math.floor(Math.random()*10) + 1;
Local Memory
num: __
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj: __
Callstack
Promise
global
Event Loop
Queues
Task Queue
Job Queue
Console



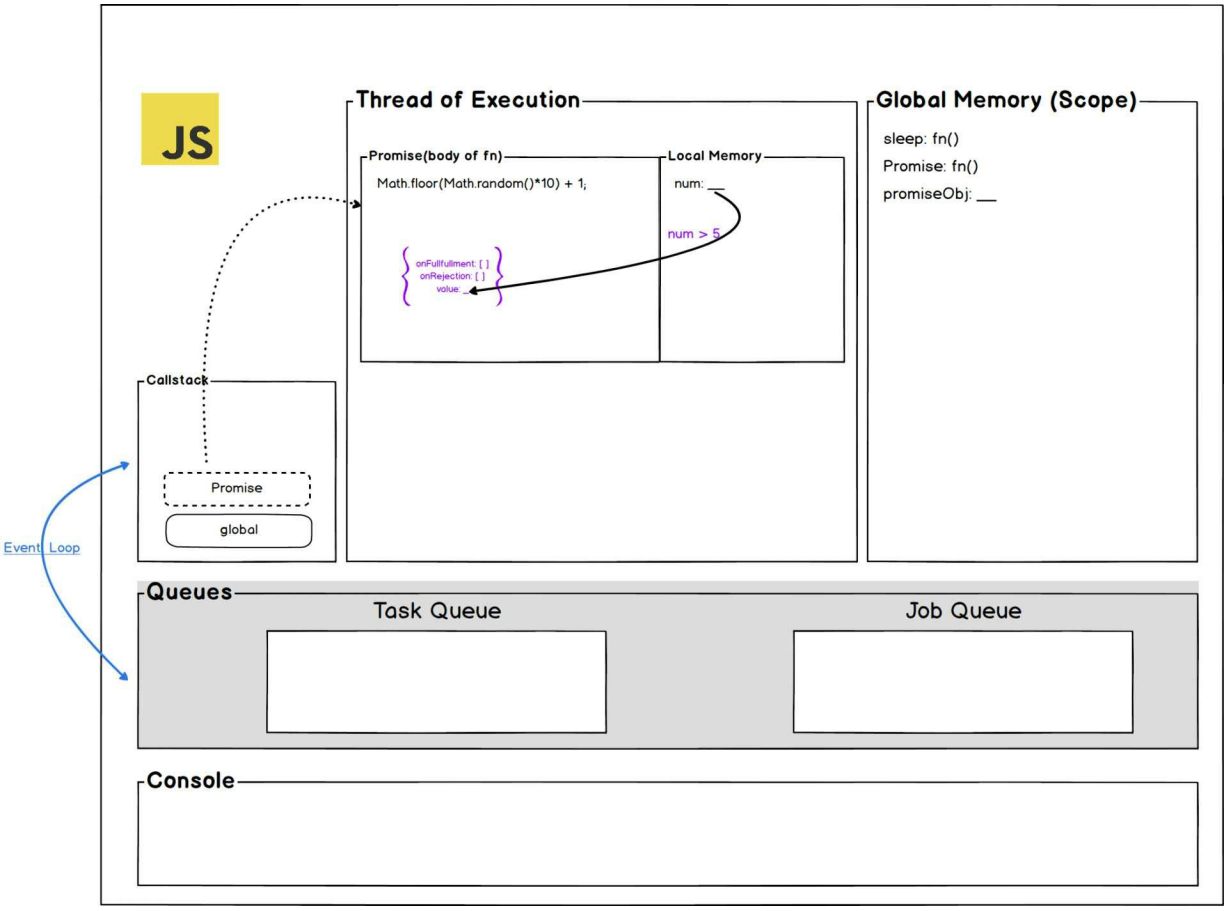
- We will see how the **new** keyword works under the hood, but here let me tell you quickly a **new** keyword creates an object, and the function constructor will determine what properties will be available inside that object.
- let’s just keep visualizing☺.


JS
Thread of Execution
Promise(body of fn)
Math.floor(Math.random()*10) + 1;
sleep(body of fn)
Local Memory
Local Memory
num: __
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj: __
Callstack
sleep
Promise
global
Event Loop
Queues
Task Queue
Job Queue
Console

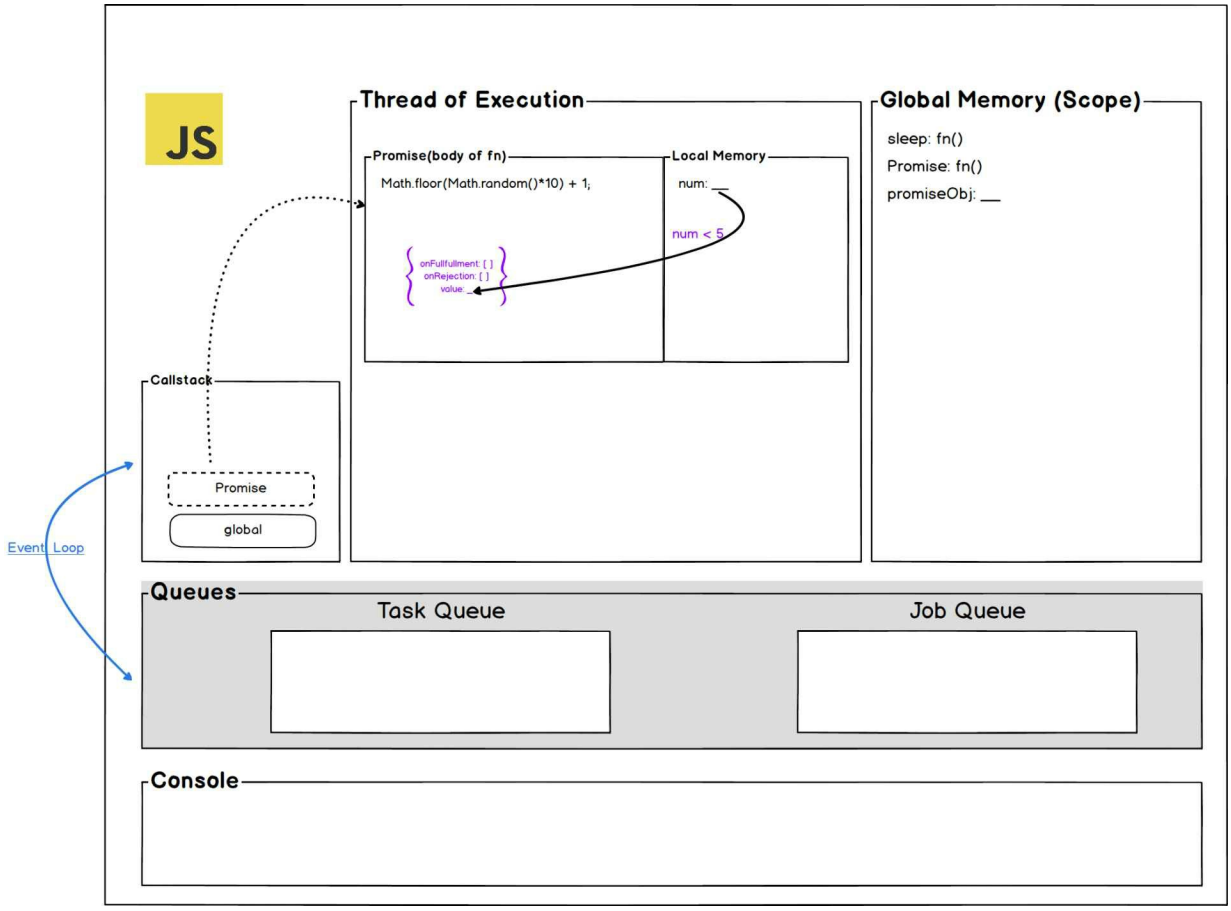



```
Now we have 2 possibilities over here.
```

- if we get **num > 5**, then we are going to call **resolve** and will pass **num** as an argument to resolve**.**



- if we get **num < 5**, then we are going to call **reject** and will pass **num** as an argument to reject**.**

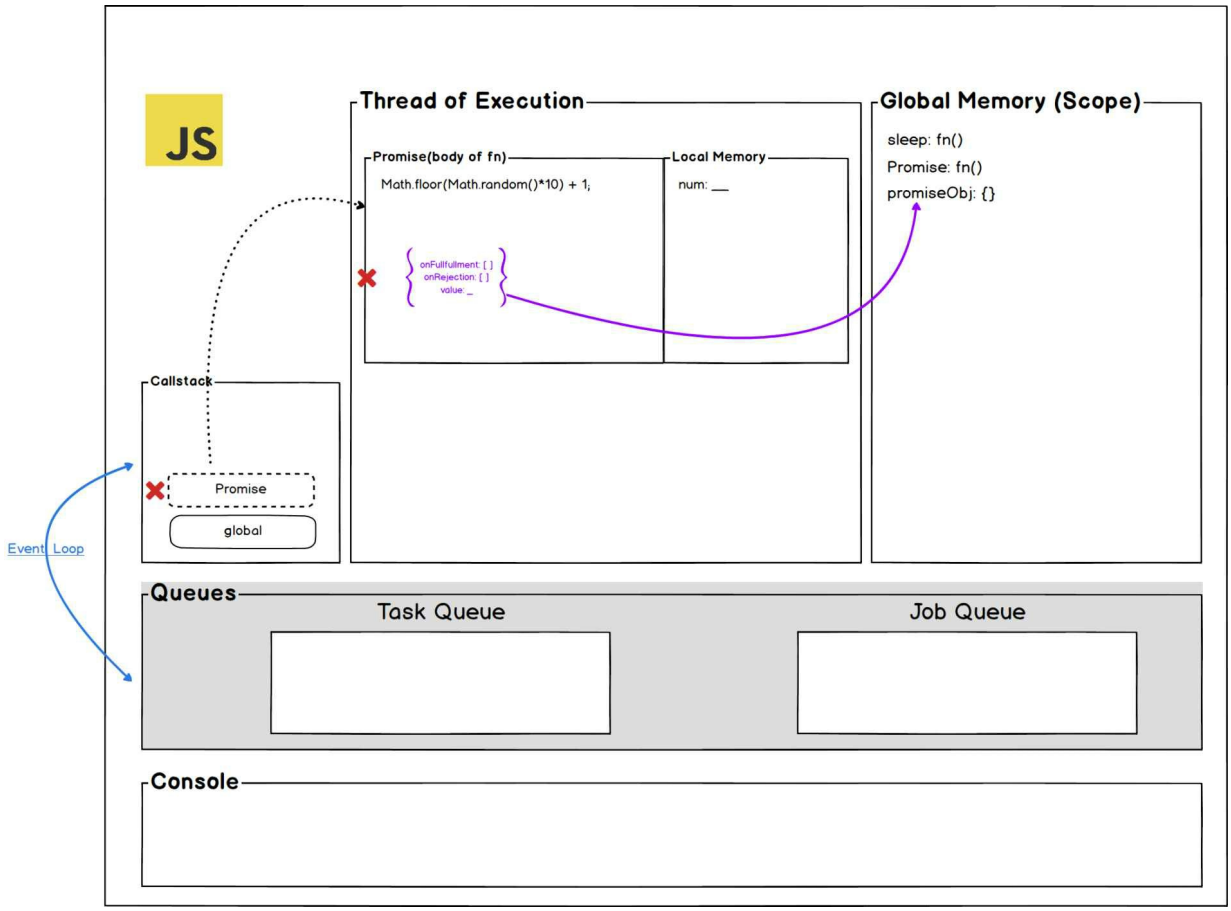


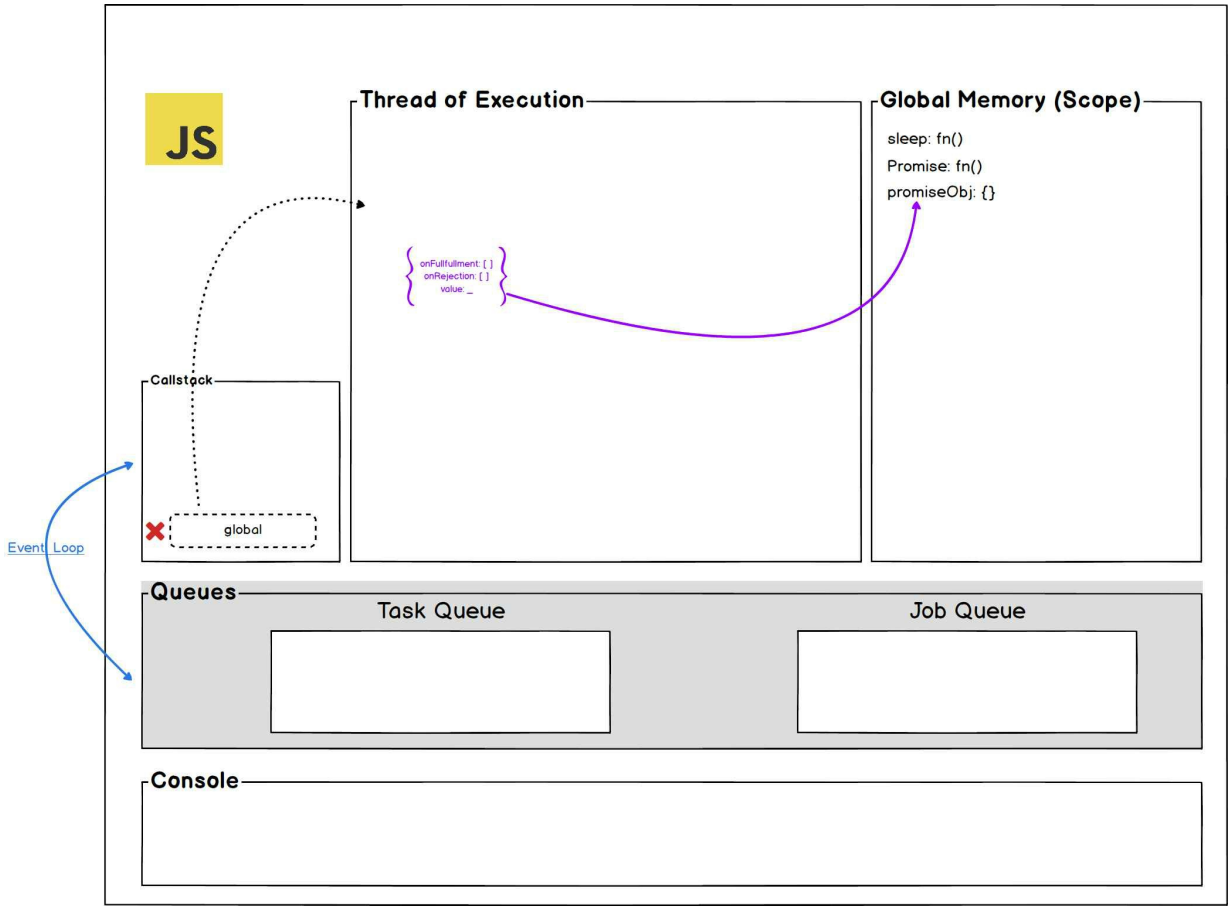
Here in both cases, as you can see it's going to be stored in the **value property of the promise object.**

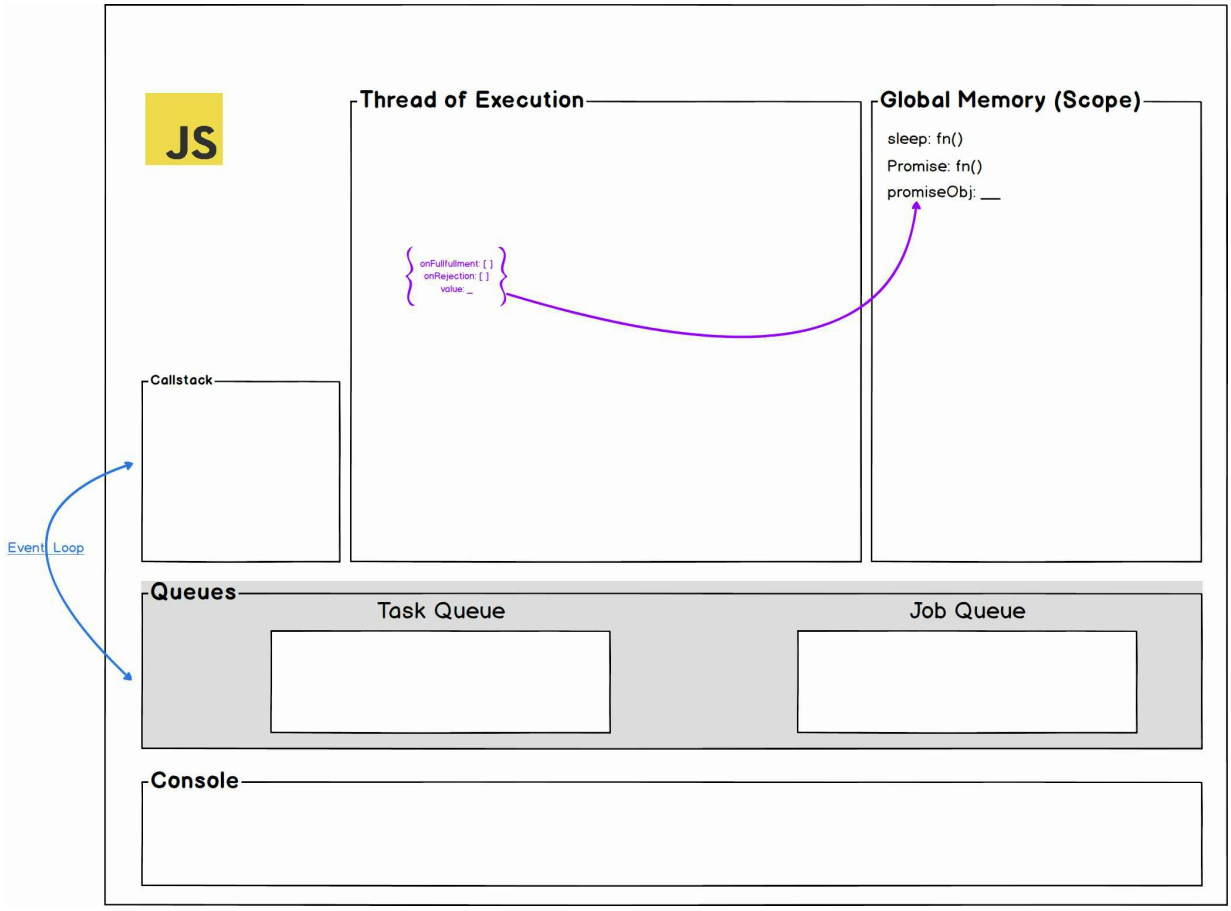
- Now, we can see promiseObj(the label given to returned object) is actually available on currently working scope and whatever state(resolved or rejected) of promise will have, is actually going to be stored inside promiseObj's internal properties and also **value** will be a placeholder for resolved/rejected value. But, **to access resolved value or rejected value we need some kind of mechanism**.
- That's why we have **then and catch** methods available to us. both methods are available to us via **__proto__**(will understand protoType

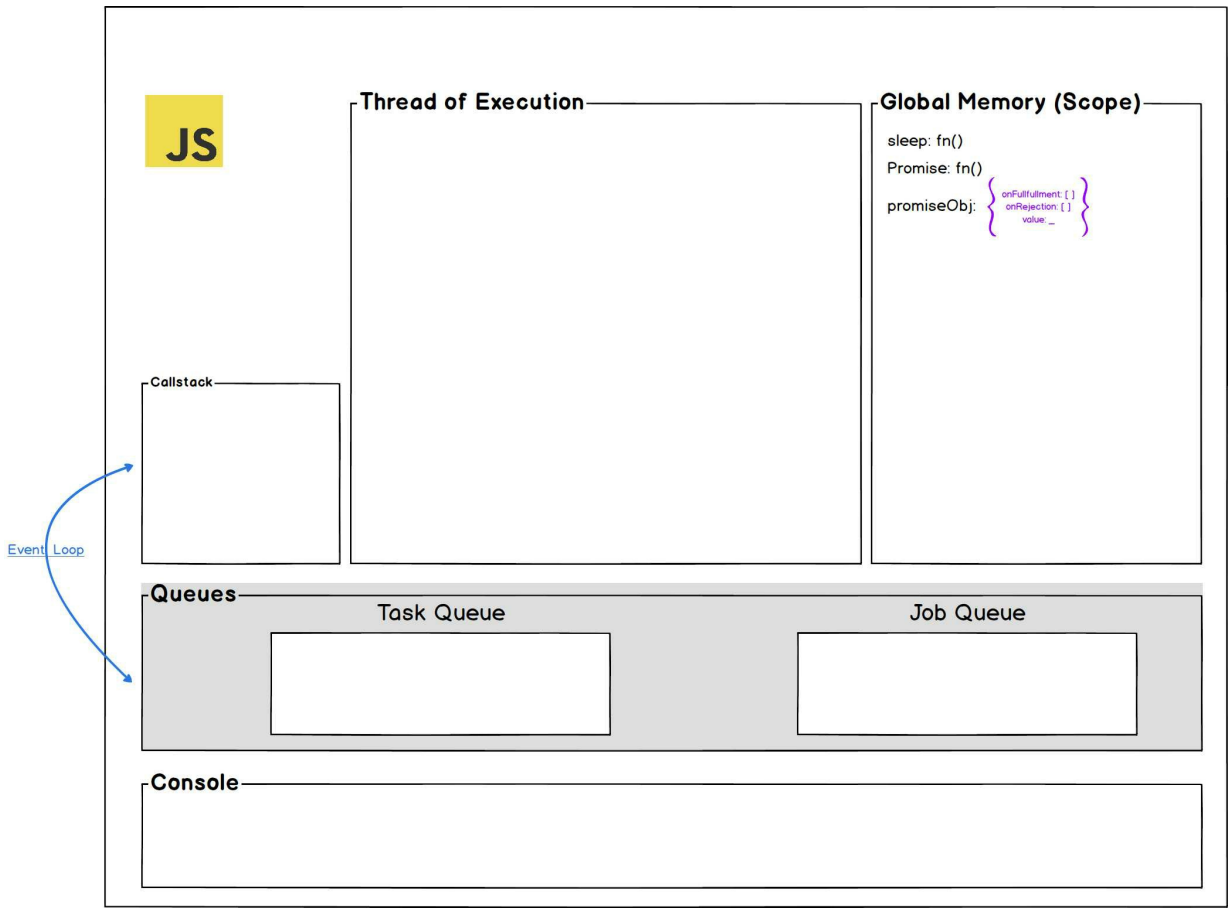
and chaining in later chapters in details).

- We are missing one part here, **onFullfillment** and **onRejection** are arrays and how it works actually!
- Here, after evaluating the value inside a promise object, work for the **Promise function constructor is done.** (for both resolve and reject scenarios).
- Let's first visualize that part.




JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj: {}
onFullfullment: [ ]
onRejection: [ ]
value: _
Callstack
global
Event Loop
Queues
Task Queue
Job Queue
Console


JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj: __
onFullfullment: [ ]
onRejection: [ ]
value: _
Callstack
Event Loop
Queues
Task Queue
Job Queue
Console



Woooh, that was a lot. but we still have to figure out for that **then and catch methods**😂**!**

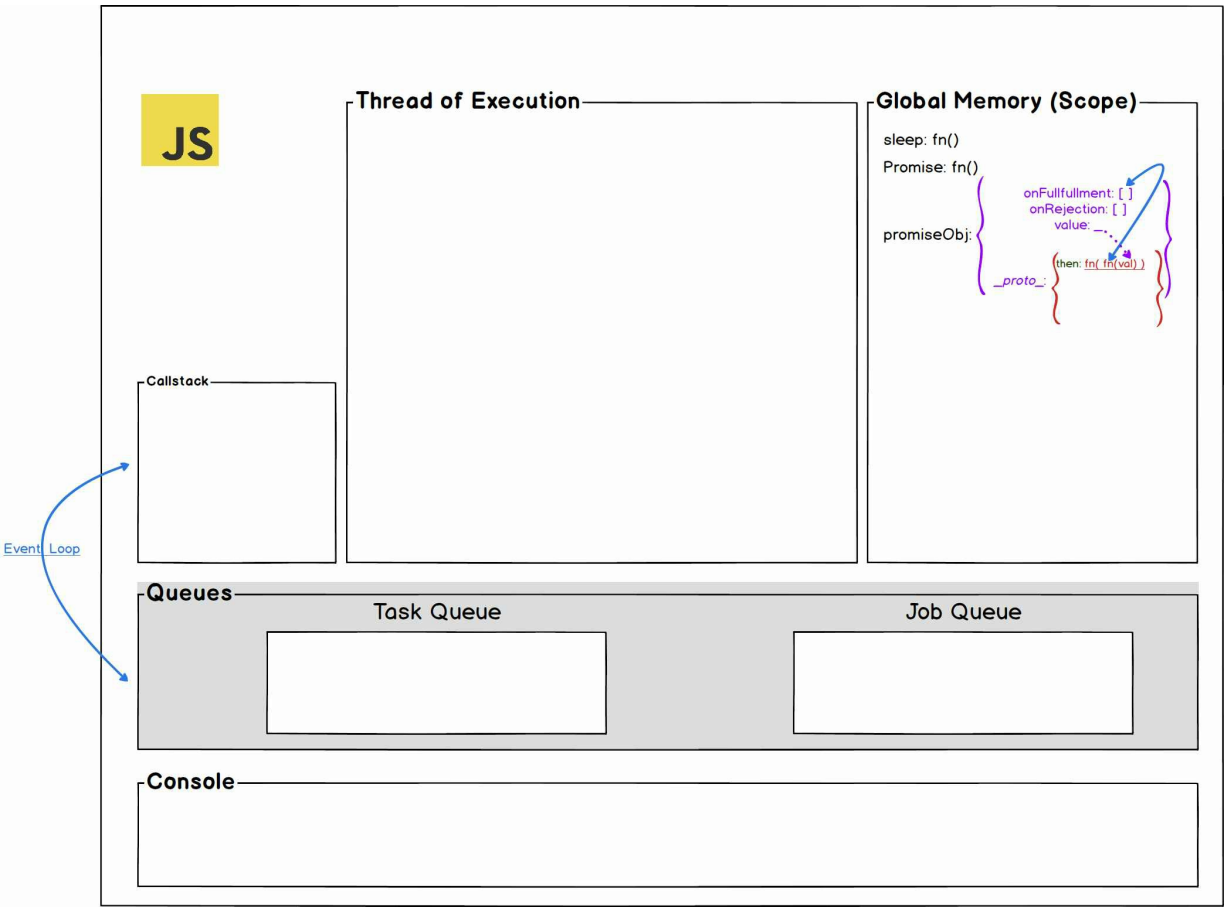
- One thing I didn't tell you that, whenever we write **then or catch method** inside the regular global context, these methods are actually made available via reference to the onFullfillment or onRejection key of promiseObj while it's creation time(promiseObj creation time).
- Not only that but whenever we evaluate the value, the value will be available to then function's argument(function as an argument) or catch function's argument.
- let's see both scenarios!

```
Resolve function scenario
```

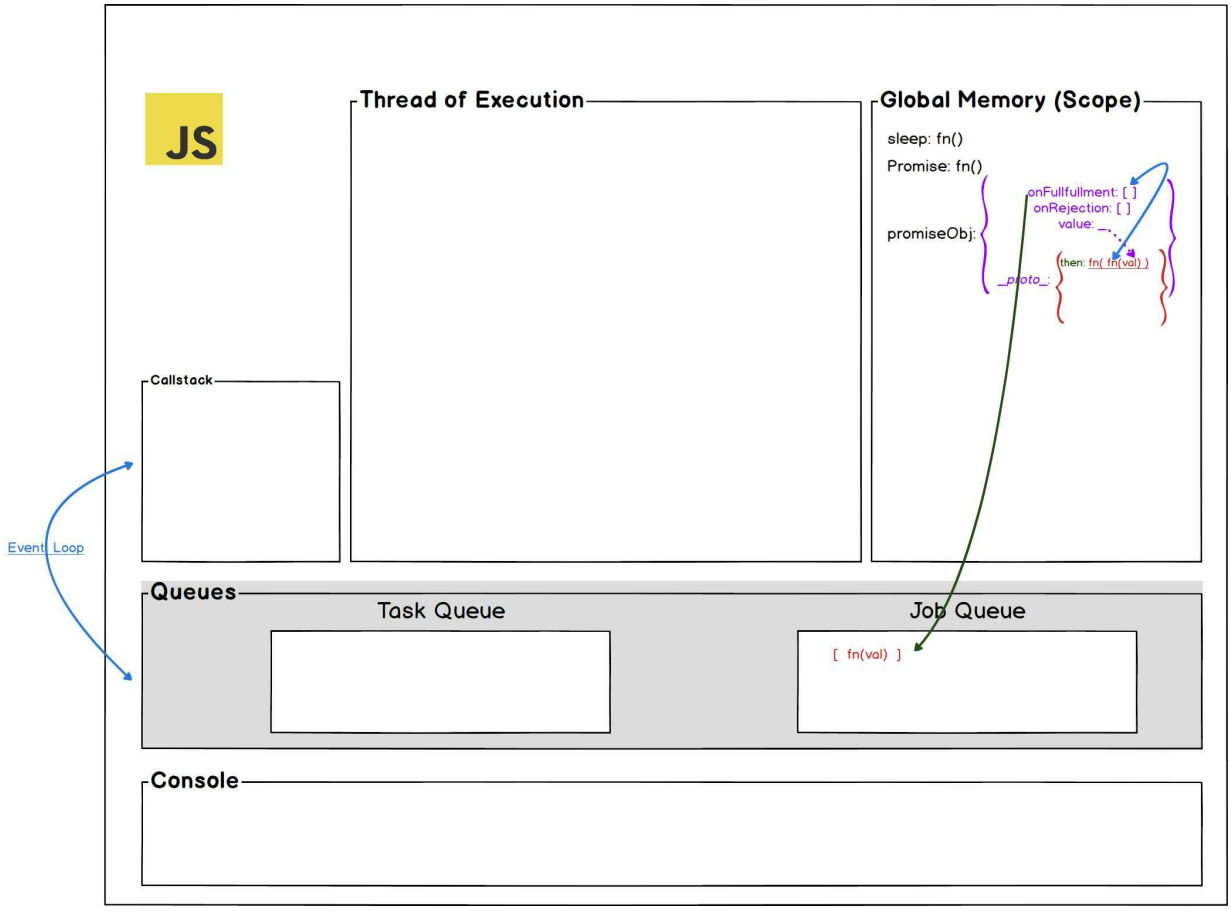
```
> function sleep(sec) {
      let start = new Date().getTime();
      let expire = start + (sec*1000);
      while (new Date().getTime() < expire) {}
      return;
  }
  const promiseObj = new Promise((resolve, reject) => {
      let num = Math.floor(Math.random()*10) + 1;
      sleep(2);
      if(num > 5) {
          resolve(num);
      } else {
          reject(num);
      }
  });

  promiseObj.then((val) => {
      console.log(val)
  }).catch((err) => {
      console.error(err)
  })
  8
<- ▼Promise {<fulfilled>: undefined}
    ▶__proto__: Promise
     [[PromiseState]]: "fulfilled"
     [[PromiseResult]]: undefined
```

- There is an illusion is given to us, and illusion is **then** method😉! so, the code which looks like we are executing on returned promiseObj is actually not run by **then** method. It's just a function that will act as a placeholder for another function. so, whatever function we pass as an argument to then method **(here (val) => { console.log(val) }) will be pushed to the onFullfillment array** of promiseObj. Also, I had not shown this before because I didn't want to confuse it with other visualization. A then method is actually accessible via __proto__ of a

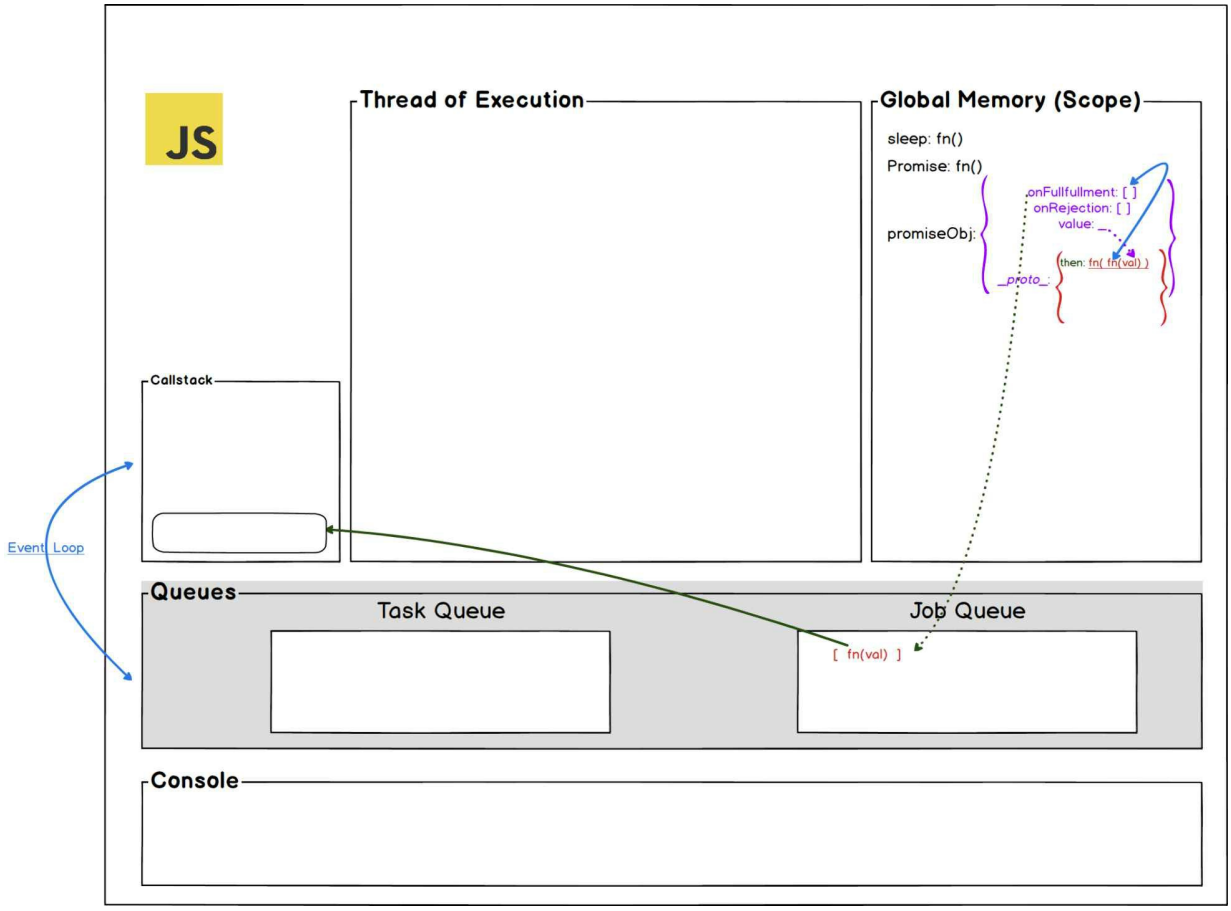
promiseObj. let’s visualize it😎.

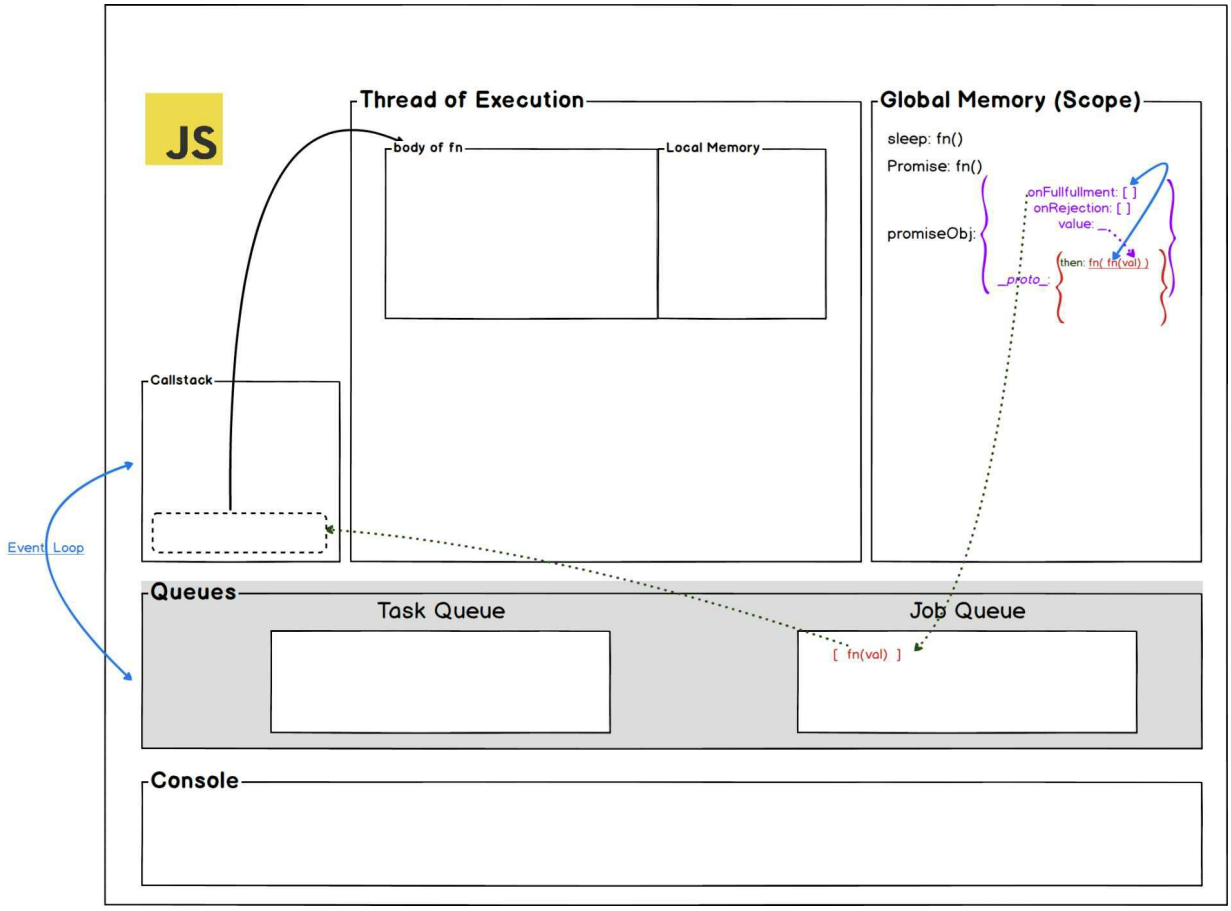


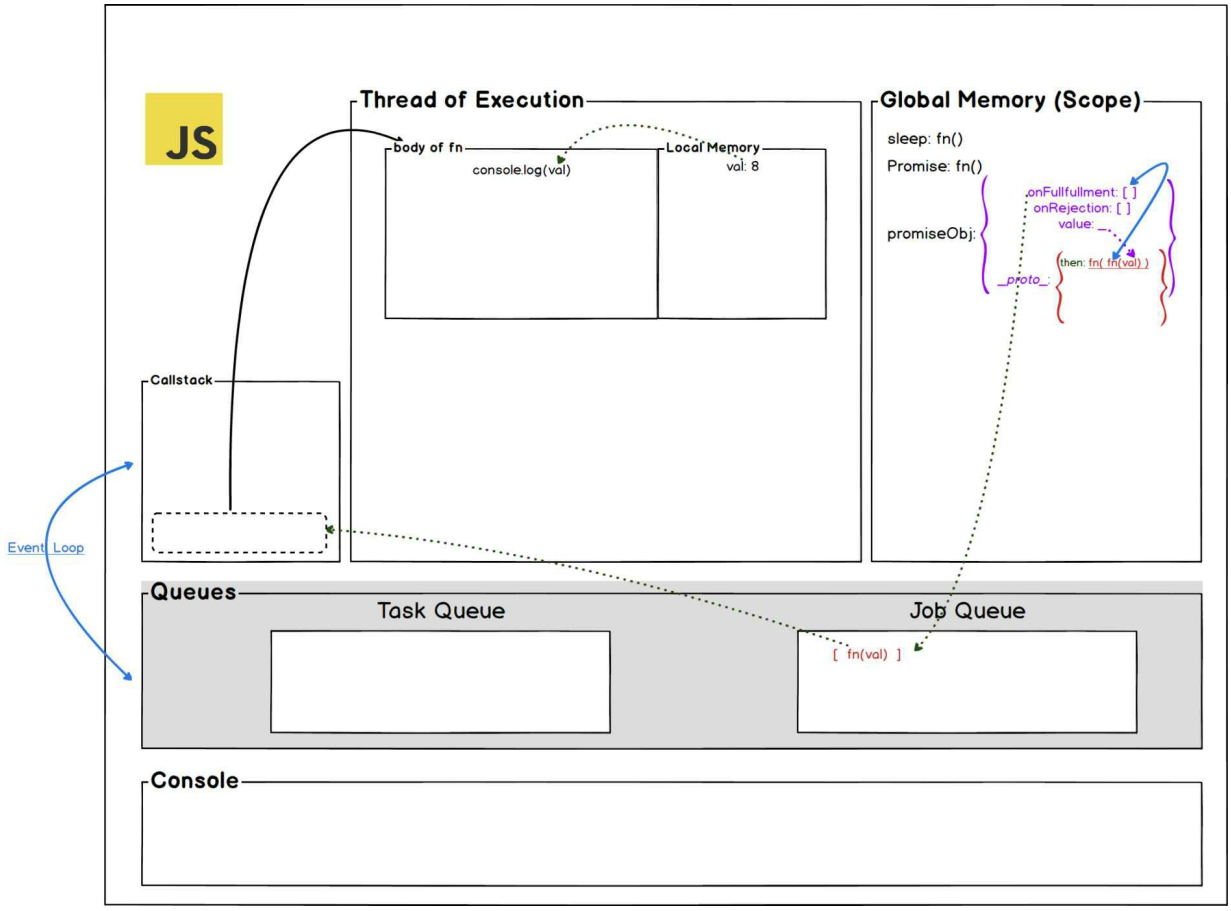
- Remember callstack became empty after all synchronous code finished its execution.
- The function of **then’s** argument will be added to onFullfillment and whenever it finishes execution it will be added to the **job queue** and later from **job queue to callstack.**

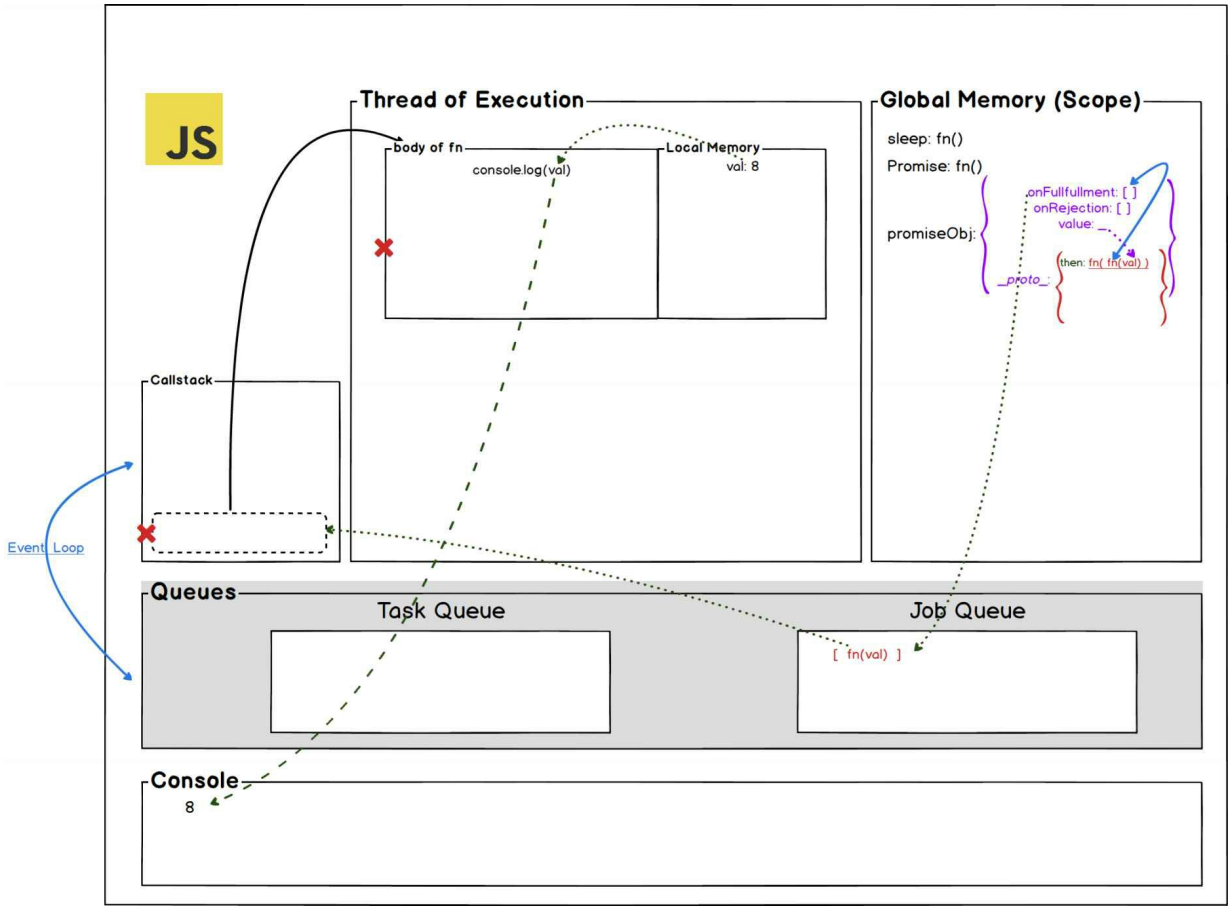


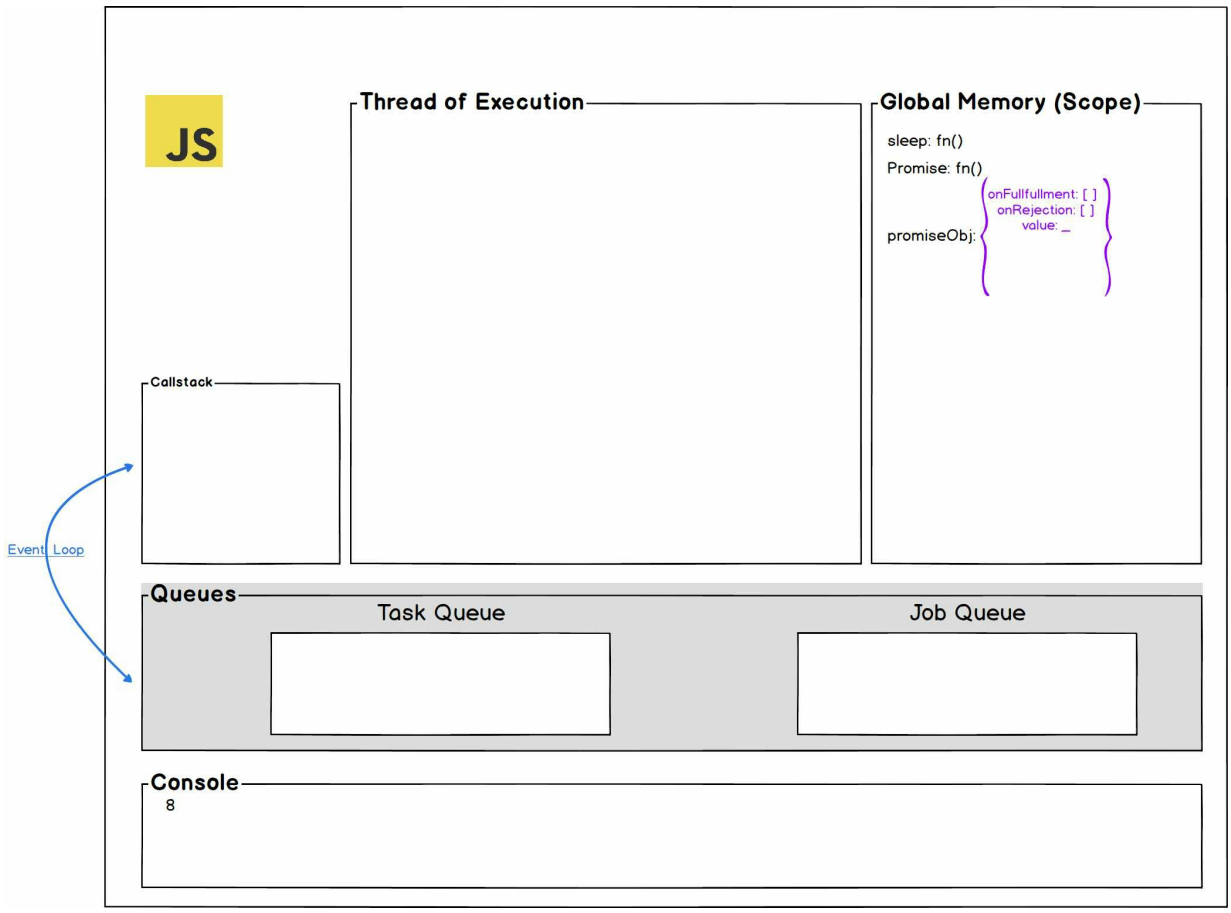
- Here function has no name (Anonymous function😊)

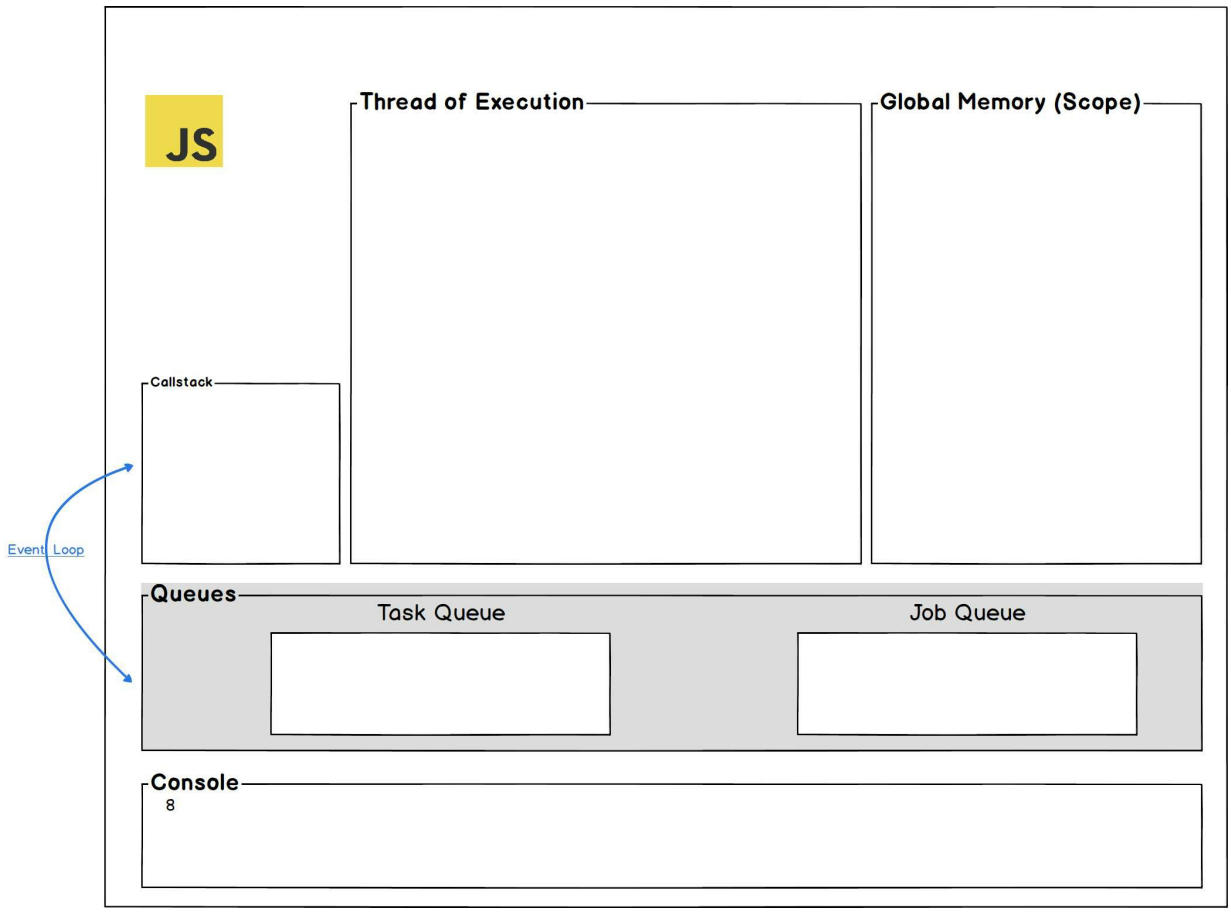

JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj:
onFullfullment: [ ]
onRejection: [ ]
value: _
then: fn(fn(val))
proto:
Callstack
Event Loop
Queues
Task Queue
Job Queue
[fn(val)]
Console


JS
Thread of Execution
body of fn
Local Memory
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj:
onFullfullment: [ ]
onRejection: [ ]
value: _
then: fn(fn(val))
proto:
Callstack
Event Loop
Queues
Task Queue
Job Queue
[fn(val)]
Console


JS
Thread of Execution
body of fn
console.log(val)
Local Memory
val: 8
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj:
onFullfullment: [ ]
onRejection: [ ]
value: _
then: fn(fn(val))
proto:
Callstack
Event Loop
Queues
Task Queue
Job Queue
[fn(val)]
Console


JS
Thread of Execution
body of fn
console.log(val)
Local Memory
val: 8
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj:
onFullfullment: [ ]
onRejection: [ ]
value: _
then: fn(fn(val))
proto:
Callstack
Event Loop
Queues
Task Queue
Job Queue
[fn(val)]
Console
8


JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj:
onFullfullment: [ ]
onRejection: [ ]
value: _
Callstack
Event Loop
Queues
Task Queue
Job Queue
Console
8



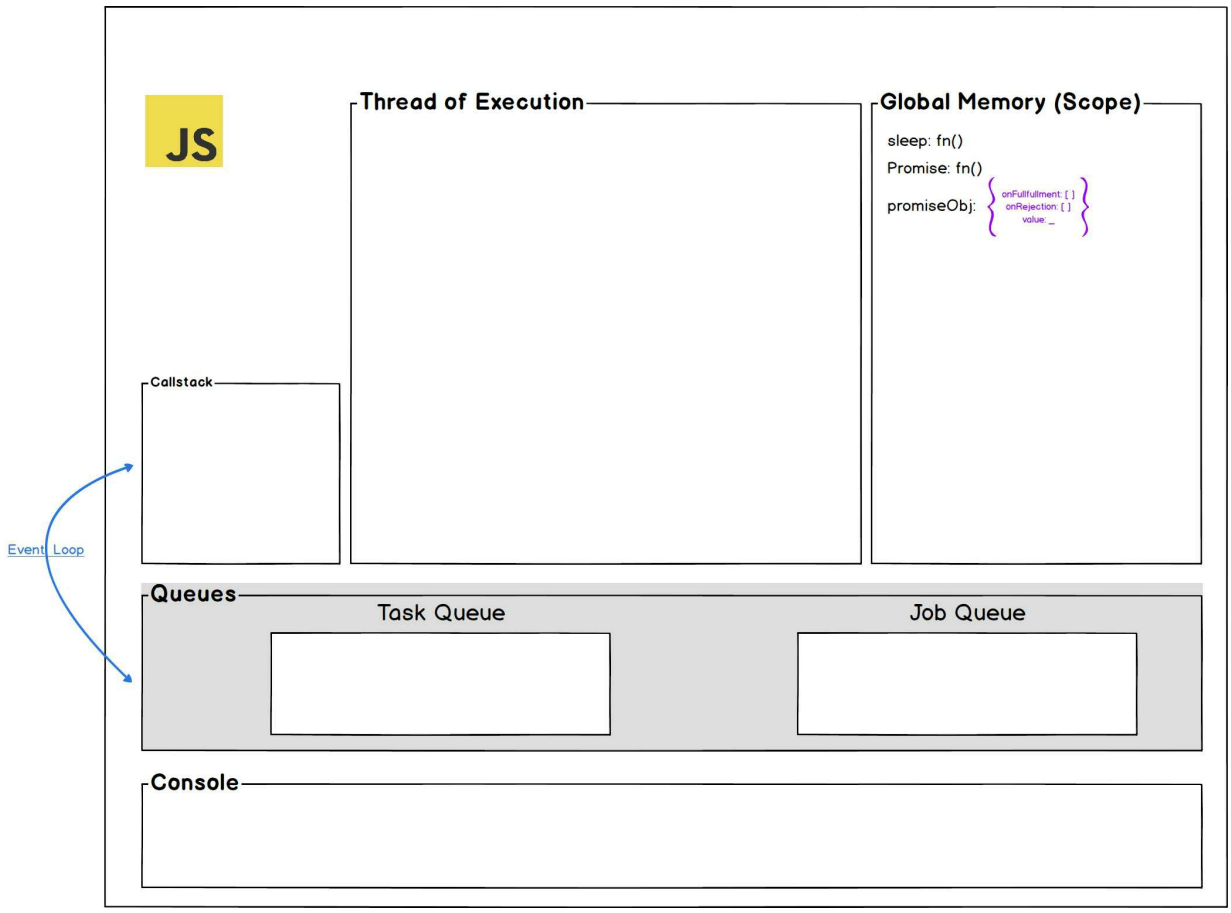
- That was too much for that little promiseObj😊, but it’s what happens under the hood.
- Let’s not forget our second scenario as well, it’s almost the same as above but it will use the onRejection array of promiseObj!
- let’s visualize that😊.

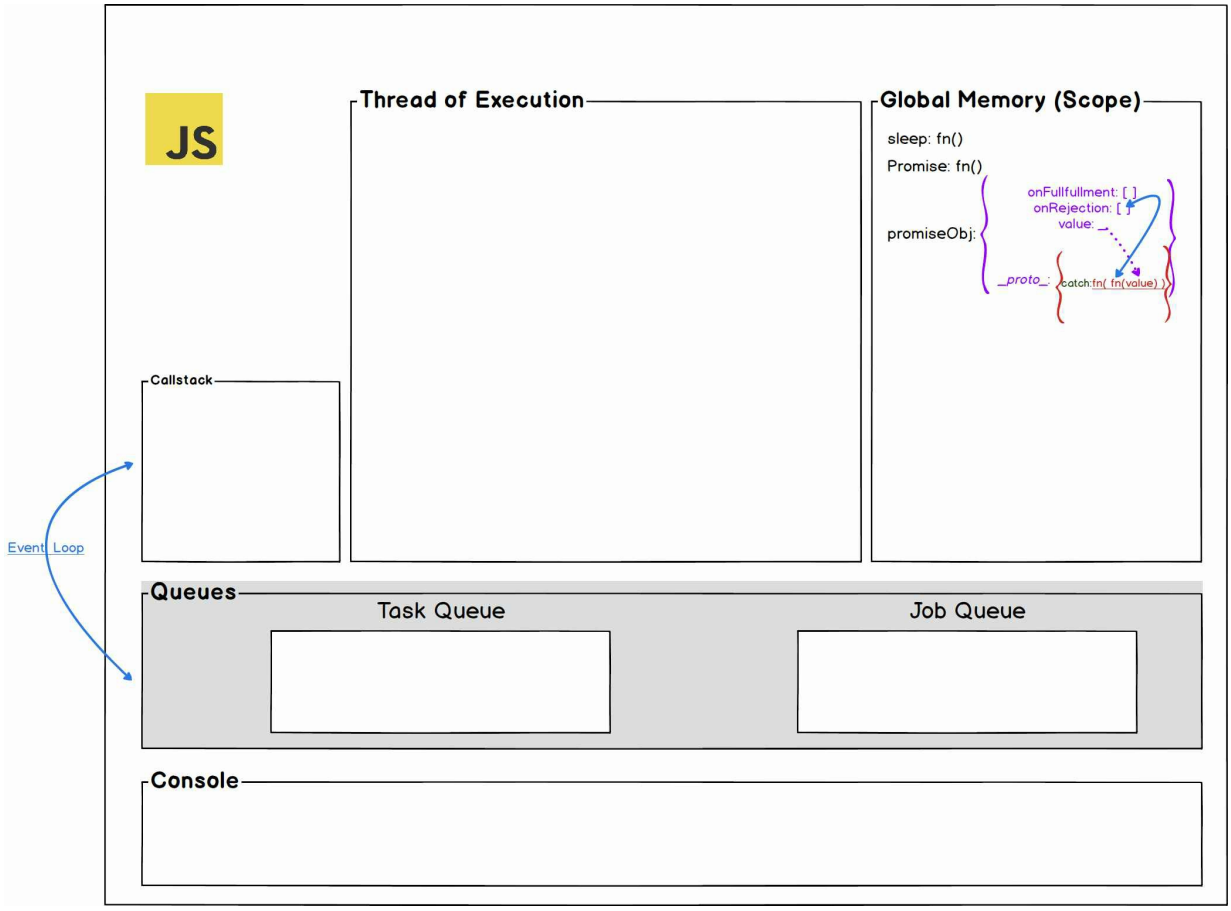
```
Reject function scenario
```

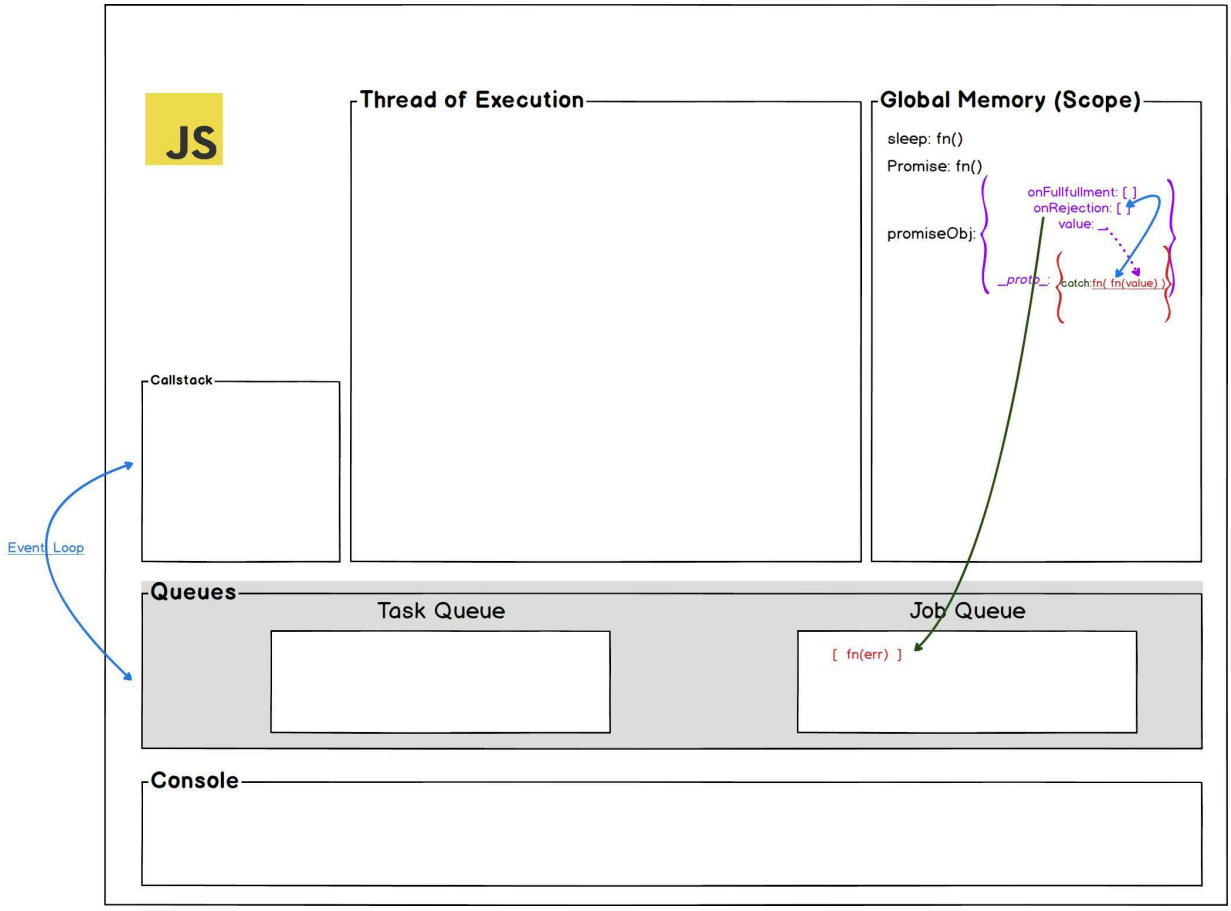
```
function sleep(sec) {
    let start = new Date().getTime();
    let expire = start + (sec*1000);
    while (new Date().getTime() < expire) {}
    return;
}
const promiseObj = new Promise((resolve, reject) => {
    let num = Math.floor(Math.random()*10) + 1;
    sleep(2);
    if(num > 5) {
        resolve(num);
    } else {
        reject(num);
    }
});

promiseObj.then((val) => {
    console.log(val)
}).catch((err) => {
    console.error(err)
})
```

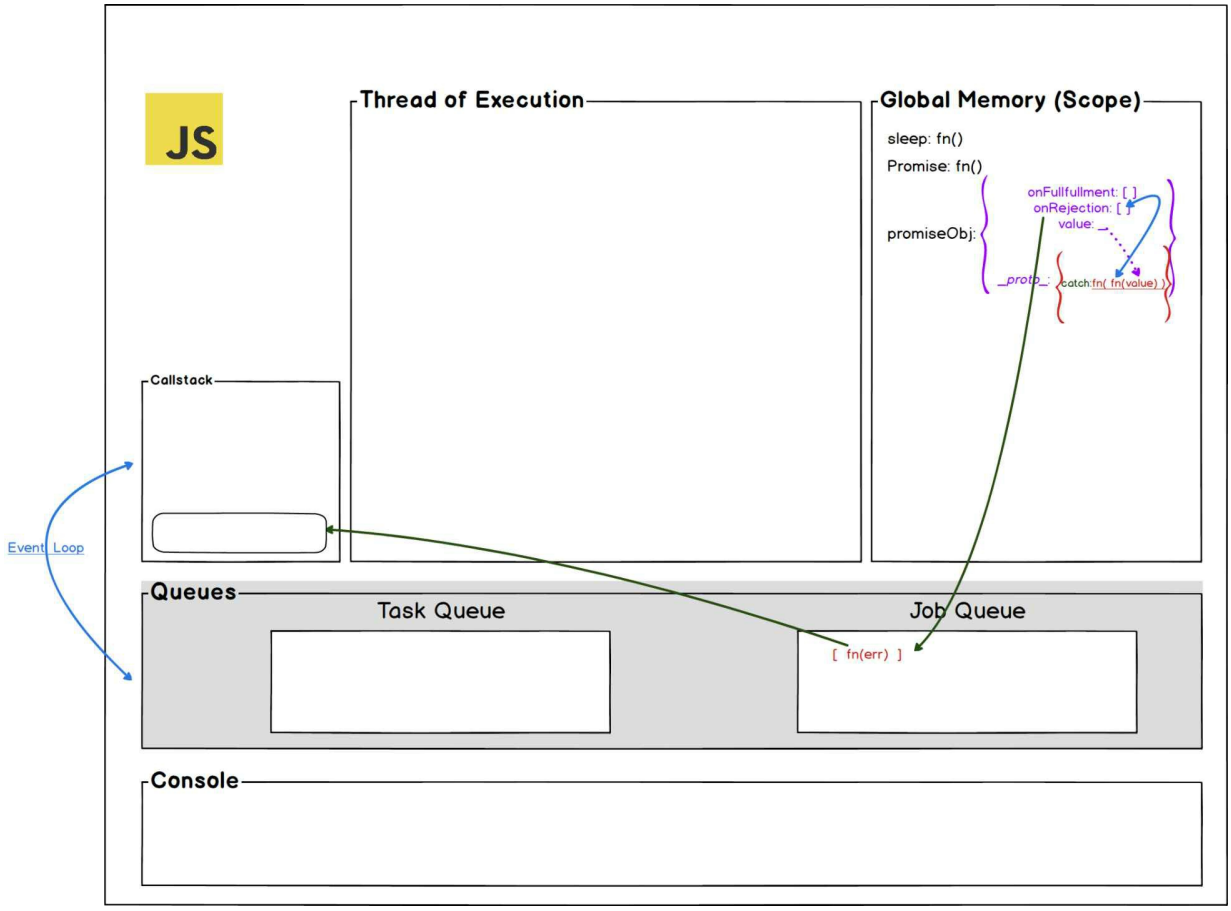
```
4
Promise {<fulfilled>: undefined}
  __proto__: Promise
    catch: ƒ catch()
    constructor: ƒ Promise()
    finally: ƒ finally()
    then: ƒ then()
    Symbol(Symbol.toStringTag): "Promise"
    __proto__: Object
  [[PromiseState]]: "fulfilled"
  [[PromiseResult]]: undefined
```

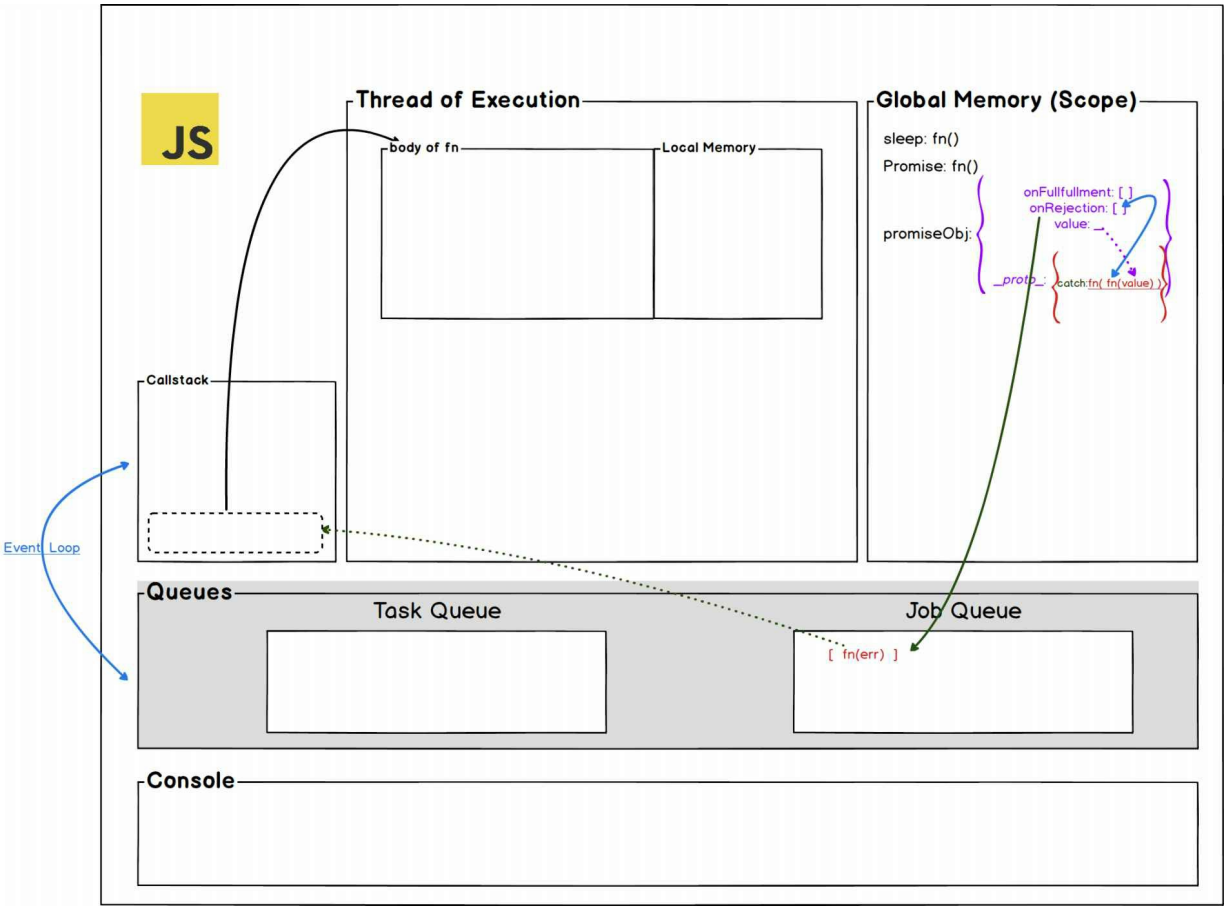
Let’s see step by step it’s execution.

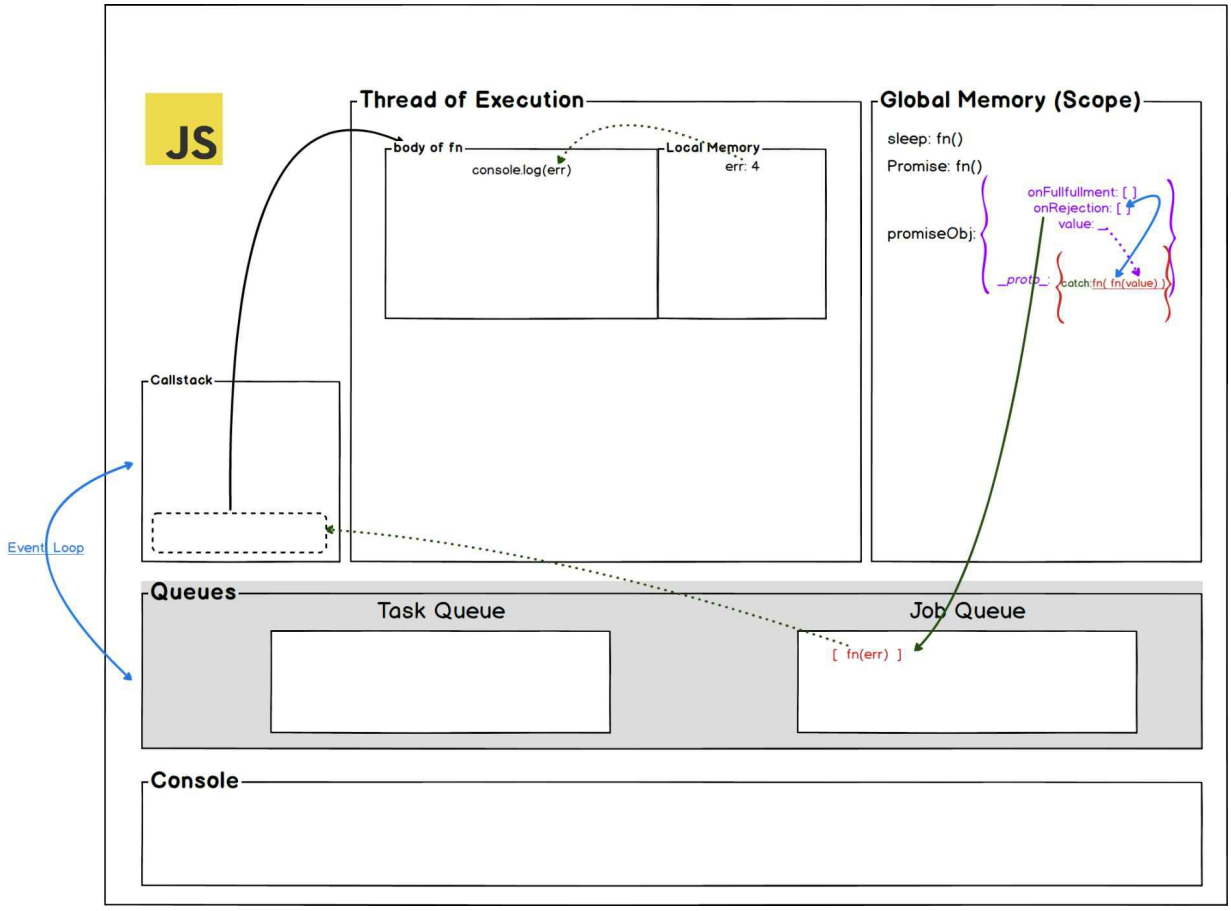

JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj:
onFullfullment: [ ]
onRejection: [ ]
value: _
Callstack
Event Loop
Queues
Task Queue
Job Queue
Console

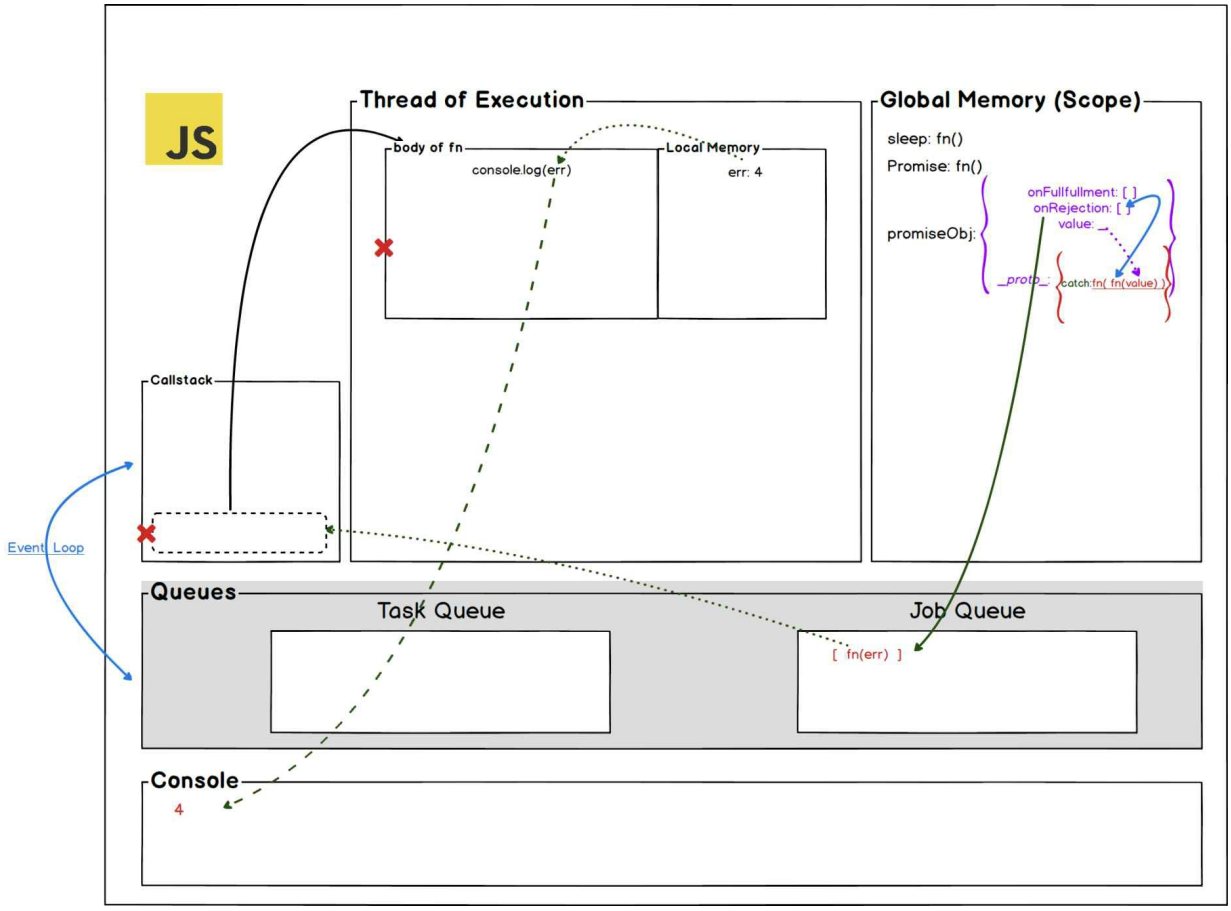

JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj:
onFullfullment: [ ]
onRejection: [ ]
value:
proto:
catch:fn(fn(value))
Callstack
Event Loop
Queues
Task Queue
Job Queue
Console

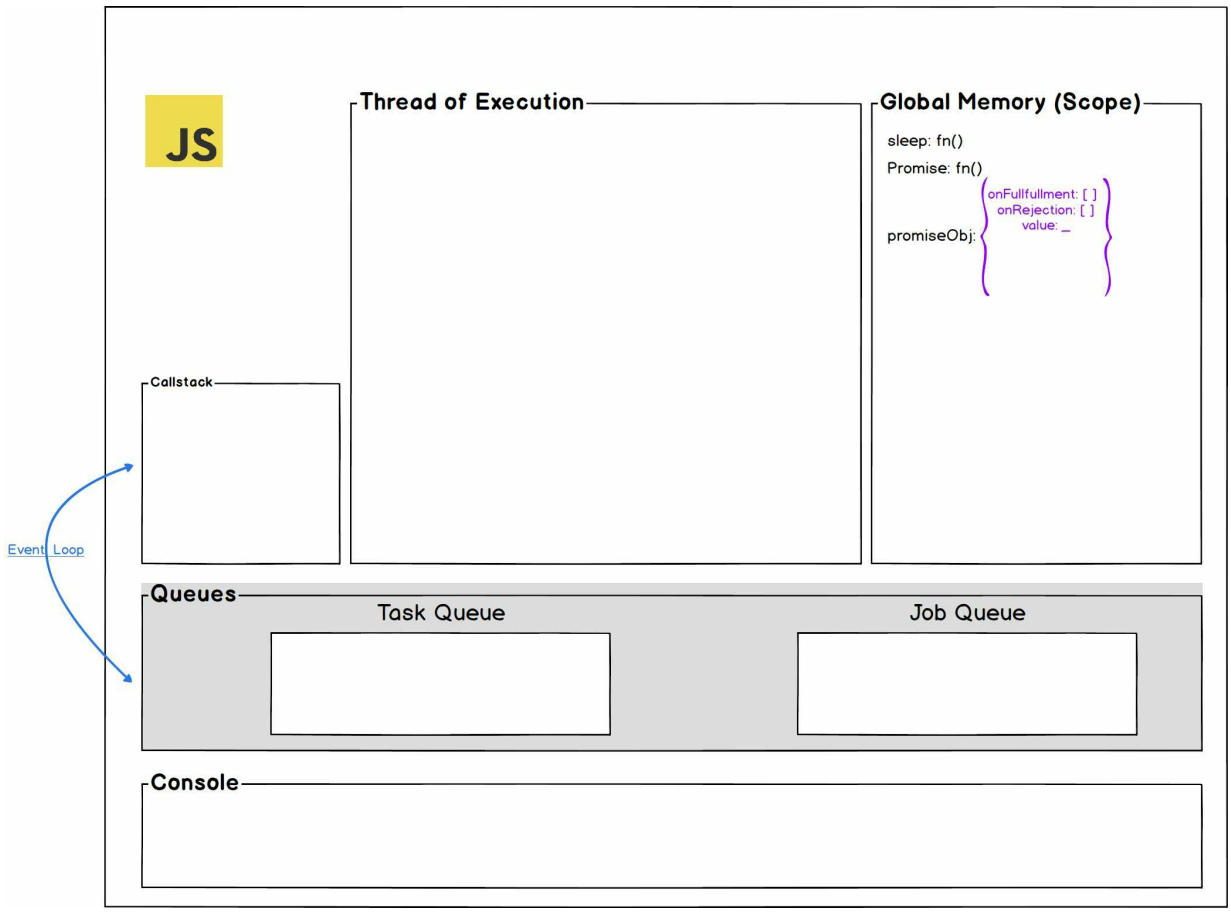

JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj:
onFullfullment: [ ]
onRejection: [ ]
value:
__proto__:
catch:fn(fn(value))
Callstack
Event Loop
Queues
Task Queue
Job Queue
[fn(err)]
Console

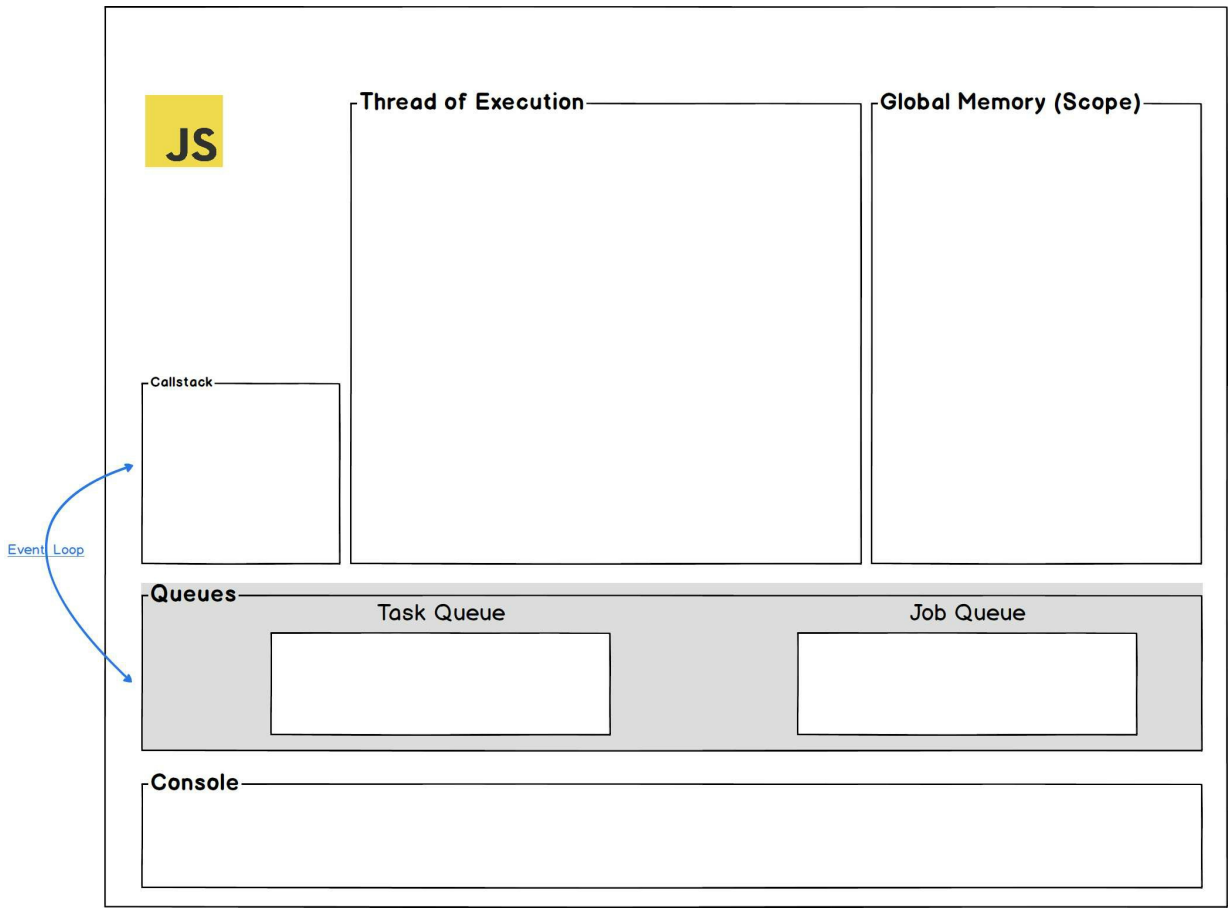

JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj:
onFullfullment: [ ]
onRejection: [ ]
value:
__proto__:
catch:fn(fn(value))
Callstack
Event Loop
Queues
Task Queue
Job Queue
[fn(err)]
Console


JS
Thread of Execution
body of fn
Local Memory
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj:
onFullfullment: [ ]
onRejection: [ ]
value:
__proto__:
catch:fn(fn(value))
Callstack
Event Loop
Queues
Task Queue
Job Queue
[fn(err)]
Console


JS
Thread of Execution
body of fn
console.log(err)
Local Memory
err: 4
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj:
onFullfullment: [ ]
onRejection: [ ]
value:
__proto__:
catch:fn(fn(value))
Callstack
Event Loop
Queues
Task Queue
Job Queue
[fn(err)]
Console


JS
Thread of Execution
body of fn
console.log(err)
Local Memory
err: 4
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj:
onFullfullment: [ ]
onRejection: [ ]
value:
__proto__:
catch:fn(fn(value))
Callstack
Event Loop
Queues
Task Queue
Job Queue
[fn(err)]
Console
4

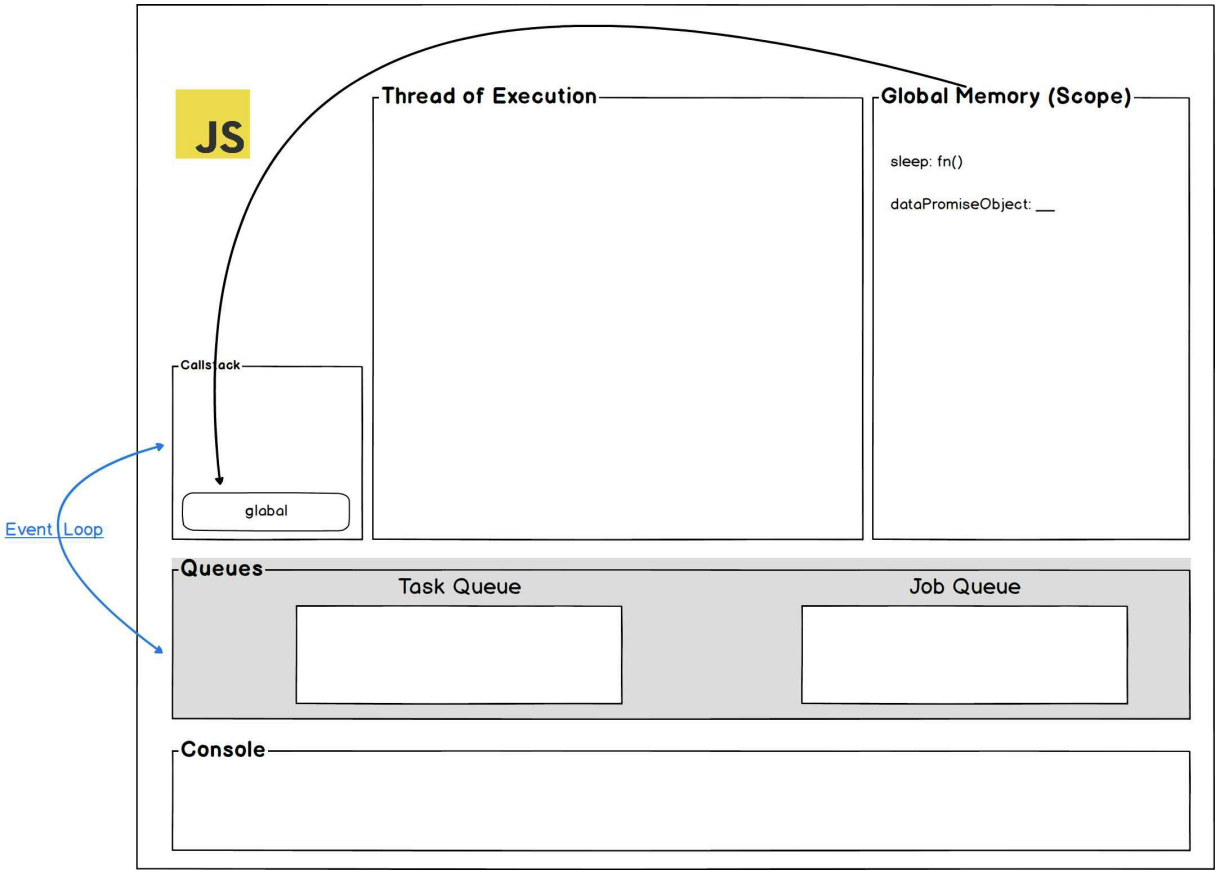

JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
Promise: fn()
promiseObj:
onFullfullment: [ ]
onRejection: [ ]
value: _
Callstack
Event Loop
Queues
Task Queue
Job Queue
Console

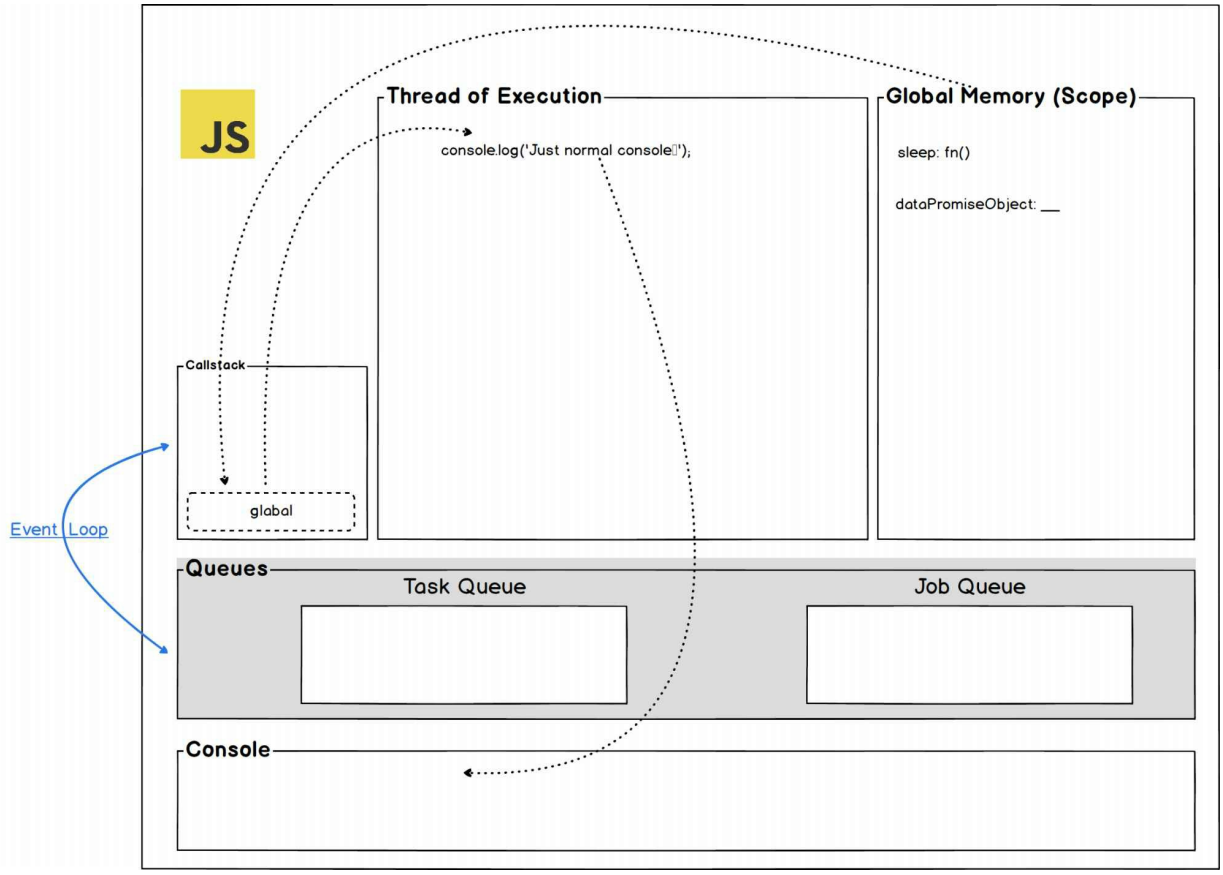


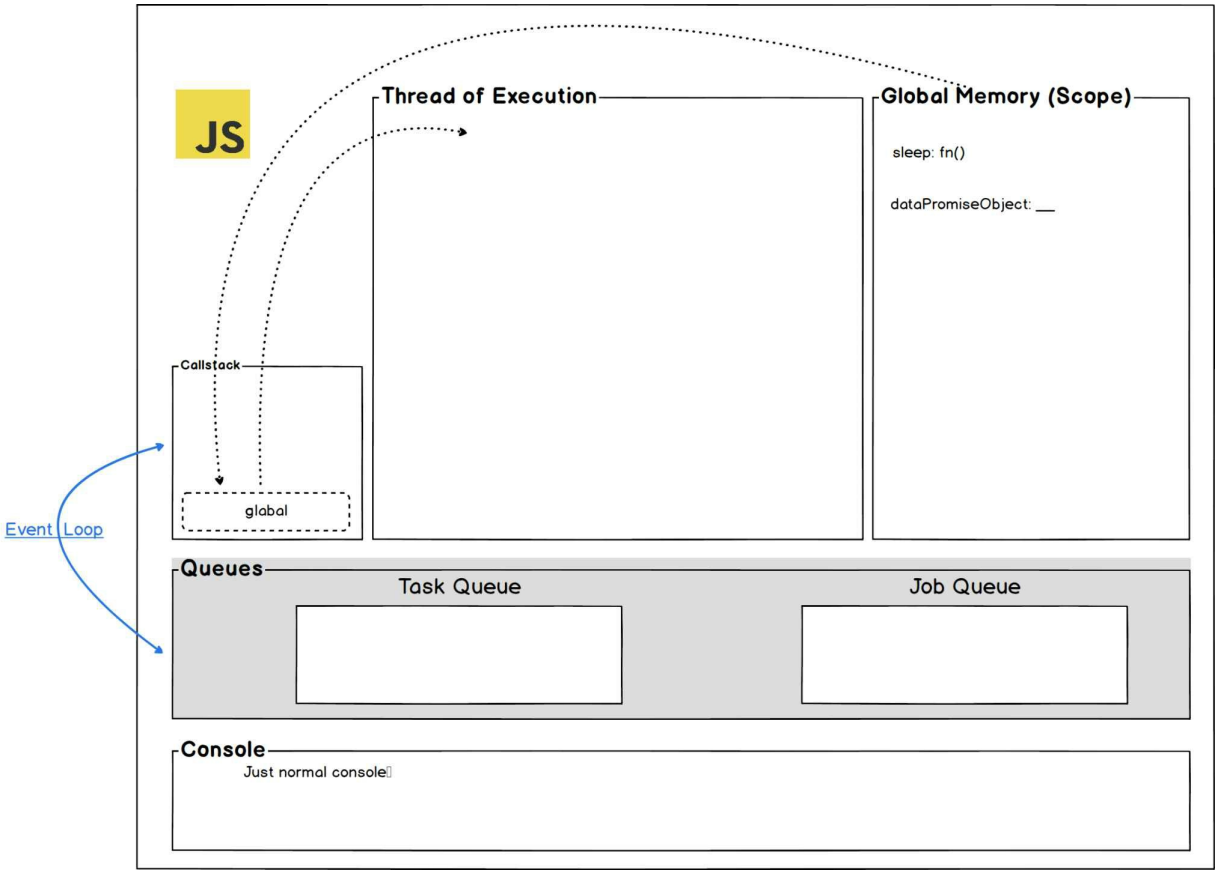
- And there you have it😊, a hell of a lot of things under the hood. This is how a promise works. Now, what about **fetch API available in the browser which is also promise-based! It works almost like the above example but with a little twist😀.** Why not explore visualization of it with a simple example😊.

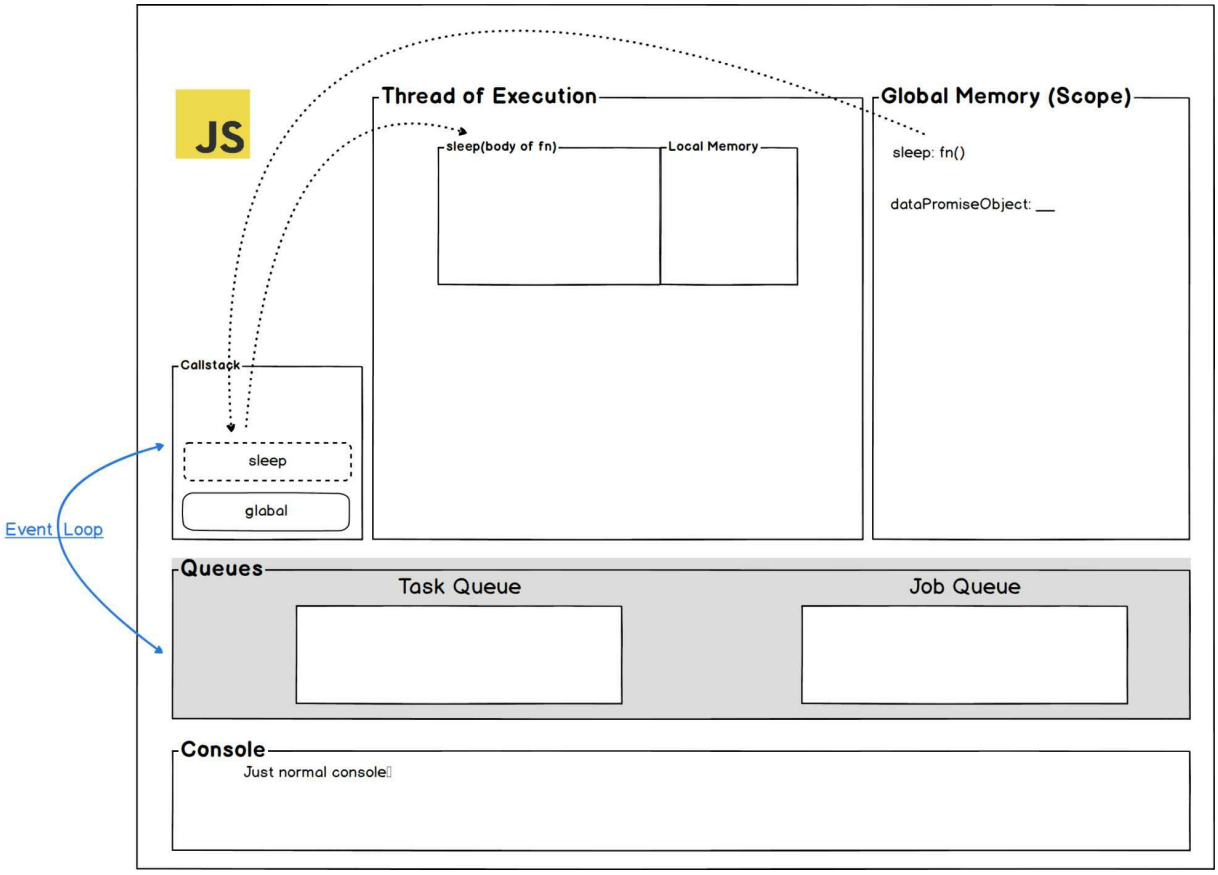
```
114
115 console.log('Just normal console😉');
116
117 function sleep(sec) {
118     let start = new Date().getTime();
119     let expire = start + (sec*1000);
120     while (new Date().getTime() < expire) {}
121     return;
122 }
123
124 sleep(2);
125
126 // ------------------------------------------------------------------------------
127 // assume here we are getting back 'hello' string back from https://test.com
128 // ------------------------------------------------------------------------------
129 const dataPromiseObject = fetch('https://test.com');
130
131 dataPromiseObject.then((data) => {
132     console.log(data) // 'hello'
133 });
134
```

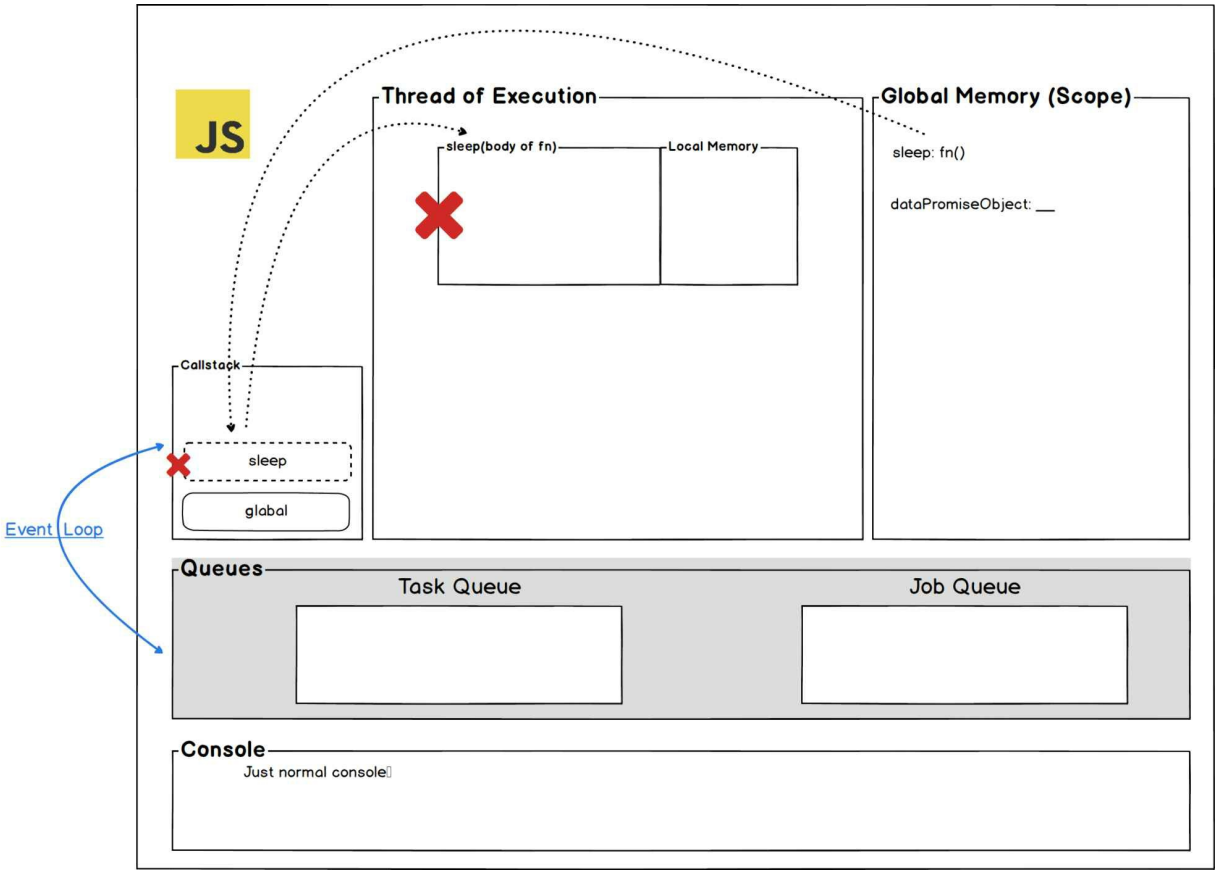
- Here, we have a simple console log, little blocking for 2 seconds, and fetch API code. let’s start visualizing it.

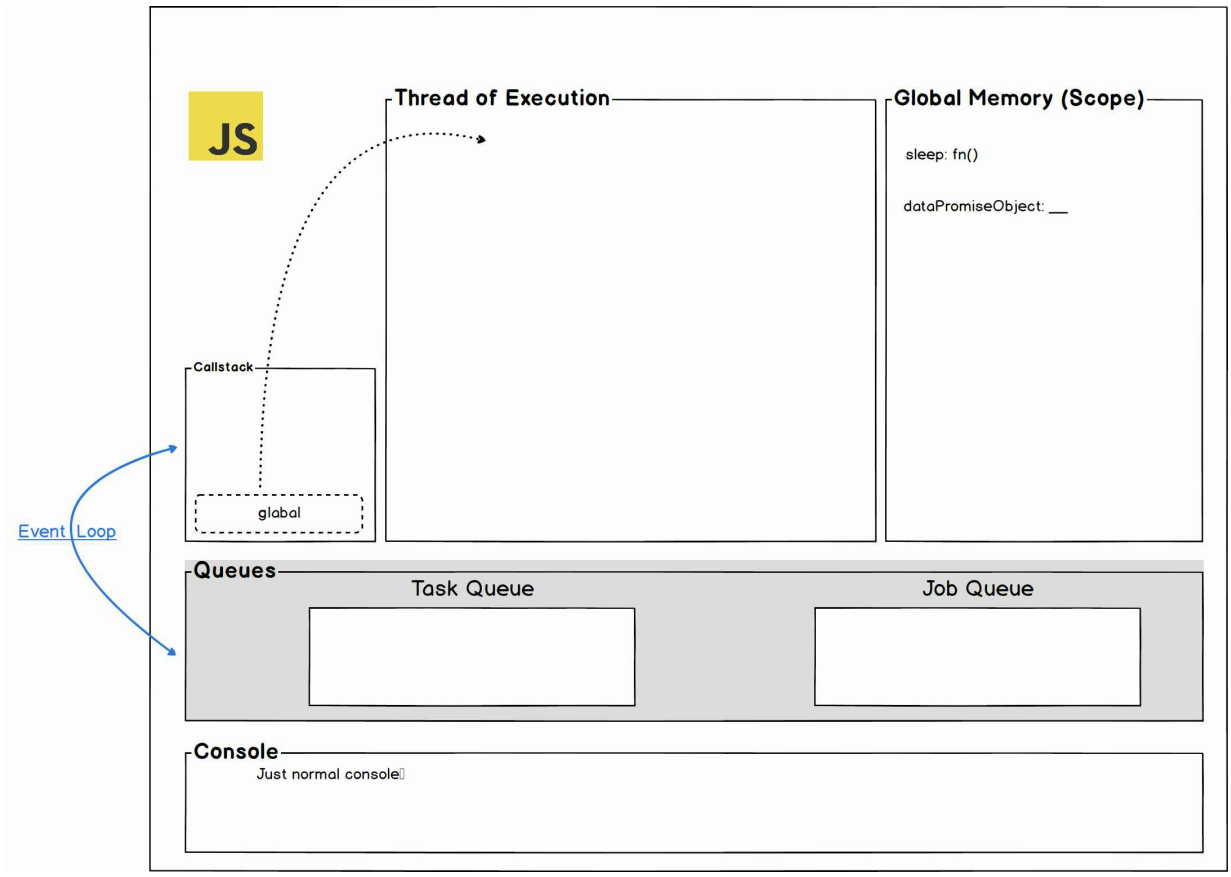

JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
dataPromiseObject: __
Callstack
global
Event Loop
Queues
Task Queue
Job Queue
Console


JS
Thread of Execution
console.log('Just normal console');
Global Memory (Scope)
sleep: fn()
dataPromiseObject: __
Callstack
global
Event Loop
Queues
Task Queue
Job Queue
Console

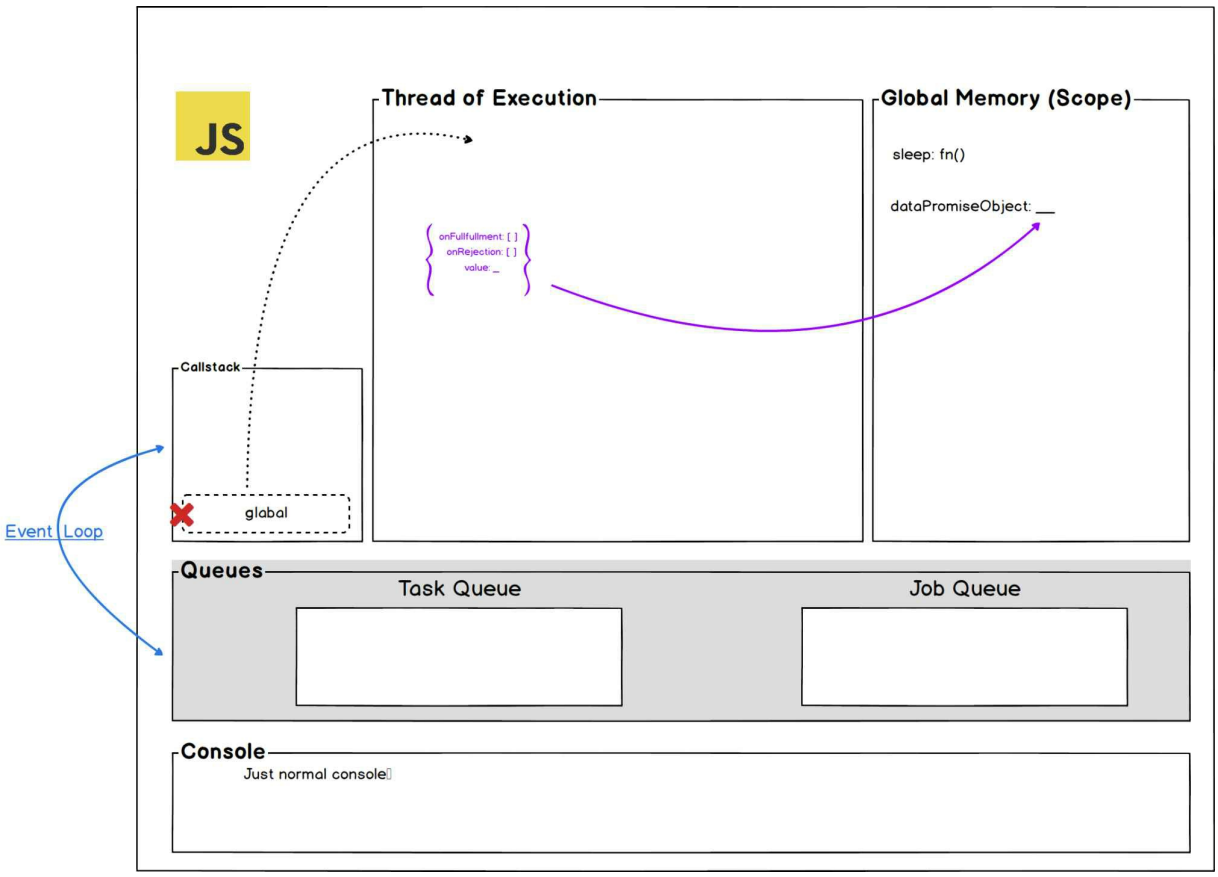

JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
dataPromiseObject: __
Callstack
global
Event Loop
Queues
Task Queue
Job Queue
Console
Just normal console

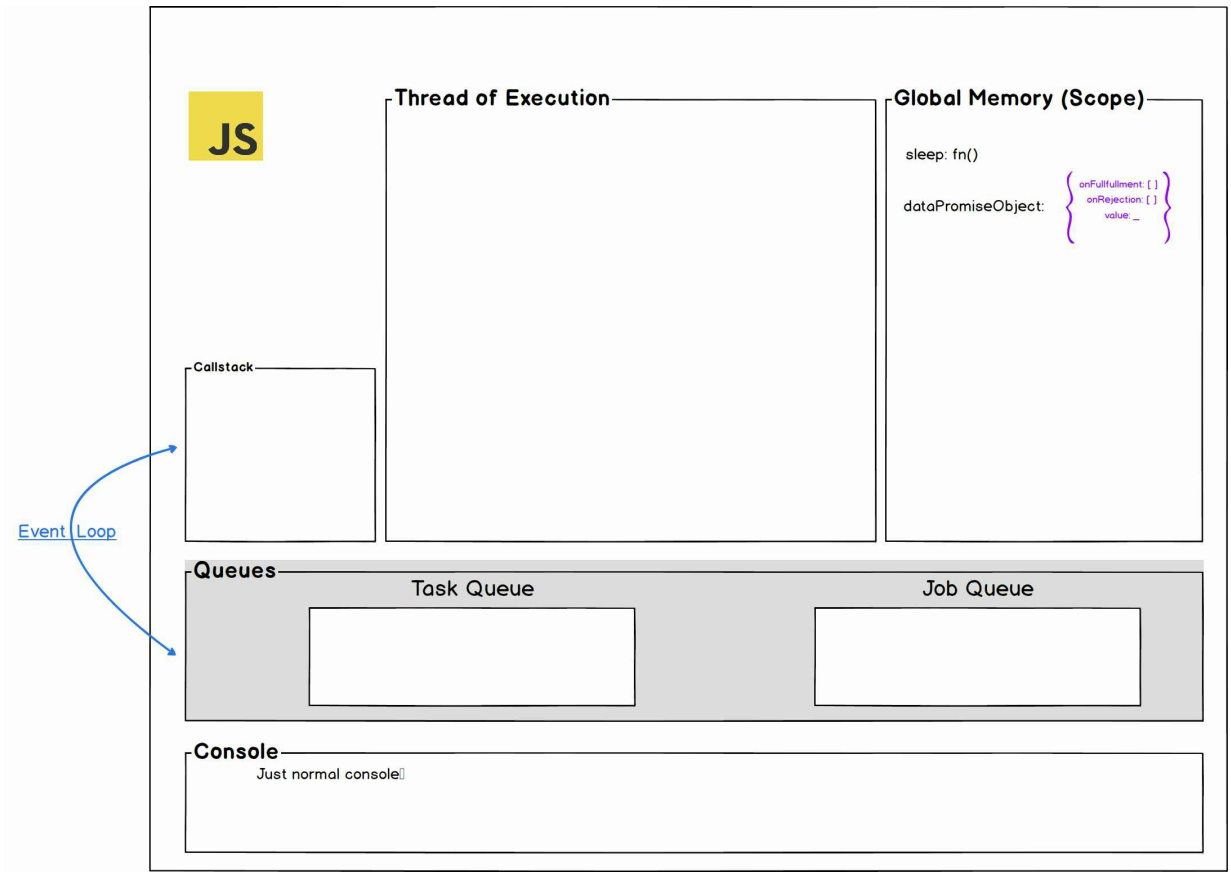

JS
Thread of Execution
sleep(body of fn)
Local Memory
Global Memory (Scope)
sleep: fn()
dataPromiseObject: __
Callstack
sleep
global
Event Loop
Queues
Task Queue
Job Queue
Console
Just normal console


JS
Thread of Execution
sleep(body of fn)
Local Memory
Global Memory (Scope)
sleep: fn()
dataPromiseObject: __
Callstack
sleep
global
Event Loop
Queues
Task Queue
Job Queue
Console
Just normal console

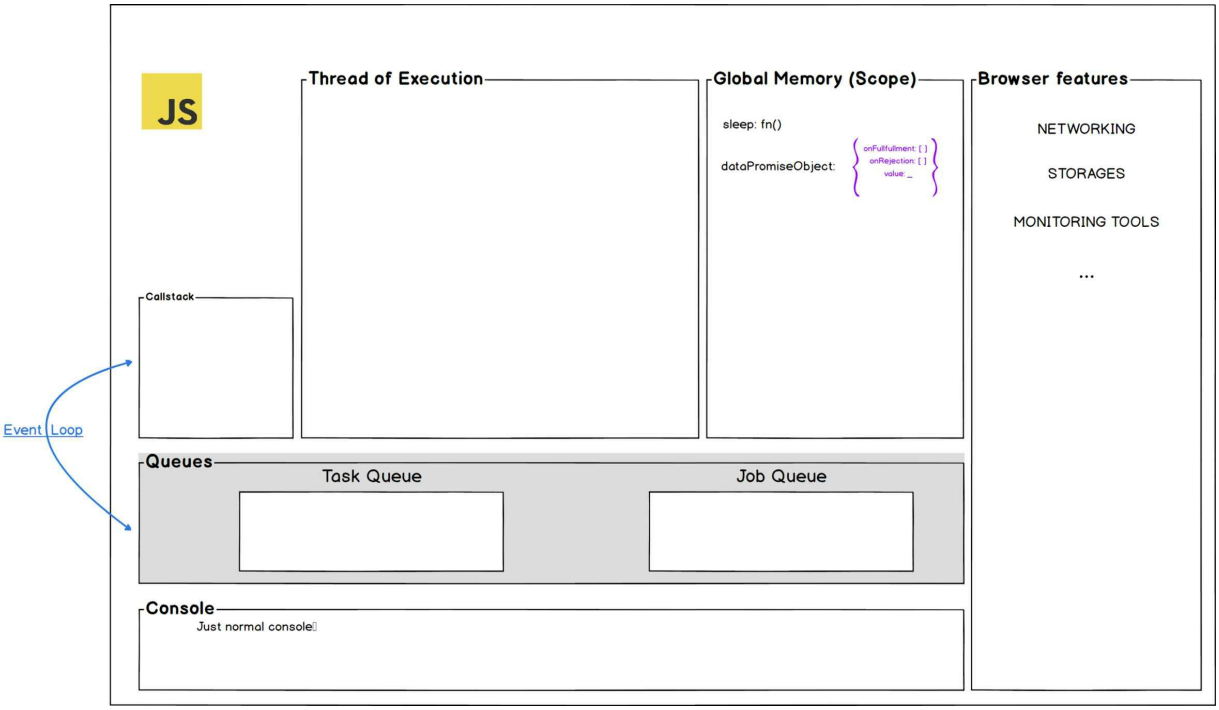


- we don’t have any other synchronous code to run so we will exit out of the global execution context and callstack will become empty.


JS
Thread of Execution
Global Memory (Scope)
sleep: fn()
dataPromiseObject: __
onFullfullment: [ ]
onRejection: [ ]
value: _
Callstack
global
Event Loop
Queues
Task Queue
Job Queue
Console
Just normal console



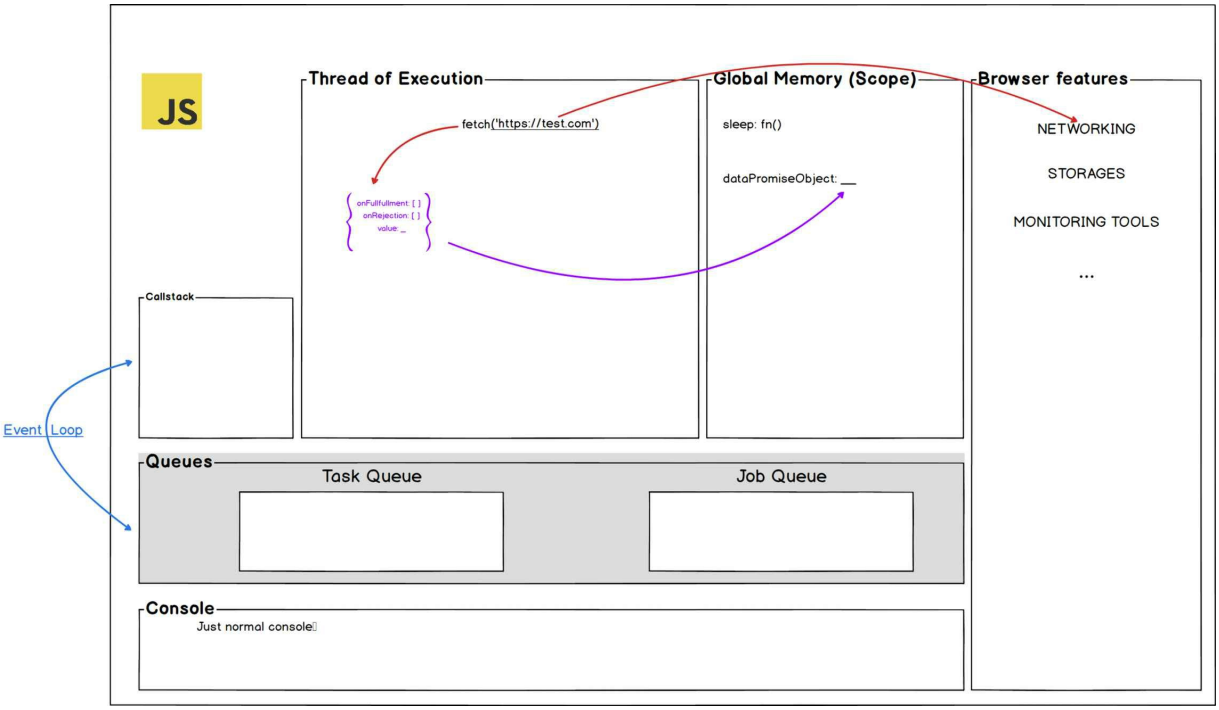
- I think till now everything seems the same as what we visualized earlier, but here the twist comes. T**he above shown mental model is still incomplete😃.**
- We have a lot of browser features (AKA browser APIs) which happen outside of the regular execution context. not only that, these browser features actually happens through our underline Operating System and using Kernel. But to keep it short and simple let’s just think these browser features are happening outside of the actual JavaScript execution context.
- One such feature is the fetch API.
- Let’s extend our mental model a little bit more😃.



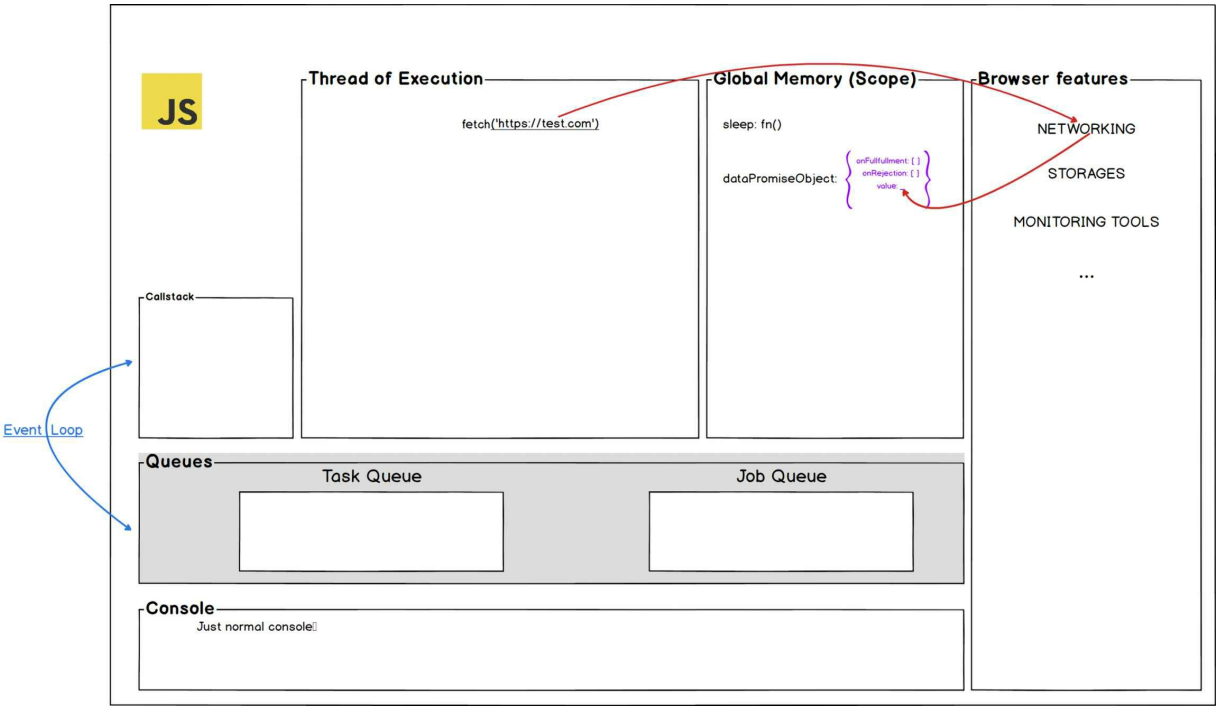
- As you can see we added browser features inside our diagram, now you can sense that our browser environment is not just JavaScript but a lot of hidden features combined under the hood.

```
// ---------------------------------------------------------------------------
// assume here we are getting back 'hello' string back from https://test.com
// ---------------------------------------------------------------------------
const dataPromiseObject = fetch('https://test.com');
```

- Here, let's see what happens when hit **fetch('https://test.com')** below the line when we execute code.
- we are going back again to what happens when we actually return dataPromiseObject.



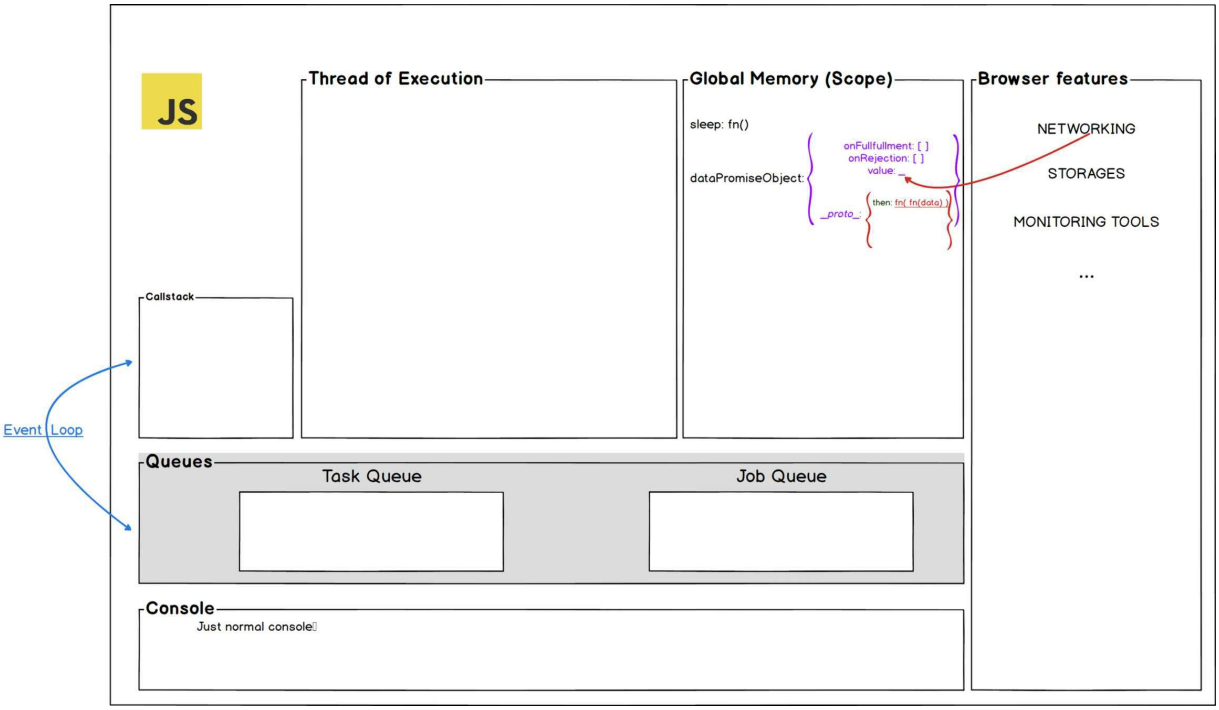
- We can see that, fetch actually make bond two way one in javascript task and other is networking work. let me continue with visualization ☺.

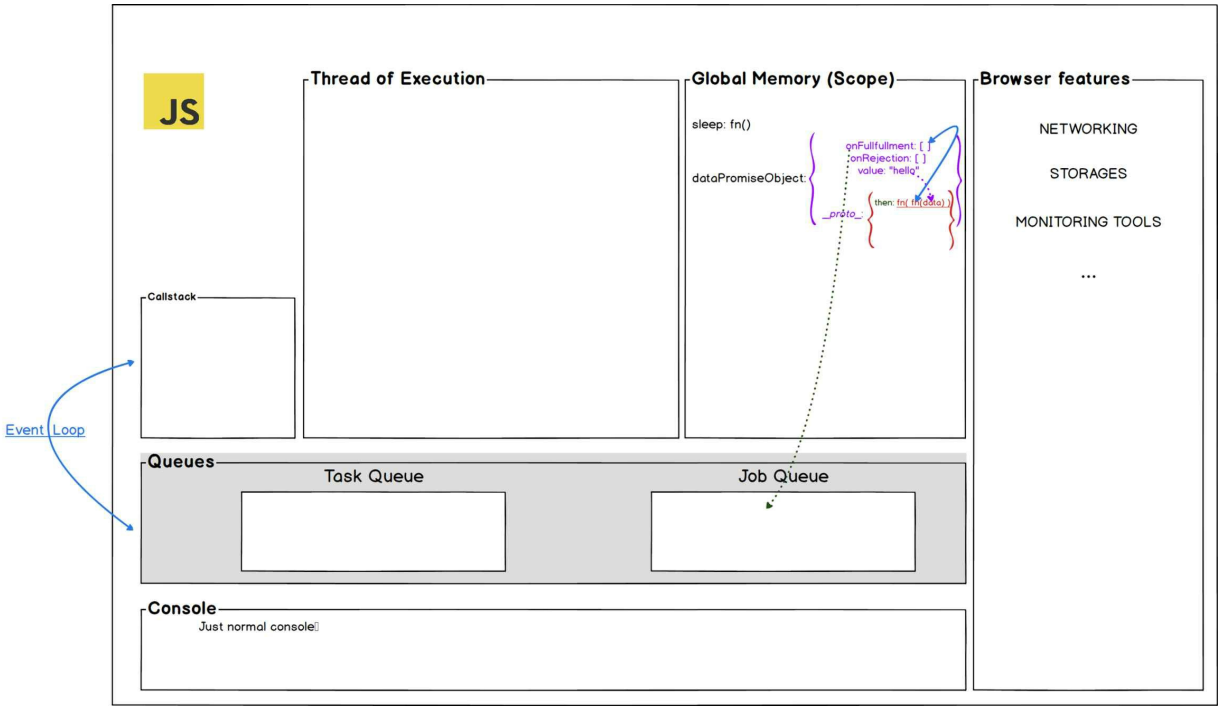


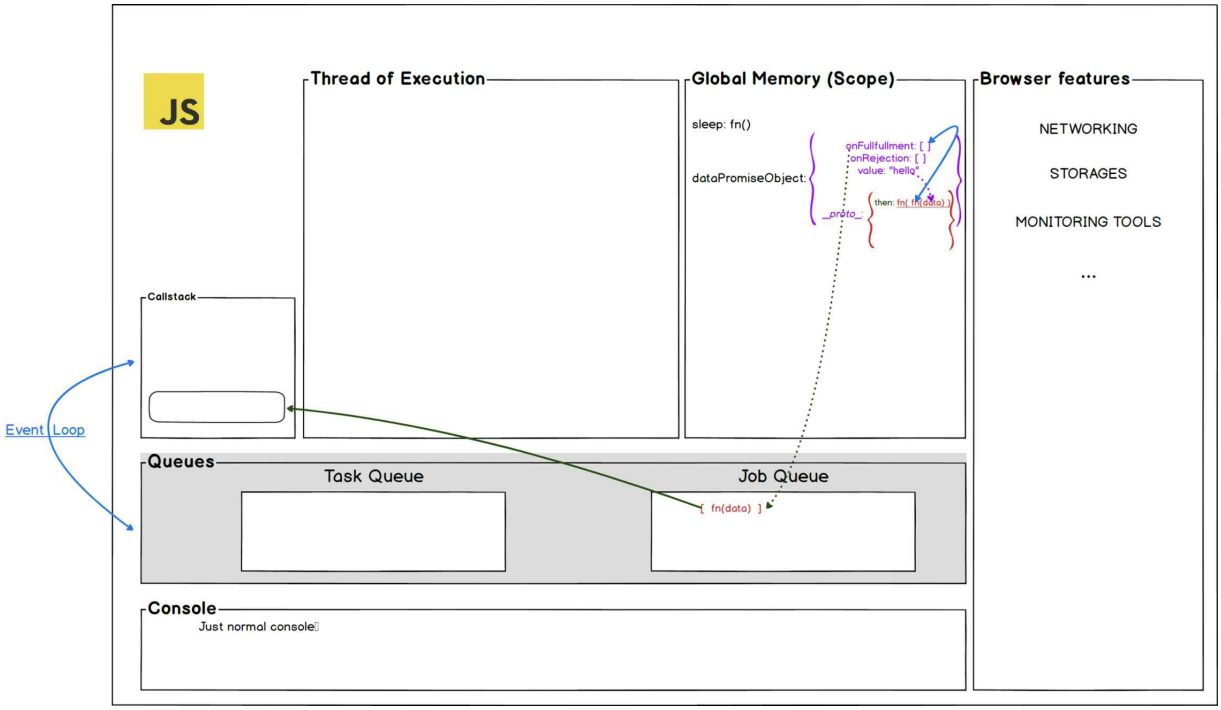
- After hitting the fetch line in our code, work for returning promise

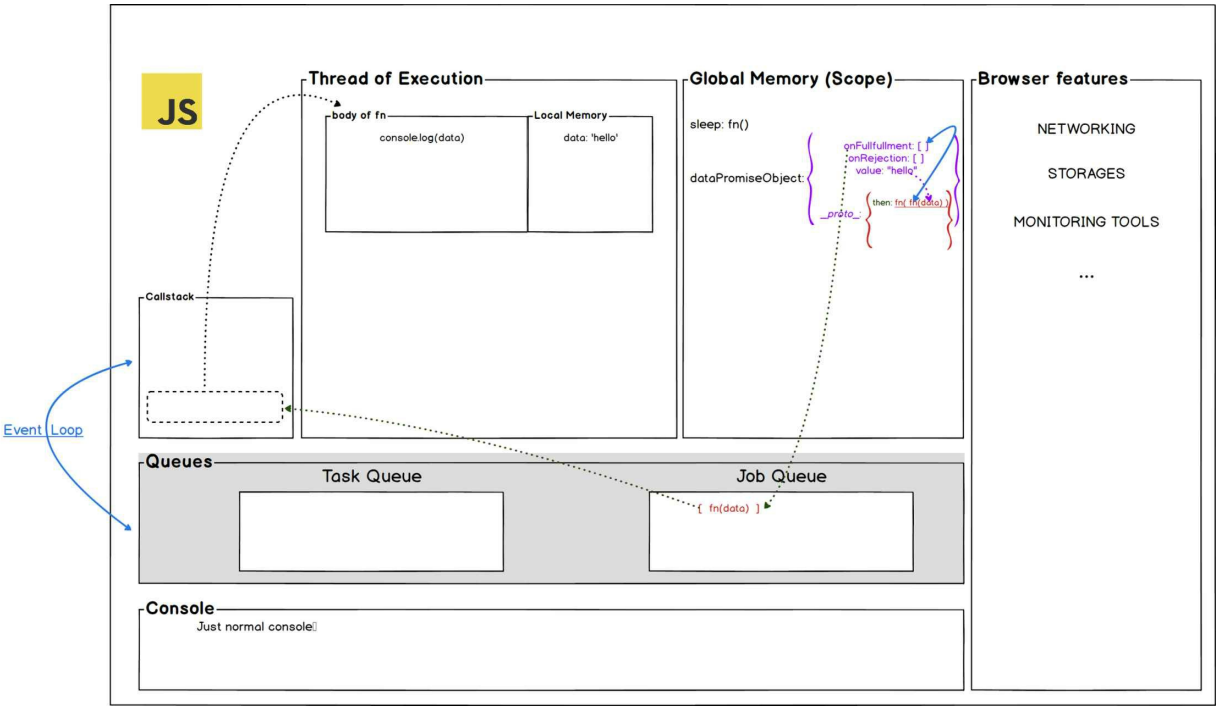
object from fetch API gets done immediately but background task for finishing API call still pending and it’s done by browser and given back to promise object when it gets completed.

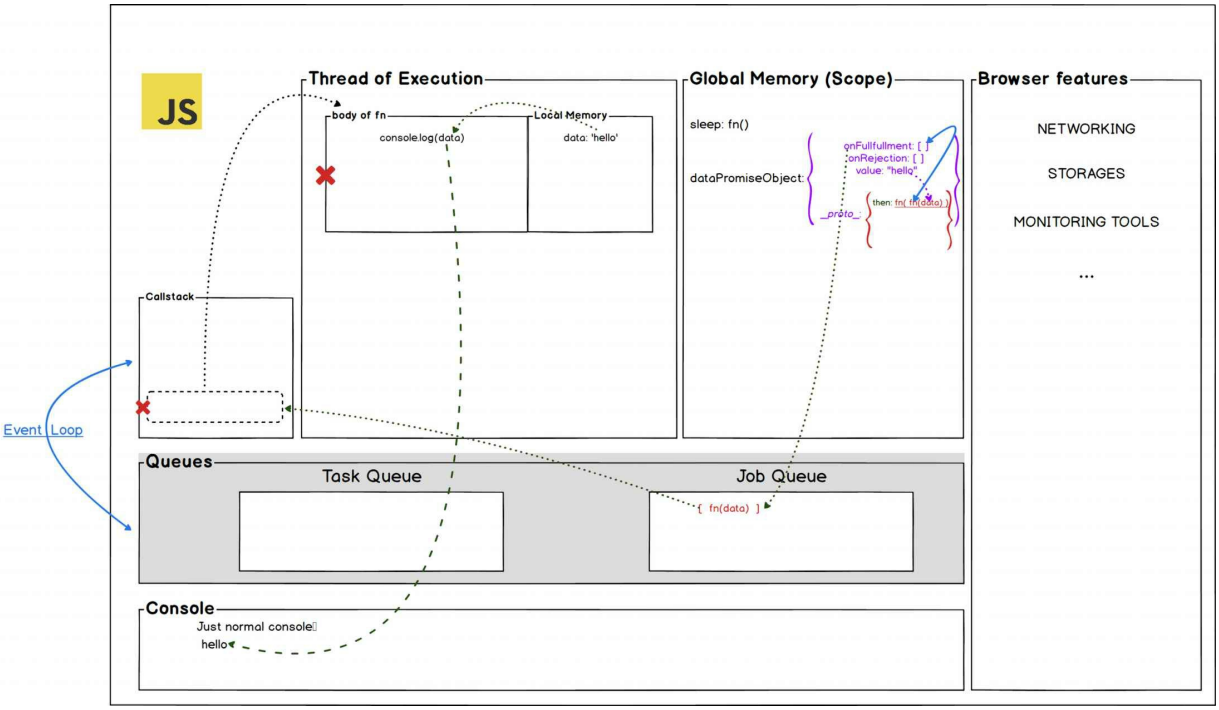
- After finishing background work, all other work is the same as before we visualized 😀.

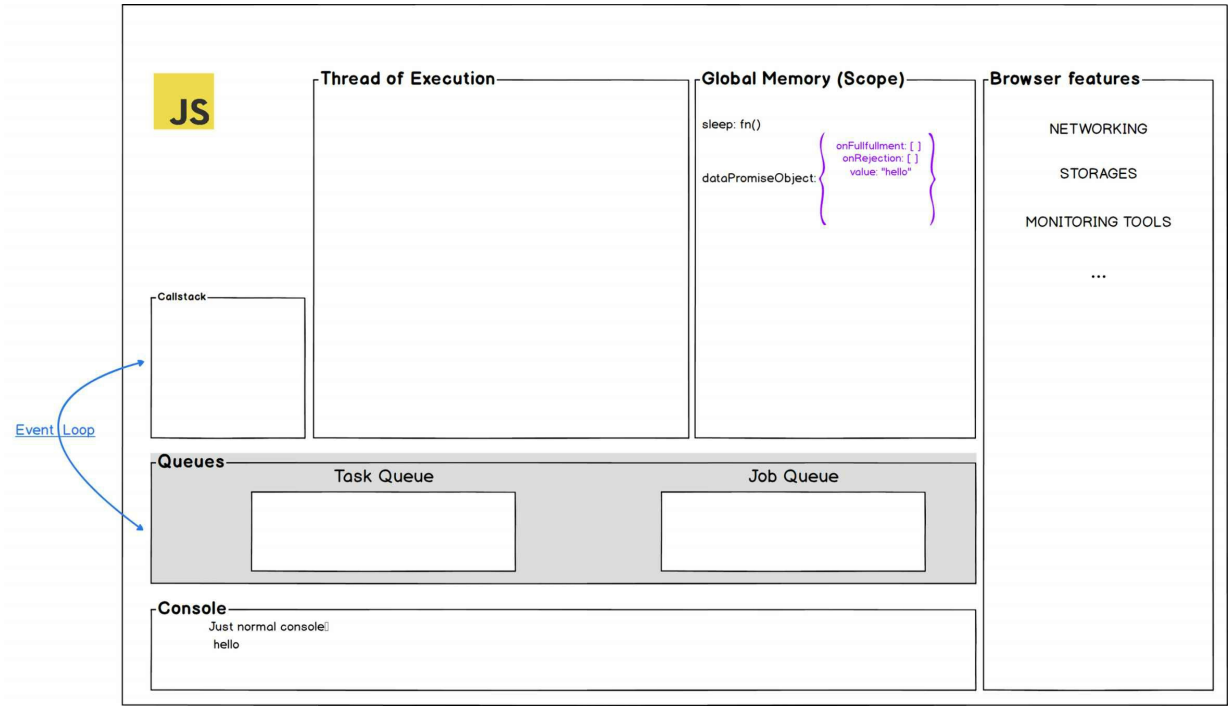


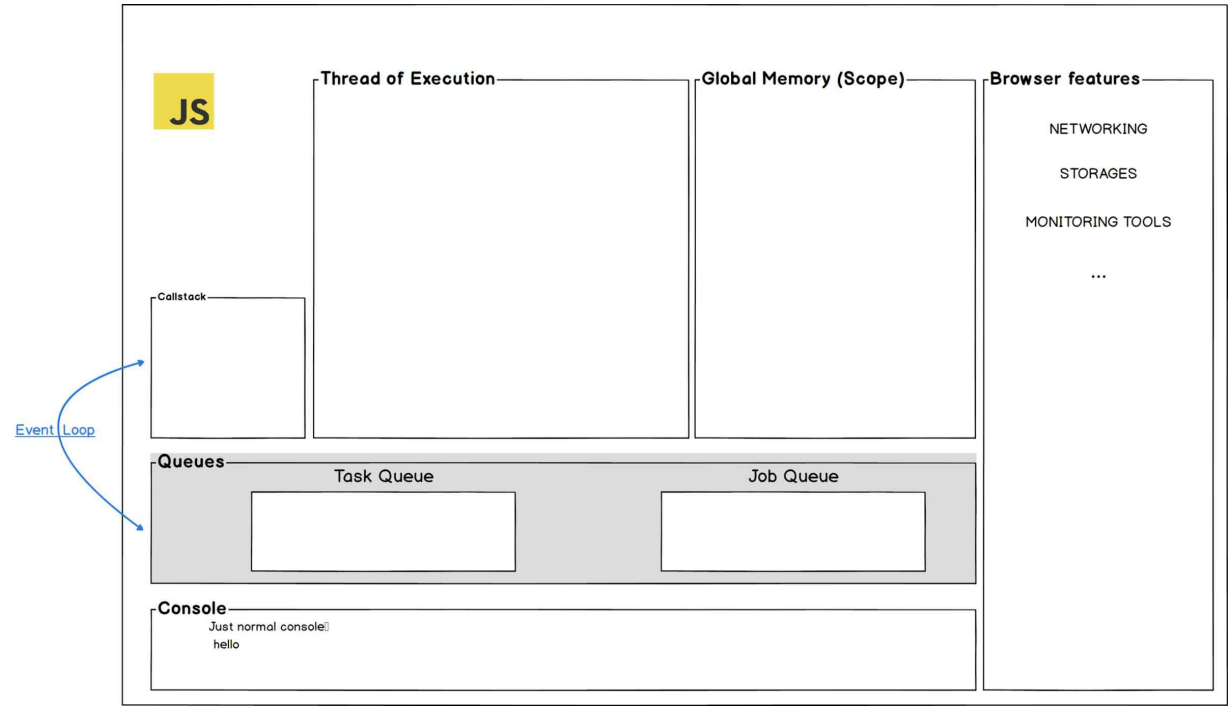

JS
Thread of Execution
Global Memory (Scope)
Browser features
sleep: fn()
dataPromiseObject:
onFullfullment: [ ]
onRejection: [ ]
value: "hello"
__proto__:
then: fn(fn(data))
NETWORKING
STORAGES
MONITORING TOOLS
...
Callstack
Event Loop
Queues
Task Queue
Job Queue
Console
Just normal console


JS
Thread of Execution
Global Memory (Scope)
Browser features
sleep: fn()
dataPromiseObject:
onFullfullment: [ ]
onRejection: [ ]
value: "hello"
__proto__:
then: fn(fn(data))
NETWORKING
STORAGES
MONITORING TOOLS
...
Callstack
Event Loop
Queues
Task Queue
Job Queue
[fn(data)]
Console
Just normal console


JS
Thread of Execution
body of fn
console.log(data)
Local Memory
data: 'hello'
Global Memory (Scope)
sleep: fn()
dataPromiseObject:
onFullfullment: [ ]
onRejection: [ ]
value: "hello"
__proto__:
then: fn(fn(data))
Browser features
NETWORKING
STORAGES
MONITORING TOOLS
...
Callstack
Event Loop
Queues
Task Queue
Job Queue
{ fn(data) }
Console
Just normal console


JS
Thread of Execution
body of fn
console.log(data)
Local Memory
data: 'hello'
Global Memory (Scope)
sleep: fn()
dataPromiseObject:
onFullfullment: [ ]
onRejection: [ ]
value: "hello"
__proto__:
then: fn(fn(data))
Browser features
NETWORKING
STORAGES
MONITORING TOOLS
...
Callstack
Event Loop
Queues
Task Queue
Job Queue
{ fn(data) }
Console
Just normal console
hello





This is all my friends, how really promise and fetch API work under the hood☺.

* * *

5

Iterators, Generators, and Async-Await

This chapter will give an in-depth explanation of and importance of iterator and generators. Also, how this stuff is utilized in real-world scenarios to make code look nicer and easy to read. not only that, this chapter will have a really high dense visualization.

Loops

- Let’s talk about traditional loops before we move to Iterators.
- We use loops to iterate over each element of **iterable** and do the operation. Let’s take a simple example of it.

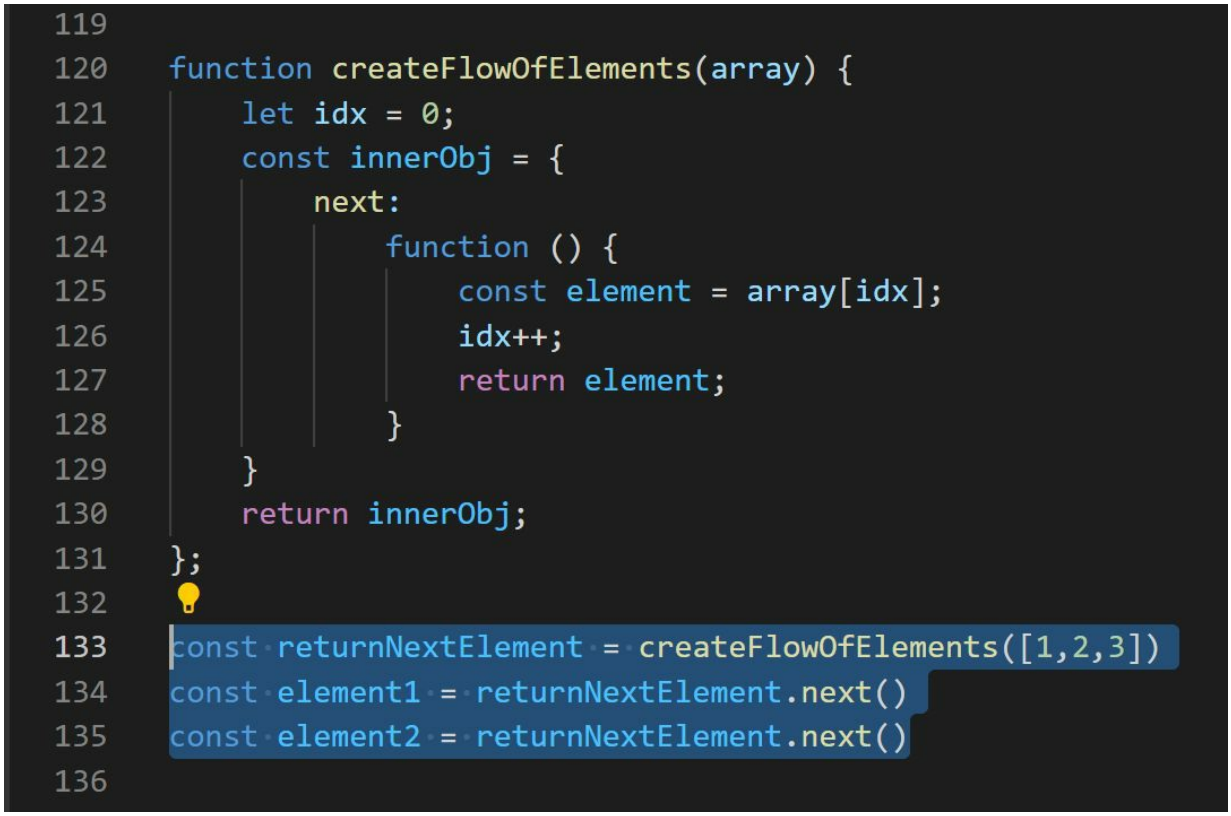
```
112
113    let numberArray = [1,2,3];
114
115    numberArray.forEach((element) => {
116        console.log(element*2)
117    })
118
```

- In the above example, we have a simple **numberArray** and we are iterating over elements and simply multiplying by 2.
- Notice here, we are doing the operation in **a single shot**. We don’t have control over in a single shot operation(you can put if-else and other stuff to control the flow but that is no real control over elements !).
- How about if we can get element one at a time and perform some operation on it. so whenever we need the next element we simply produce it. At first impression, it might seem a little weird to think but let me give an example by building a custom iterable.
- To make things simple, I will use the smaller diagram of our mental model and we will make an entire mental model diagram when we combine all the concepts😊. I will only show the thread of execution and memory for the below example.

Iterators

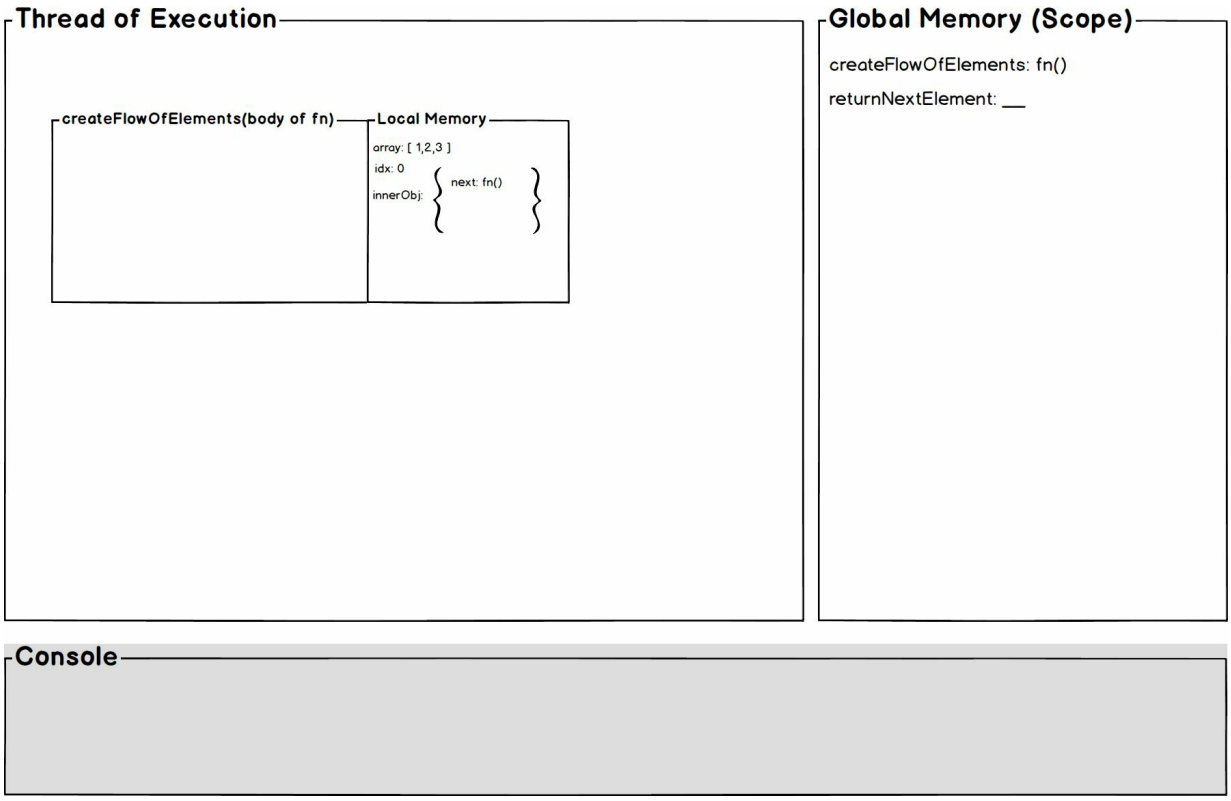
```
119
120  function createFlowOfElements(array) {
121      let idx = 0;
122      const innerObj = {
123          next:
124              function () {
125                  const element = array[idx];
126                  idx++;
127                  return element;
128              }
129      }
130      return innerObj;
131  };
132
```

We have declared one function called **‘createFlowOfElements’** which is actually a function that takes an array as an argument and returns an **innerObj**. Let’s execute this function.



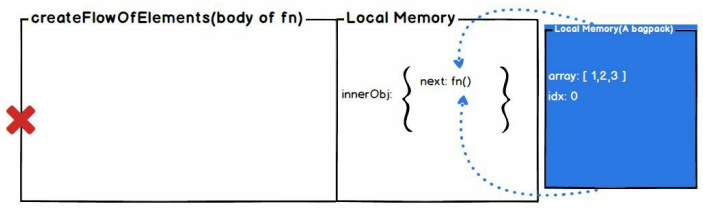
```
119
120  function createFlowOfElements(array) {
121      let idx = 0;
122      const innerObj = {
123          next:
124              function () {
125                  const element = array[idx];
126                  idx++;
127                  return element;
128              }
129      }
130      return innerObj;
131  };
132
133  const returnNextElement = createFlowOfElements([1,2,3])
134  const element1 = returnNextElement.next()
135  const element2 = returnNextElement.next()
136
```

Now it’s time to visualize what happens when we execute the **createFlowOfElements.**



- **In case you haven't guessed,** here we have a function inside a function(even if it's defined inside innerObj). So, if we return innerObj or function of innerObj we will have a backpack of outer scope (closure my friends😀).
- **createFlowOfElements** is a function that will actually be going to create iterator!. Let's visualize it.

Thread of Execution



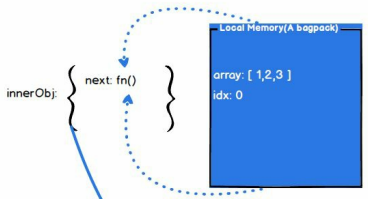
Global Memory (Scope)

createFlowOfElements: fn()

returnNextElement: __

Console

Thread of Execution

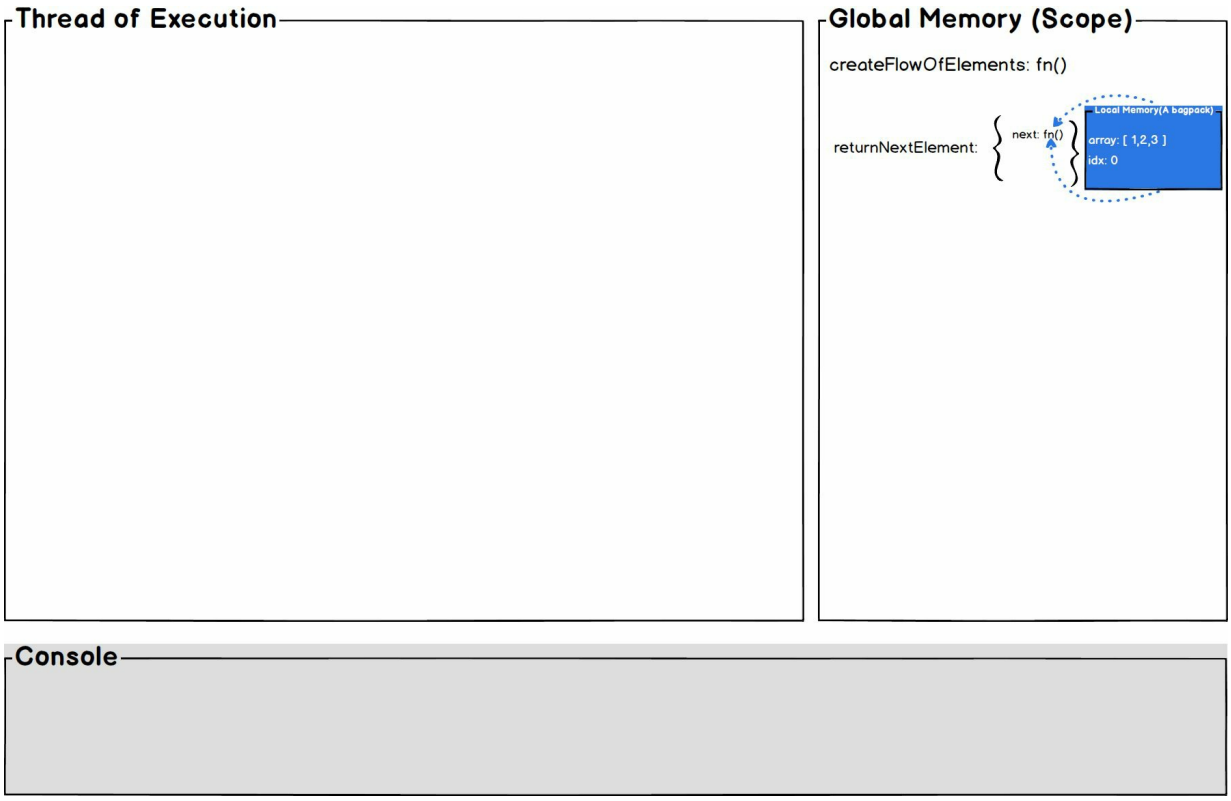


Global Memory (Scope)

createFlowOfElements: fn()

returnNextElement: next: fn()

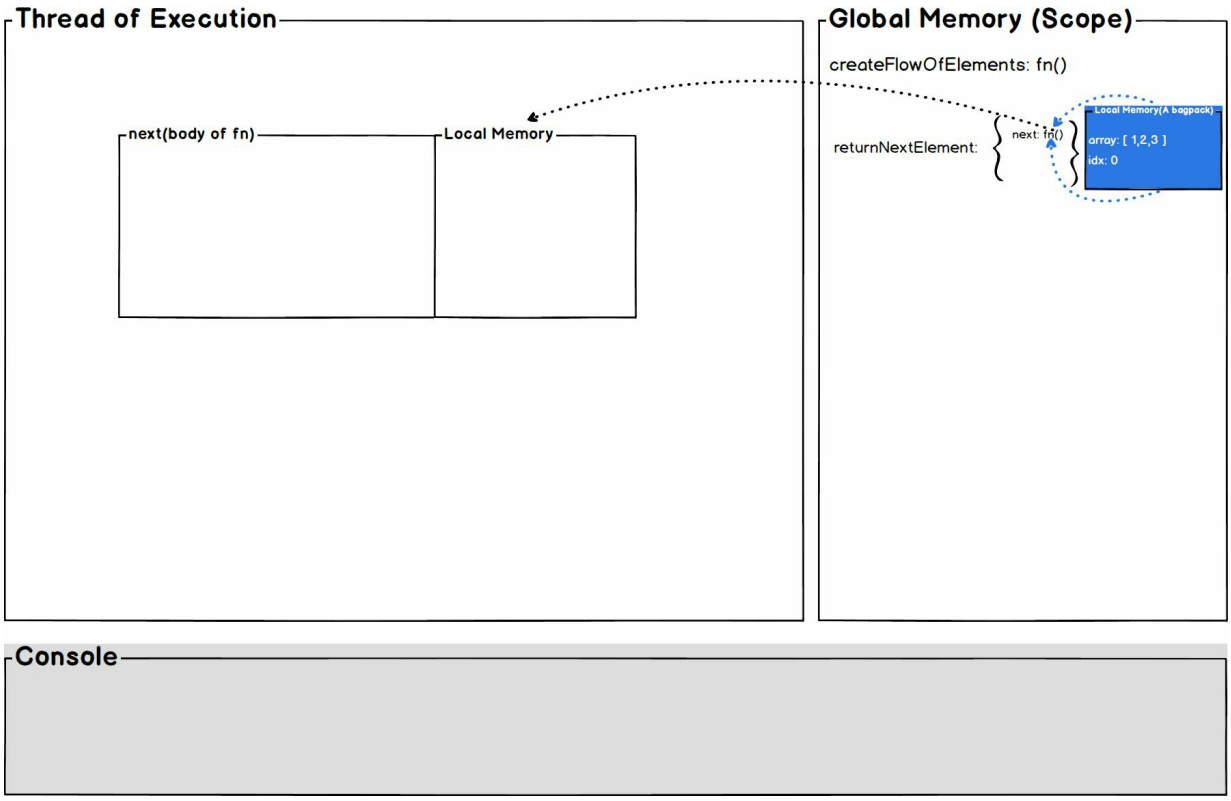
Console



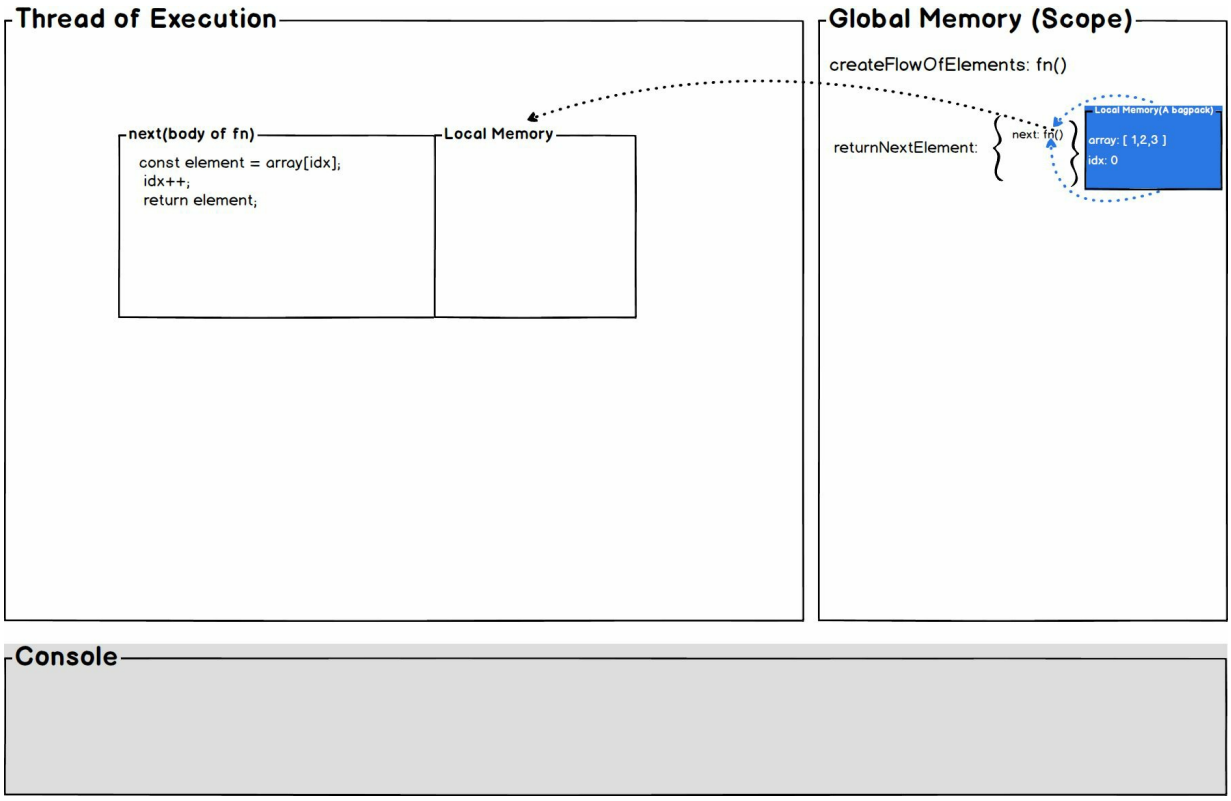
- Amazing, as we can see here **returnNextElement** has closure, and not only that we have the **next** method inside the object as well(people usually call a function of the object as method😊). Also, I hadn't shown two variables of the global scope which are **element1** and **element2** because real visualization of those variables was not needed in the above diagrams.

```
133   const returnNextElement = createFlowOfElements([1,2,3])
134   const element1 = returnNextElement.next()
135   const element2 = returnNextElement.next()
```

- Now, It's time to see what happens when we execute the next method of **returnNextElement**.

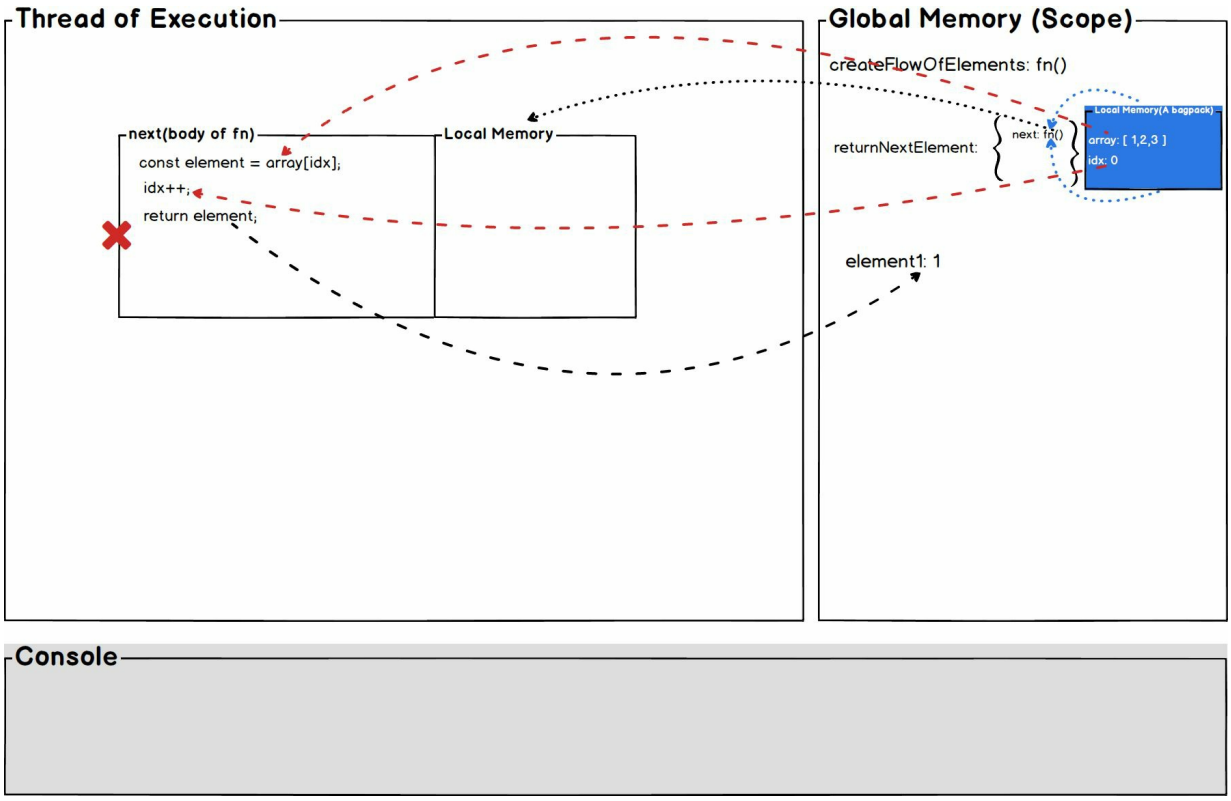


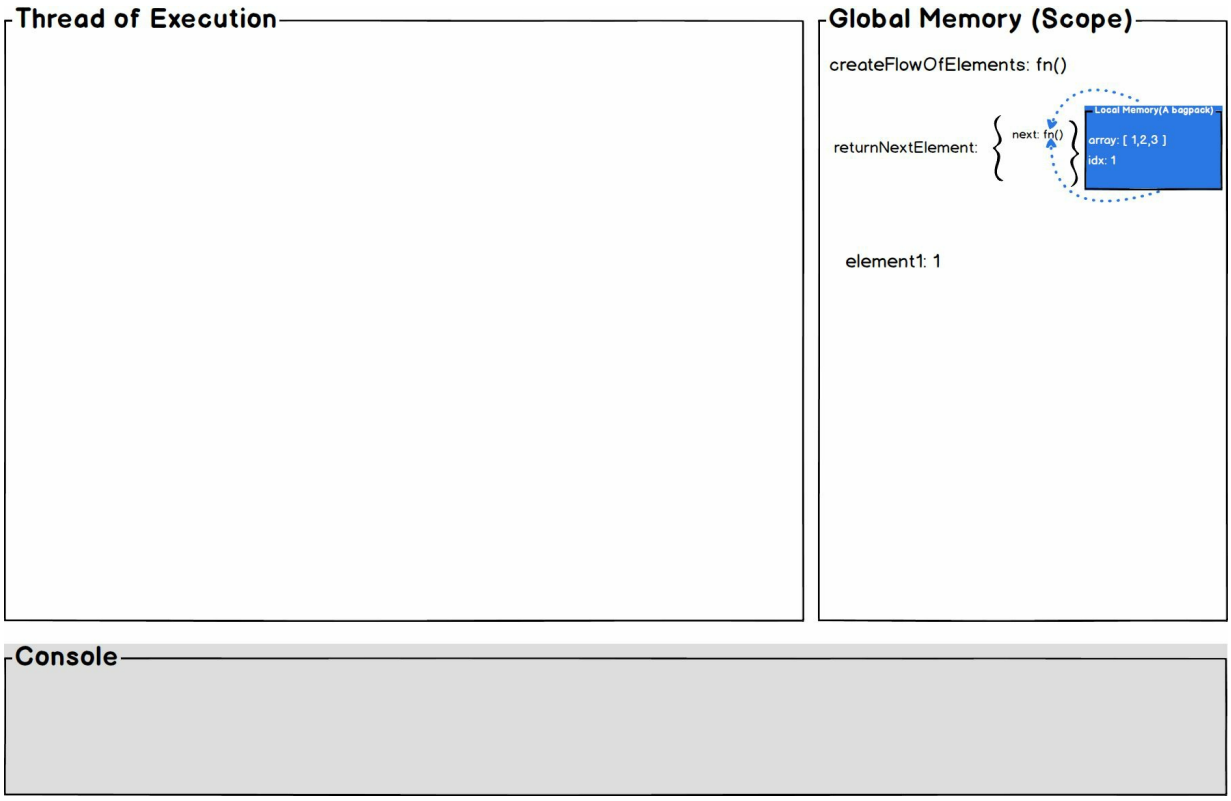
Console



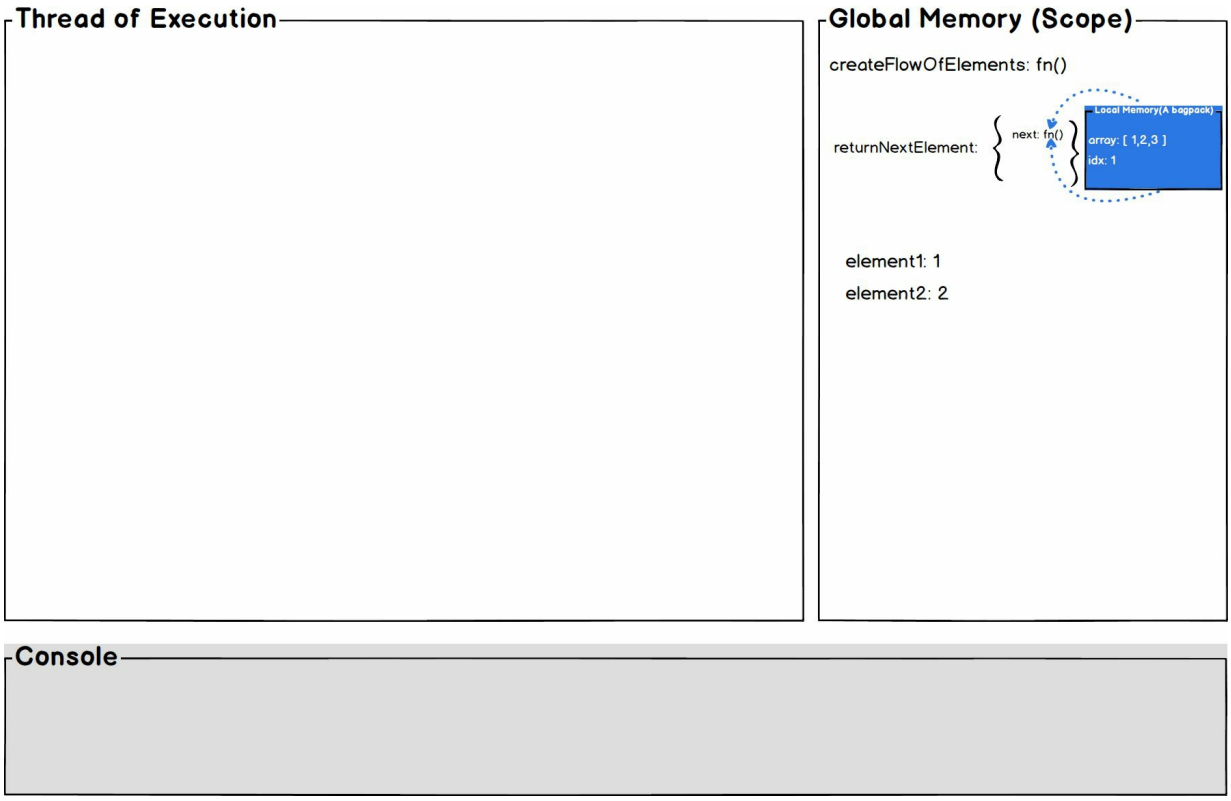
Console

- Because we don’t have variables inside the function’s local memory so we look into our closure to find those variables and here we find it.





- Now, for **element2** same things happen so I am not going to draw a diagram for that😊. Here is our final diagram after all execution finishes.

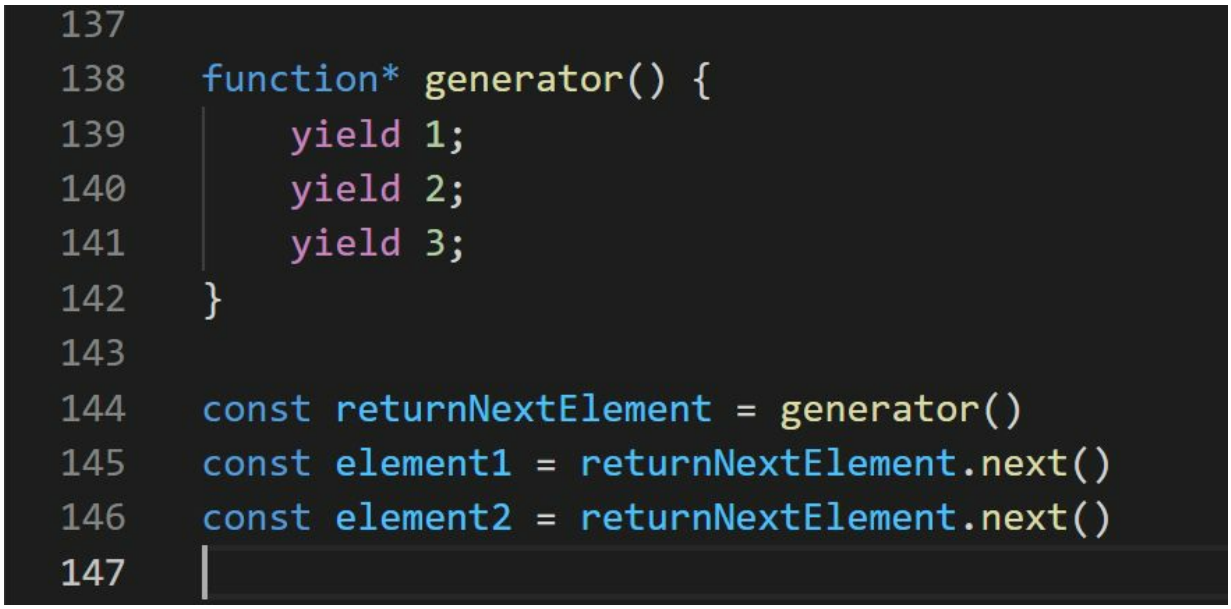


- Great, we just made our first custom iterable, that is to say, now we have a mechanism to control the flow of data, and instead of doing all stuff in just a shot. So, instead of throwing it all elements into the pipe directly now, we have a switch that we can use to get the element. We are controlling our flow of elements😎.
- This is the reason why iterators are important we can get control. It's a powerful mechanism and will be helpful to build some ES7+ features from the scratch(will see soon).
- Also, you can imagine an **event loop is maintained through iterators, where we are literally iterating over queued methods one by one and make them available to callstack😁.**

Generators

The **generator** is just simply a function that returns an iterator! In the above example, **createFlowOfElements** was literally a generator function that returned an iterator😯!

ES7+ has a native generator function. Let's see a simple example of JavaScript's native generator.



```
137
138  function* generator() {
139      yield 1;
140      yield 2;
141      yield 3;
142  }
143
144  const returnNextElement = generator()
145  const element1 = returnNextElement.next()
146  const element2 = returnNextElement.next()
147
```

- As you can see from the above example, native generators are pretty simple looking and easy to write. But there is little twist how native generator works from our custom made generators.
- The first major difference is a native generator and a special kind of function (you see ***** in function definition). Native generator function relies on hidden/internal properties (Remember functions are objects 😊).
- The second difference is, a generator function is not put in callstack, but it's store in memory and put directly on the thread of execution😯!

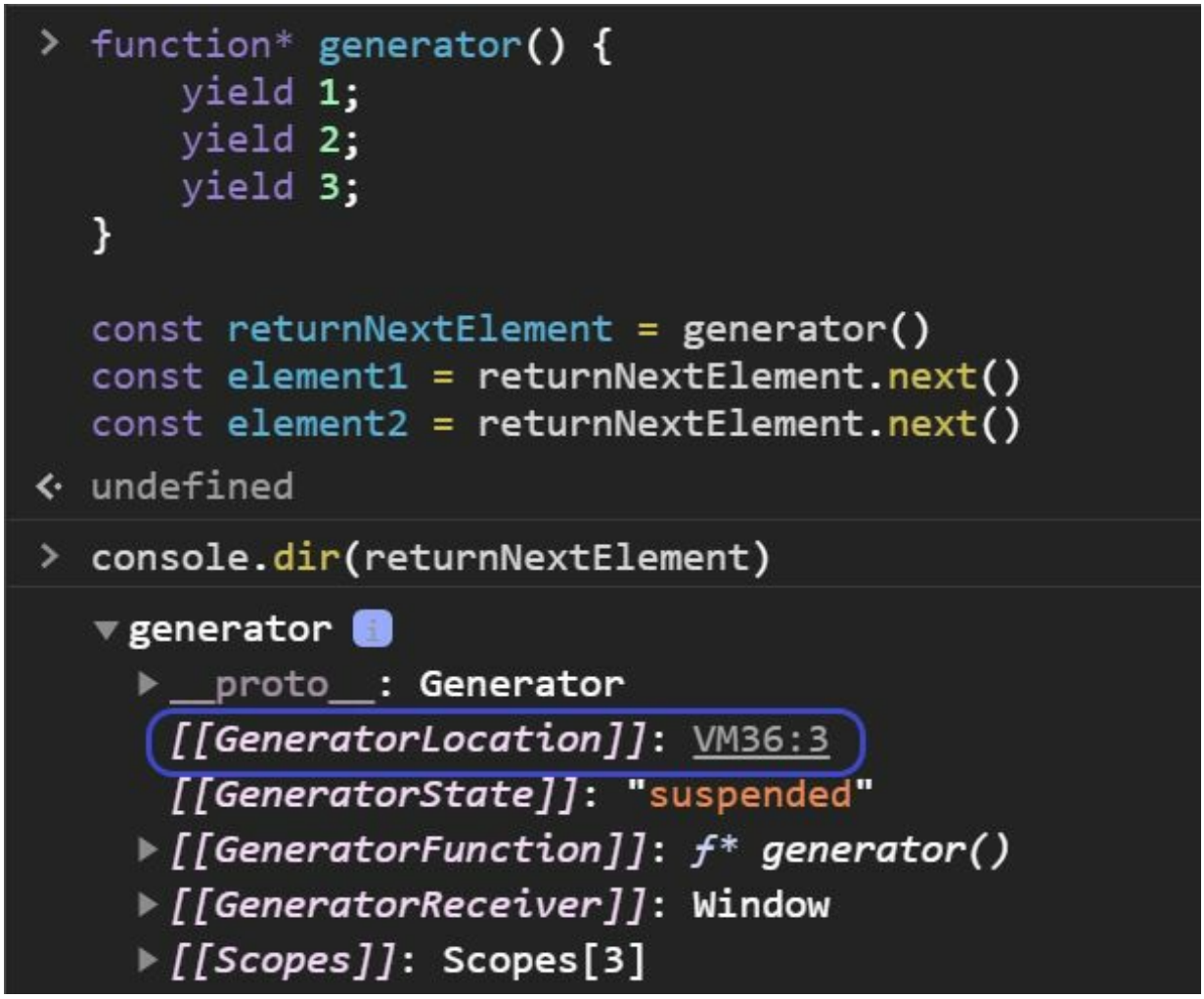
- The **yield** is a special type of keyword that does a lot of hidden stuff behind the scene. Also, it is completely different from the **return keyword.**

return

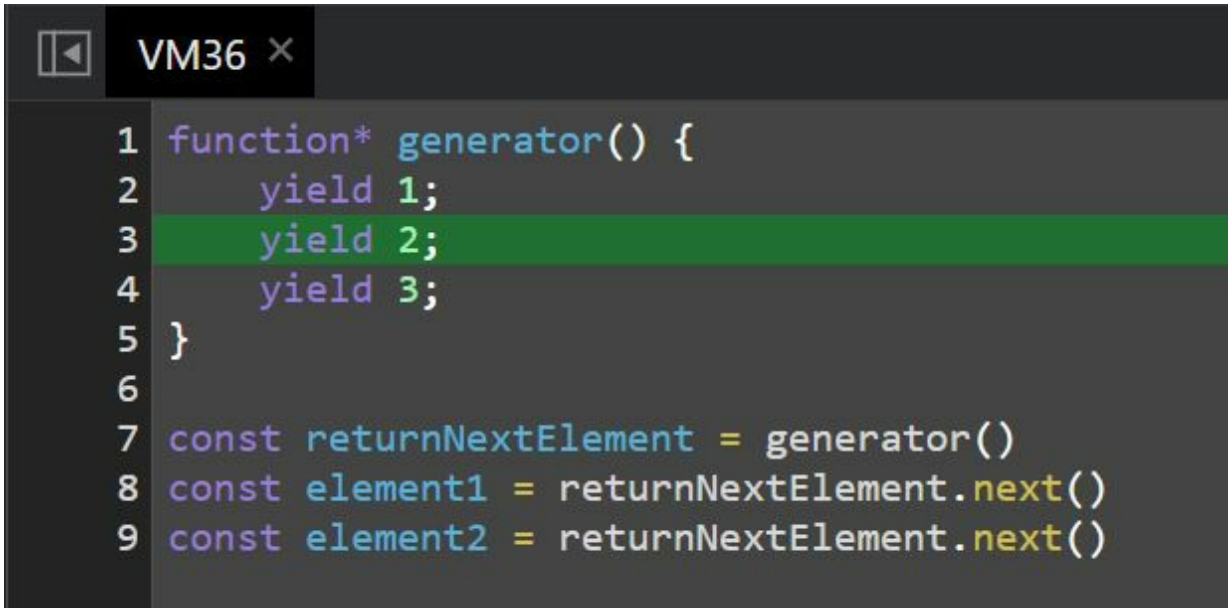
- return keyword immediately stops the function execution and literally allows you to return any value from a function.
- the return keyword is mostly used to do some stuff and the result of the operation is usually return to the outer scope.

yield

- This is a special type of keyword and used inside the generator function.
- yield keyword **will not stop the execution of the generator function** but it will **suspend the execution(pause the execution)**. so, we can later use the same suspended execution context.
- Now, when we **suspend** function execution(kind of pausing the execution of a function), we need some kind of mechanism to keep track of where exactly our execution was suspended. so, in the next iteration, we will able to start execution again from where it is suspended (paused).
- We have a hidden/internal property called ***[[GeneratorLocation]]*** *available on the* ***returned iterator from the generator function.***
- **yield** keyword will suspend the execution and **store the location where function execution is suspended** in ***[[GeneratorLocation]]***. let's see😫.



- The reason we have **[[GeneratorLocation]] at** VM36: 3, because you can see we have called **returnNextElement.next() 2 times. So, a** 2-time value was yielded and the function was paused and continued. So, in the next iteration, we will start our execution from the 3rd line (rather at the 3rd position of yielded value).
- If you click on VM36:3 (VM36 => this is randomly generated by the console of the browser), we will able to see where our last suspension of function is.



Now, what if we have some other stuff inside the generator function? like console.log() or any other operation!

- As you can see, it works exactly like before but now we have ***[[GeneratorLocation]]*** relative to our function definition. This means it is literally keeping track of line number where a function is paused!.
- Now if we click on that VM107:4, we can see below where it's paused. again because we have used **returnNextElement.next() 2 times,** that is why we have paused after at second yielded value.

VM107

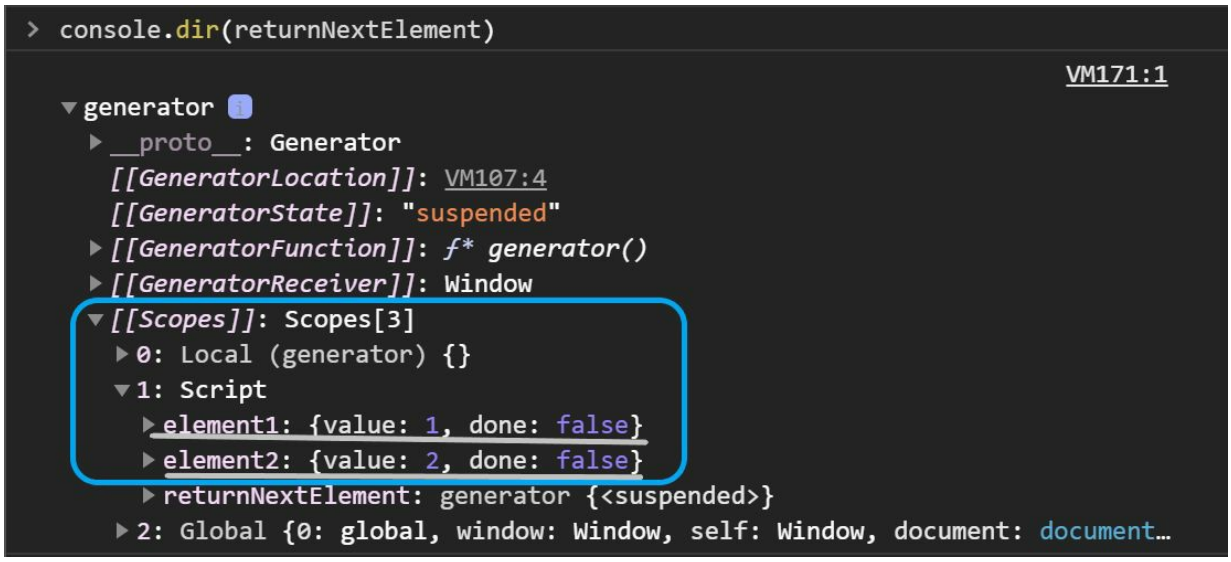
```
1  function* generator() {
2      console.log("Hello my friends😁")
3      yield 1;
4      yield 2;
5      yield 3;
6  }
7
8  const returnNextElement = generator()
9  const element1 = returnNextElement.next()
10 const element2 = returnNextElement.next()
```

- We still haven't seen what actually inside the **element1** and **element2** variables😂. Let's see.

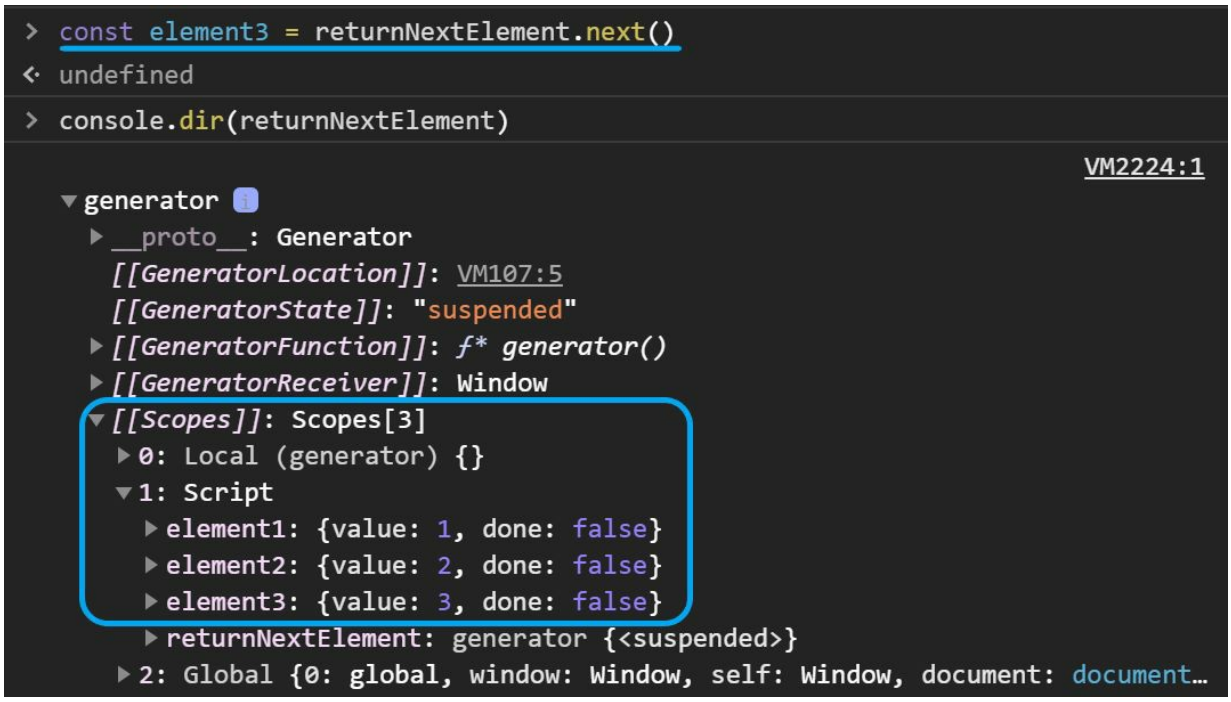
```
> element1
<· ▶ {value: 1, done: false}
> element2
<· ▶ {value: 2, done: false}
```

- Unlike our previous custom generator where we returned values directly, here we are returning an object with some key-value pairs.
- We have value property itself and done property.
- The **value** holds the whatever was yielded and done tells us is there anything remained of execution of paused execution function!
- But, wait we don't have closure here then where the hell this object is

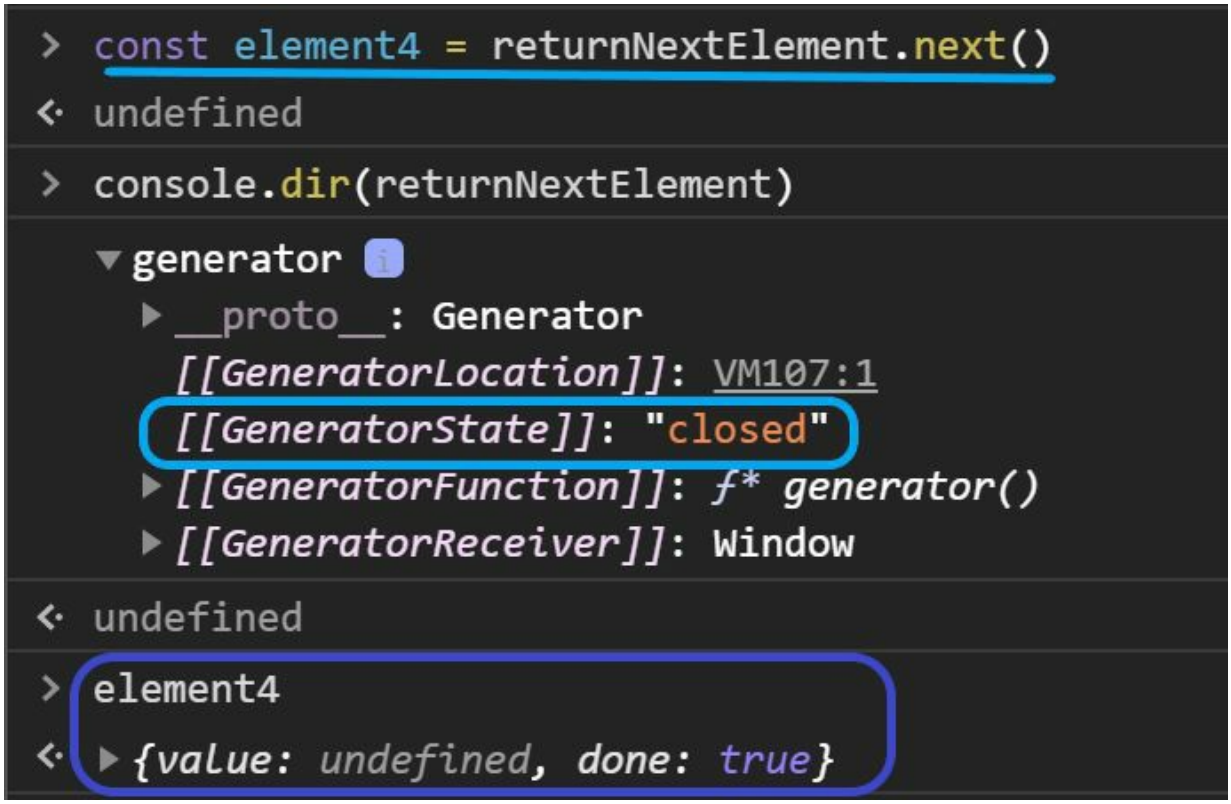
retrieved! let’s see.



- As you can see we are storing yielded values and keeping them inside hidden property called ***[[scopes]].***
- now what if we run **returnNextElement.next()** one more time and see the result.



- now again what if we run **returnNextElement.next()** one more time and see the result.

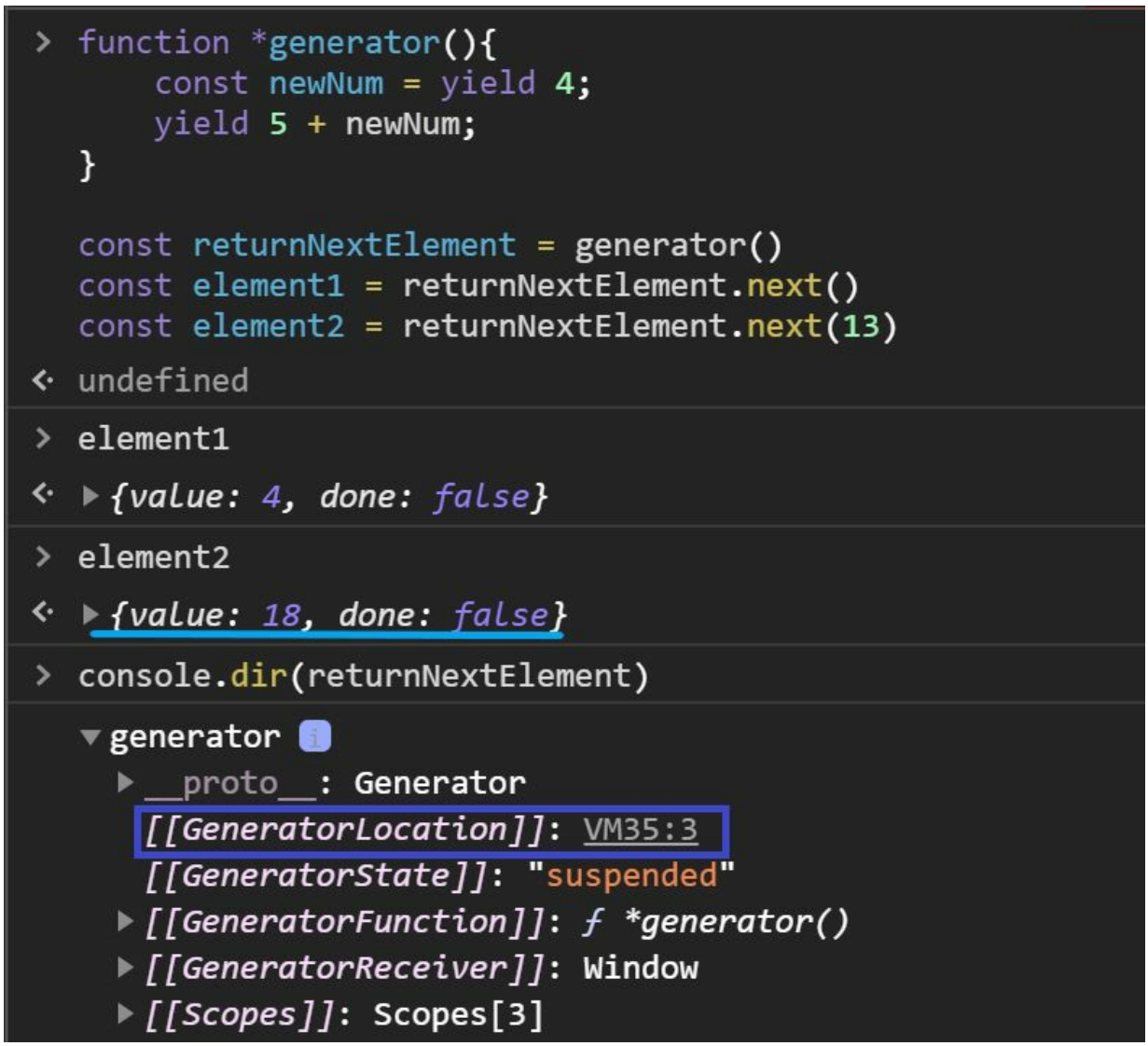


- As you can see that when we finished all yielding of values, we **closed** (finished) the function execution, and the **done** property became **true.**
- **So, long story short** generators allow us to create **custom iterators** and we can **use the next method on our iterable to retrieve yielded values**.
- We need to explore one more thing in order to get a full idea of generators and iterators. Let’s look at the below example.

```
172
173 function* generator(){
174     const newNum = yield 4;
175     yield 5 + newNum;
176 }
177
178 const returnNextElement = generator()
179 const element1 = returnNextElement.next()
180 const element2 = returnNextElement.next(13) // pass 13 as argument to next function
181
```

The common thinking would be, we will **yield 4** in the first next() call and in the second call of next() we will have the **newNum** variable to be **4**. so, the second next() call should yield us to **9** (5 + 4 == 9). unfortunately, that is not how it works😅.

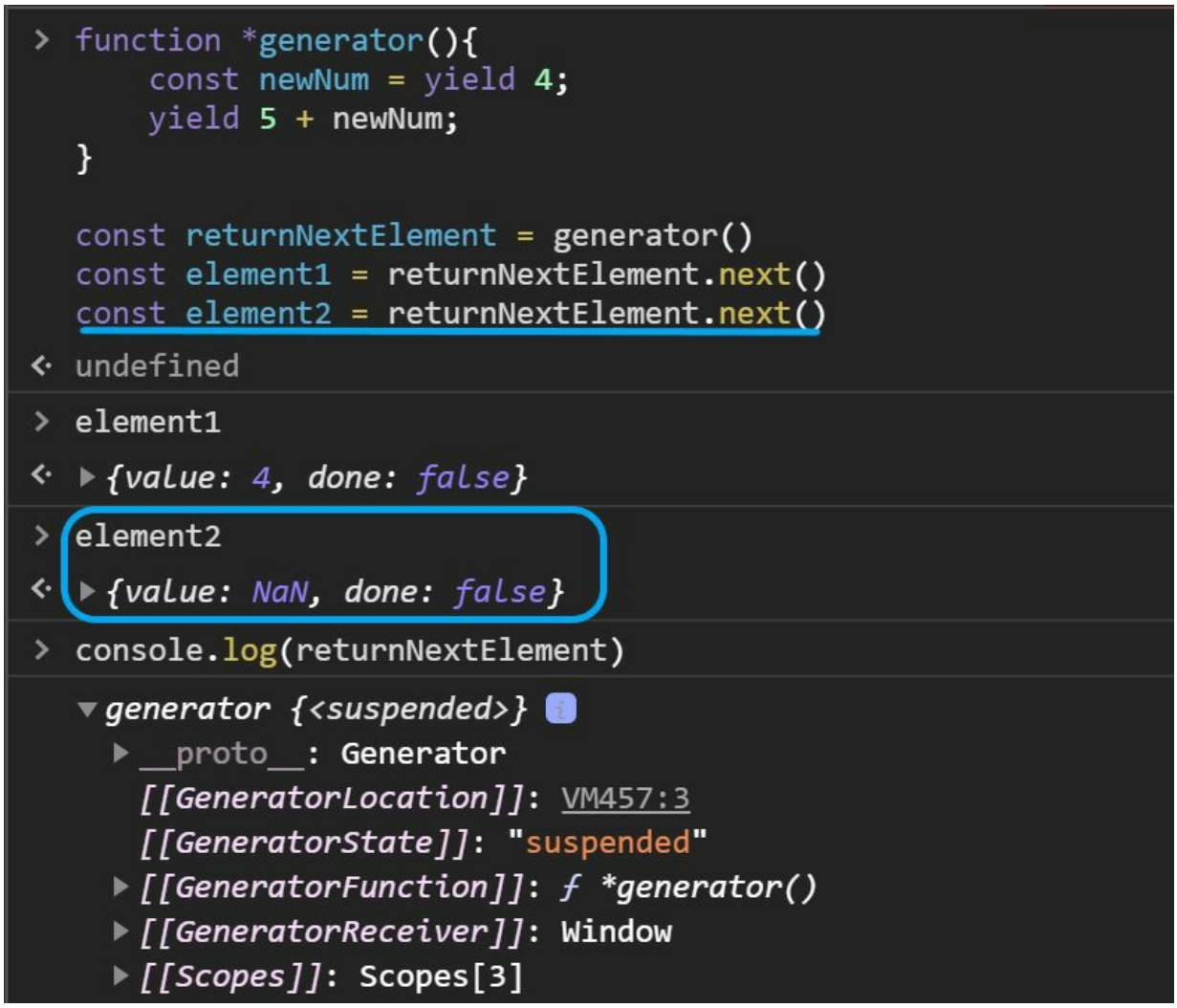
let’s see what we get in the output.



- As you can see in the second call of next() it has a totally different output than our expectation. The reason why it happened because the iterator returned from the generator function will allow you to pass an argument to the next() function and whatever we pass to the next() function's argument will become available to wherever our function suspended execution!
- so, ultimately **13** that we passed to second next() call as an argument made available and got stored in **newNum**. so, in the next yield, it

became 18 (5 + 13)😊.

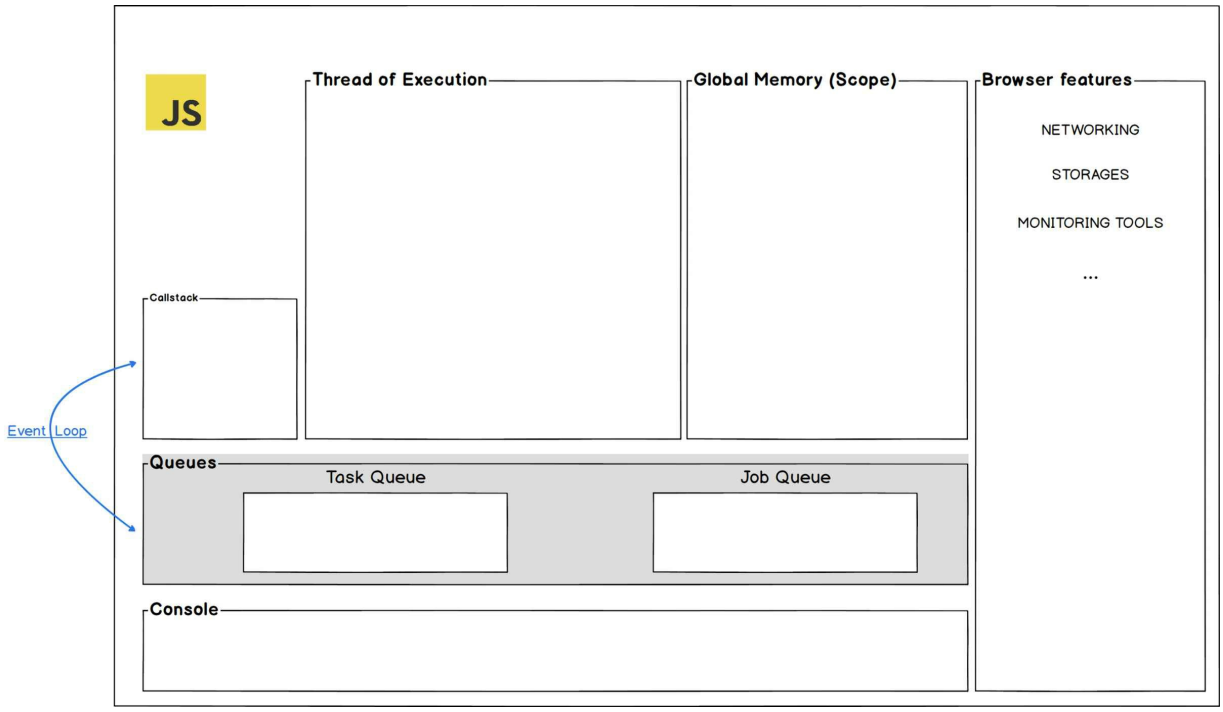
What if we didn't provide or pass any argument to the second next() call ?



- As you can see if we don't provide any argument then **newNum** will have a value of **undefined.**
- **So,** when we second time yielded value it became **NaN(5 + undefined).**
- The above example has a great use case, and that is what we are going to do next.
- Let's build **async-await** from scratch😎.

```
182
183  // ---------------------------------------------------------------------------
184  // assume here we are getting back 'hello' string back from https://test.com
185  // ---------------------------------------------------------------------------
186  function* generator() {
187      const data = yield fetch('https://test.com');
188      console.log(data);
189  }
190
191  const returnNextElement = generator();
192  const dataPromiseObject = returnNextElement.next();
193
194  dataPromiseObject.then((data) => {
195      returnNextElement.next(data);
196  });
197
```

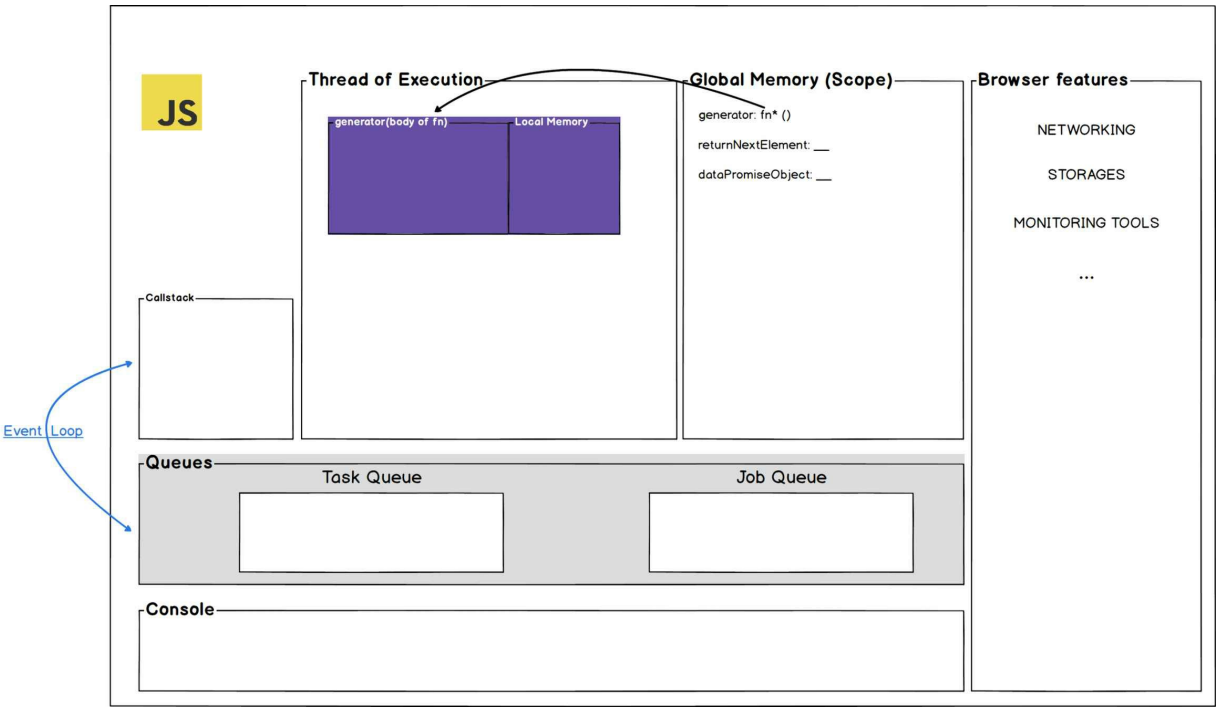
- This time we are going to visualize the above code😊😊.let me tell you ahead of time that the diagram that we are going to draw soon will be the most intense and complicated. It'll involve almost every single concept that we have understood so far😮. Let's start then.



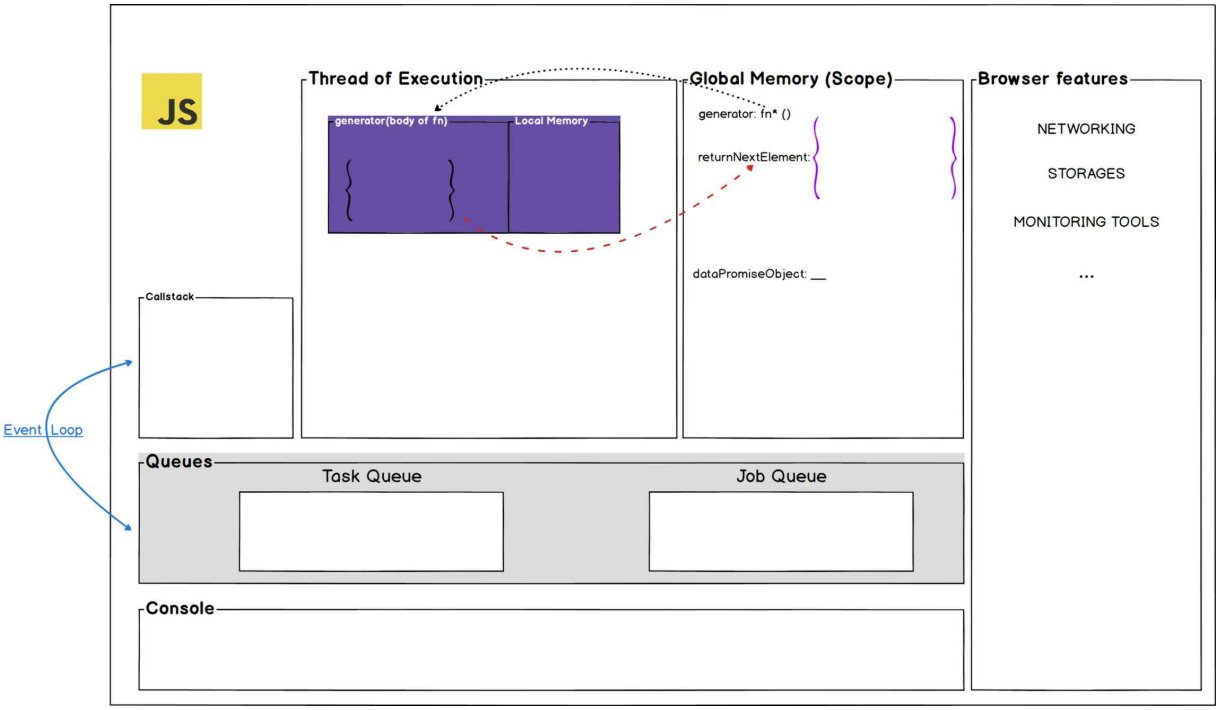
Before we move further, let me tell you some color schemas that we will use.

Dark purple - generator function in execution

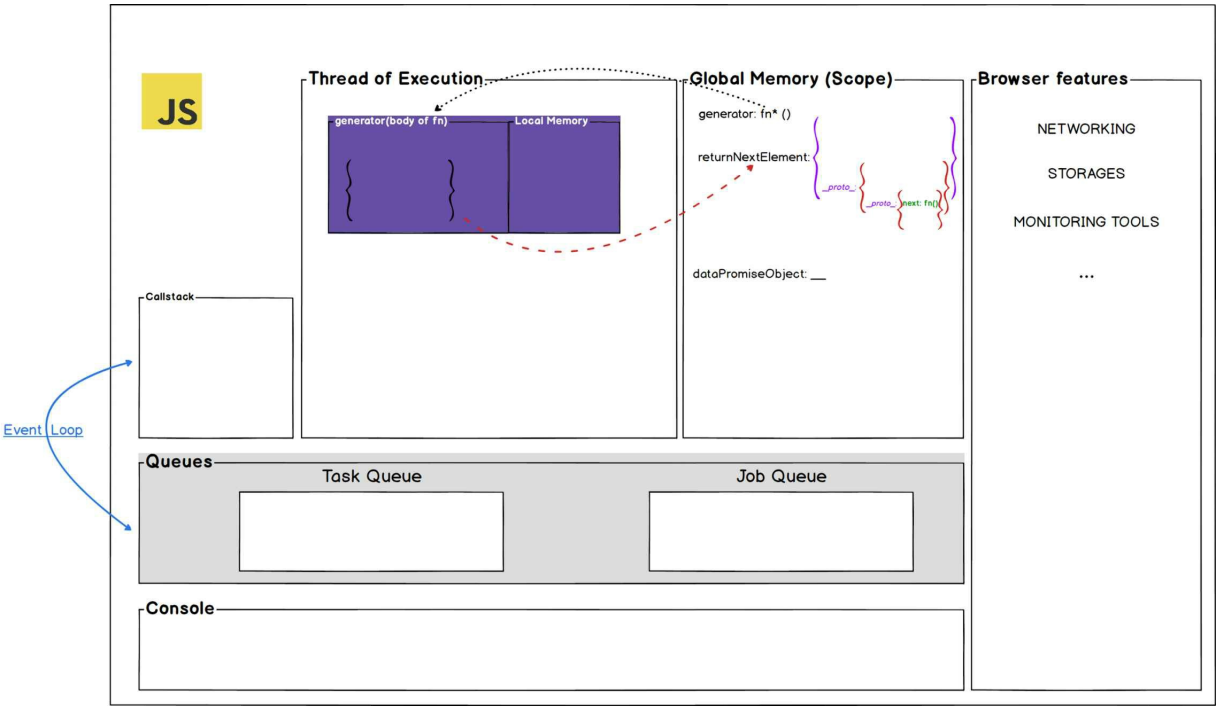
light purple - generator function in suspended state(paused).

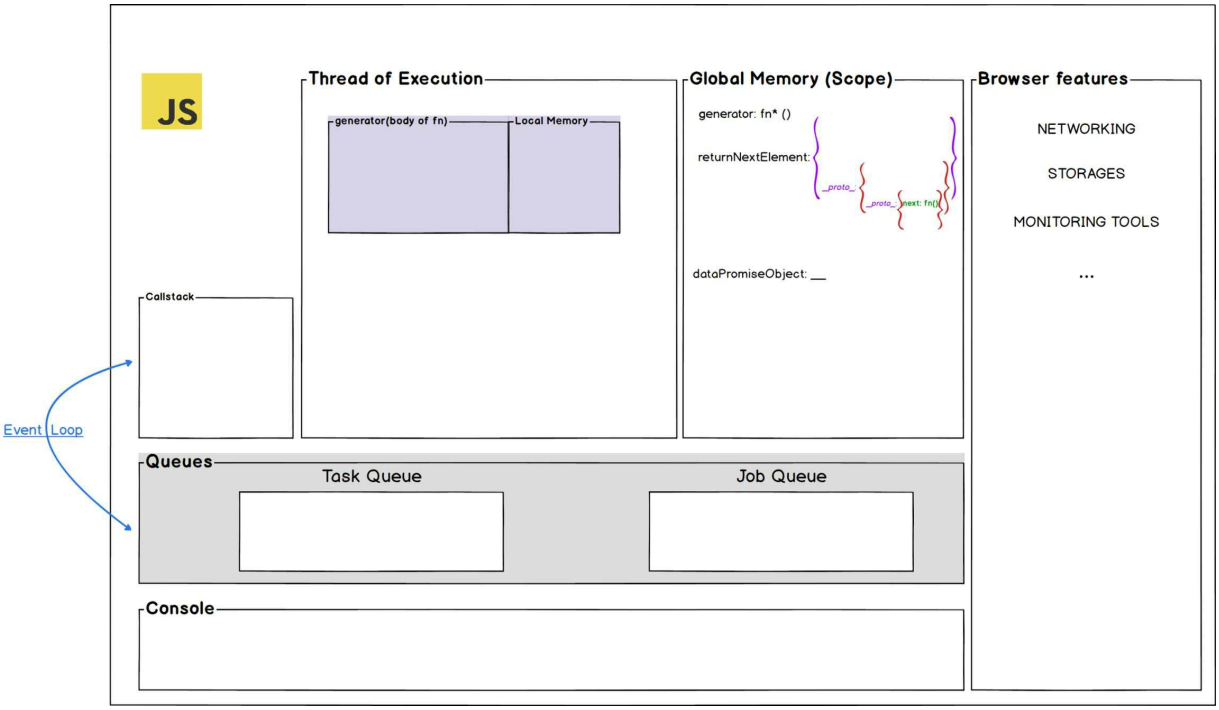


- One more thing, generator functions are not normal functions and they **don’t get garbage collected**(It stays in memory even after its execution is finished).
- Moreover, the execution of the function is put directly on the thread of execution.

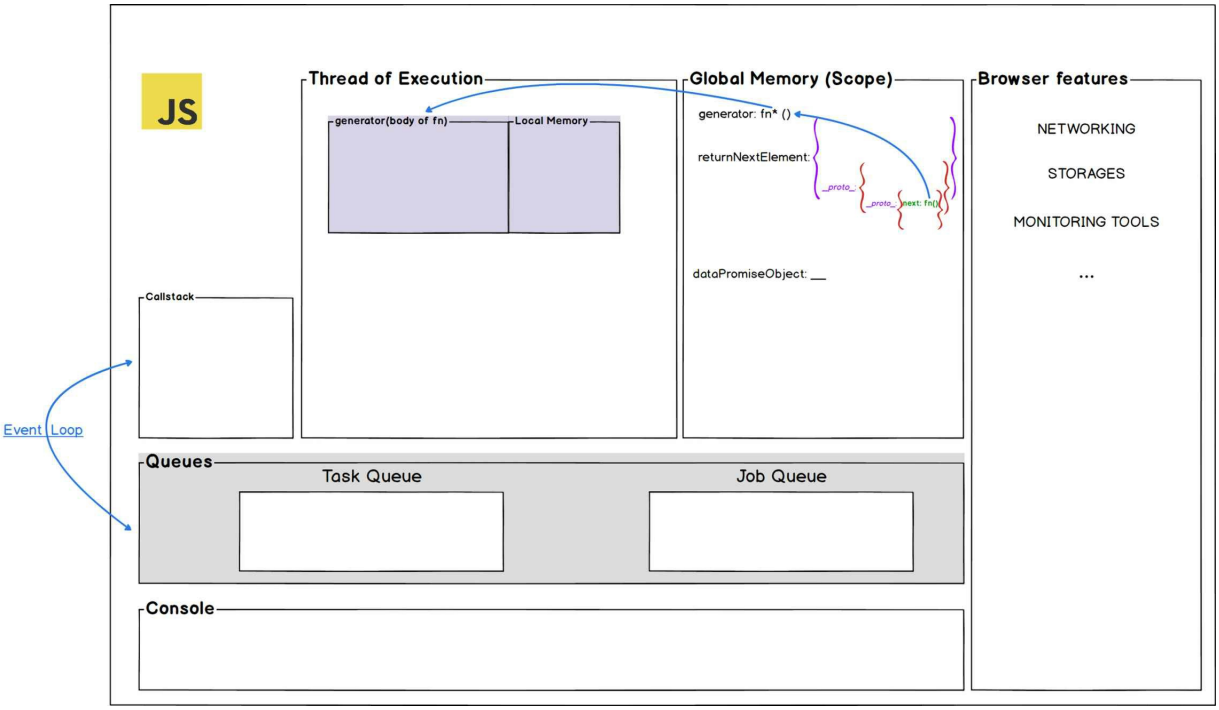


- Regarding that __proto__, it's actually simple. The *generator function object was build from the **"Generator" object(yes, it's direct object).** So, generator function object's __proto__ point to prototype object of Generator(**generator .__ptoto__ = Generator.prototype*).The same way returnNextElement was build from ***generator function**. So, returnNextElement object's __proto__ point to prototype object of *generator function(*returnNextElement.__proto__ = *generator.prototype*).
- Ultimetly, out returnNextElement will have access to next() method via __proto__ chain.

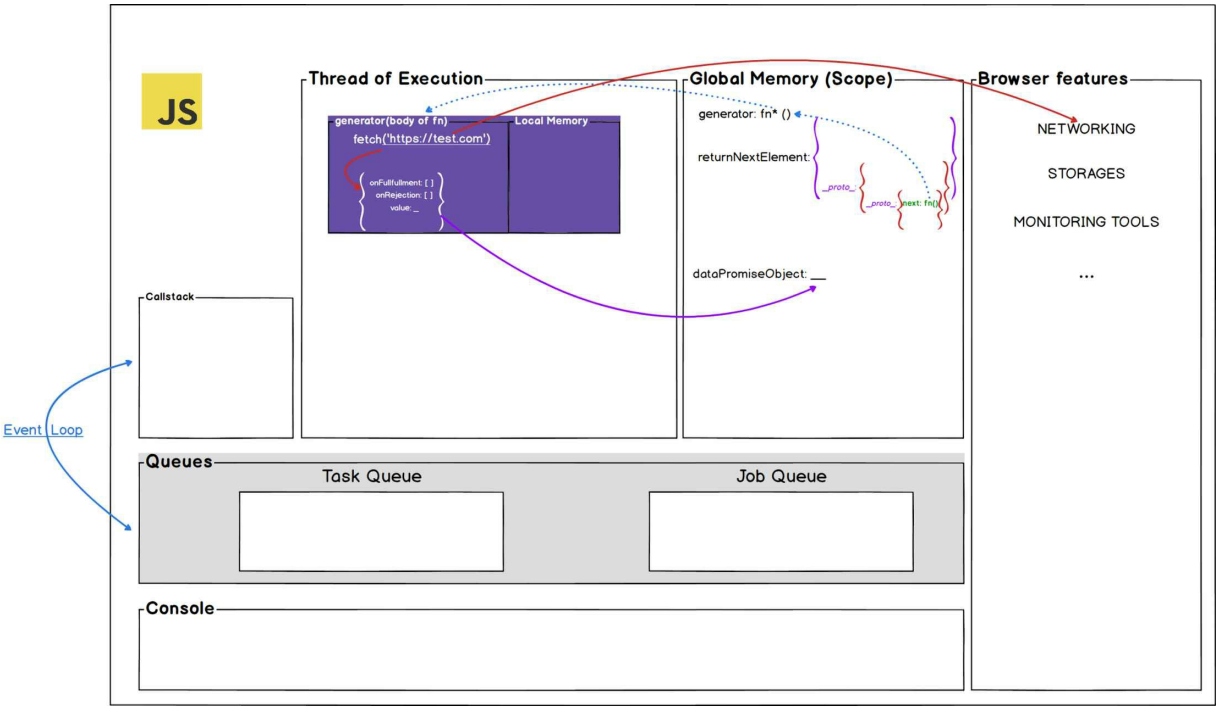


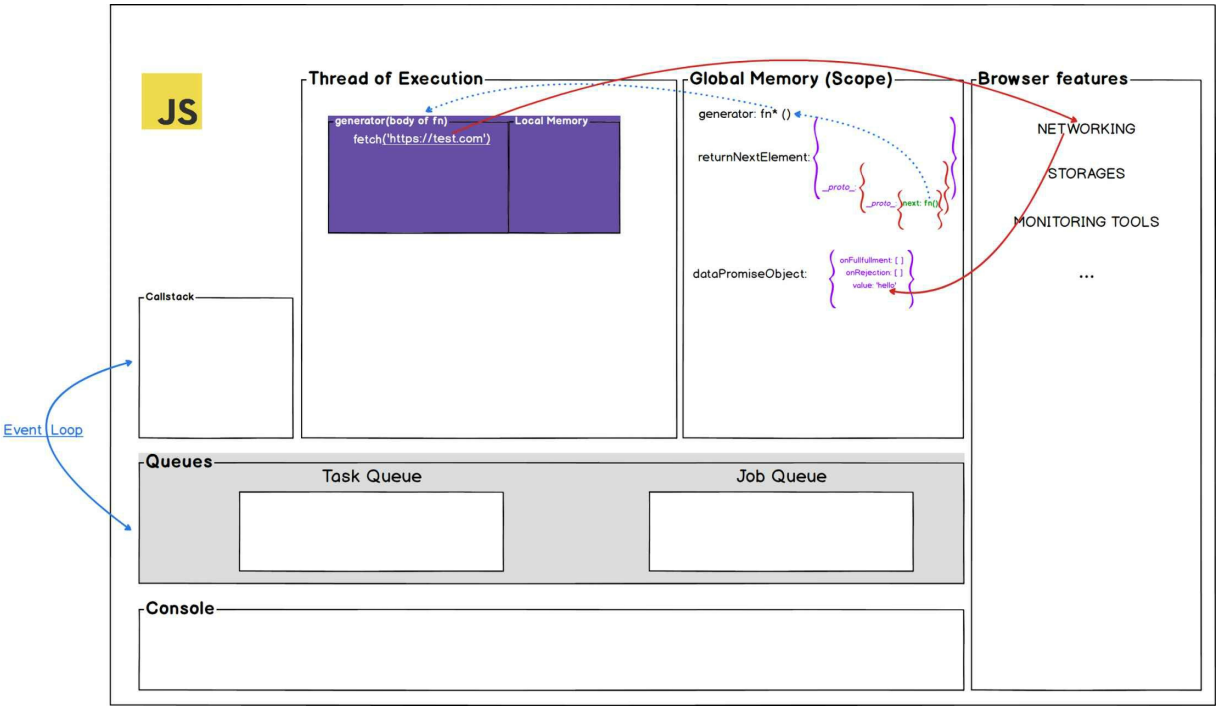


- After finishing the creation of **returnNextElement** iterator when we use the next() method on that part we will hit **fetch() API** part. let's see how it goes.



- See below diagram, where generator function is directly put into execution and not only that our **fetch API create two-way bonding as well(**background networking work + promise object creation in javascript execution thread**).**
- Try remembering fetch API’s mental model from the previous chapter and everything will follow smoothly

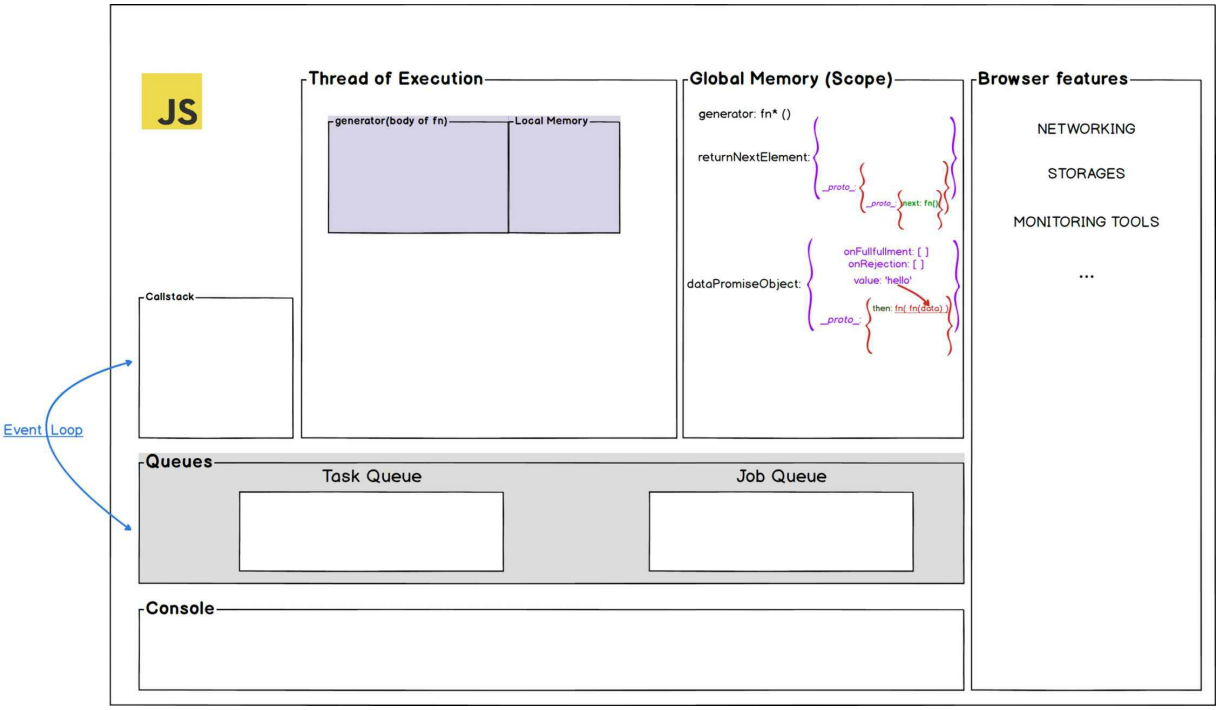


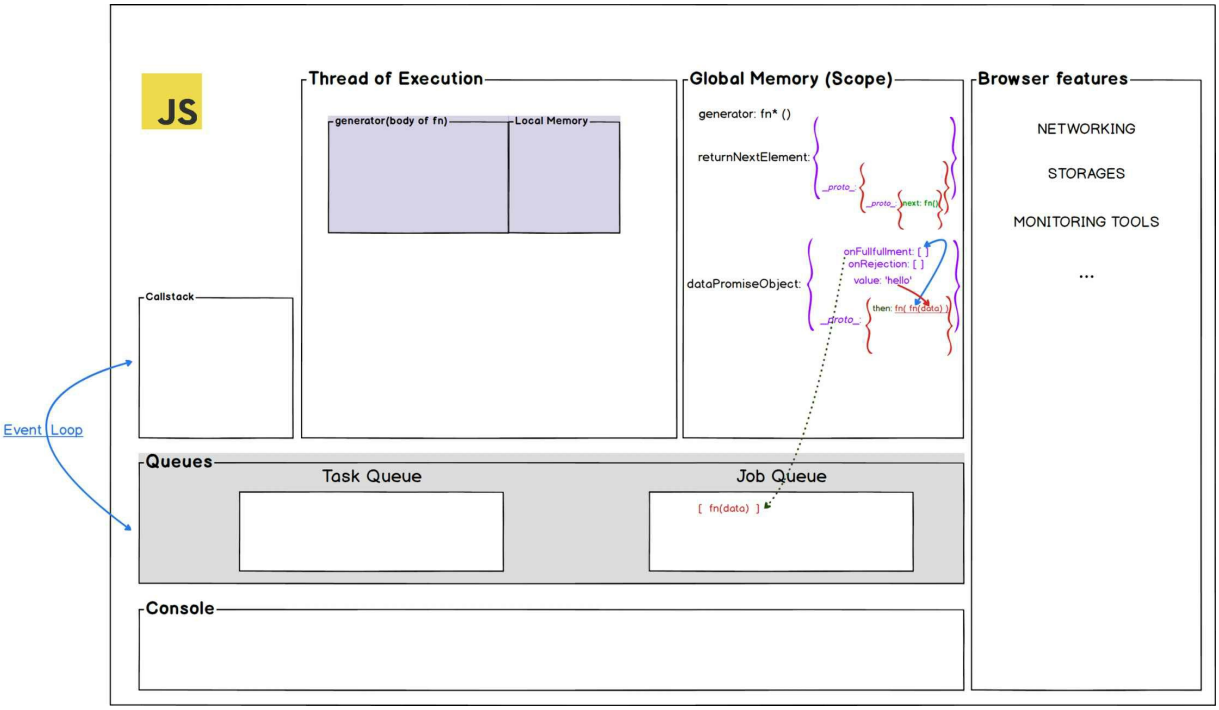


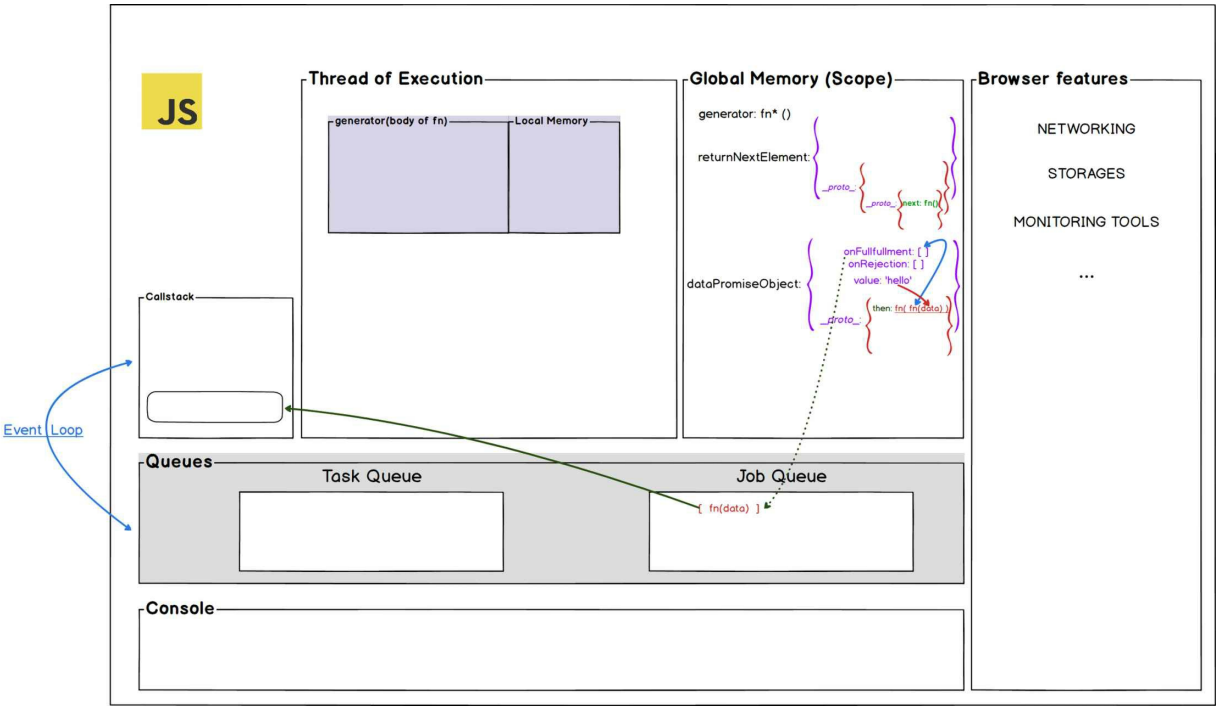
- As we know when we finish working on our networking task, the result will be given back to the promise object's **value** property and the further mental model is the same that we learned in the previous chapter for the

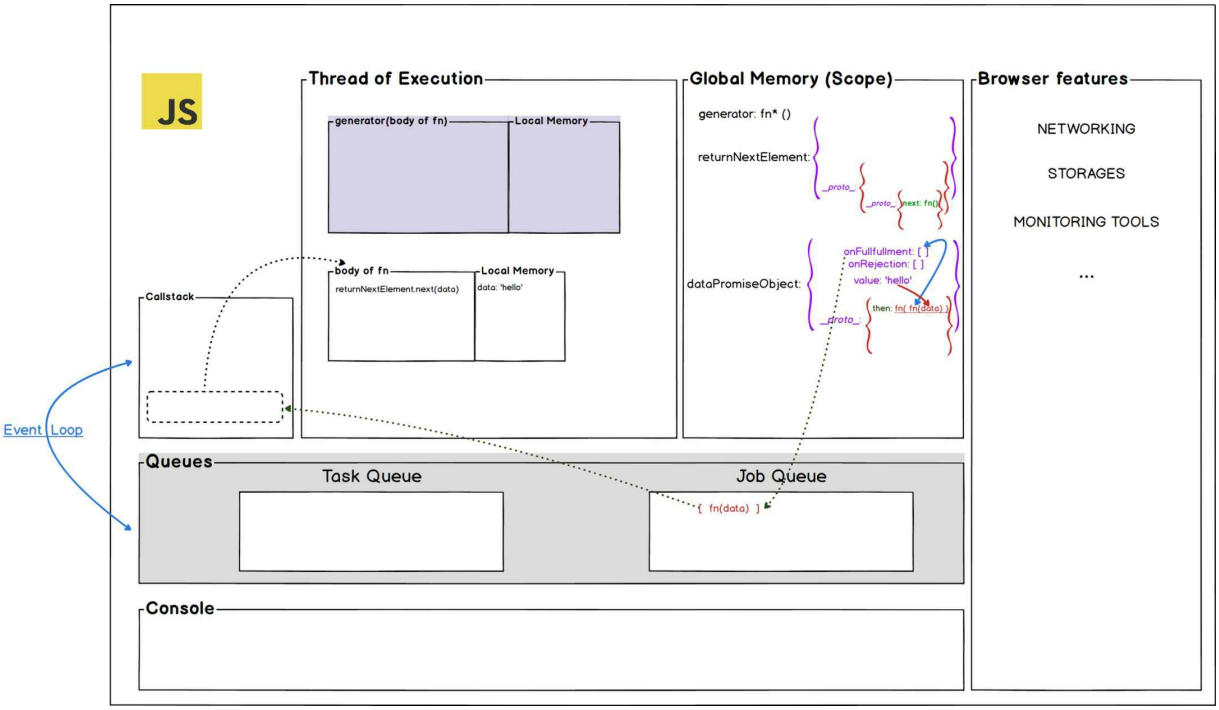
promise object.

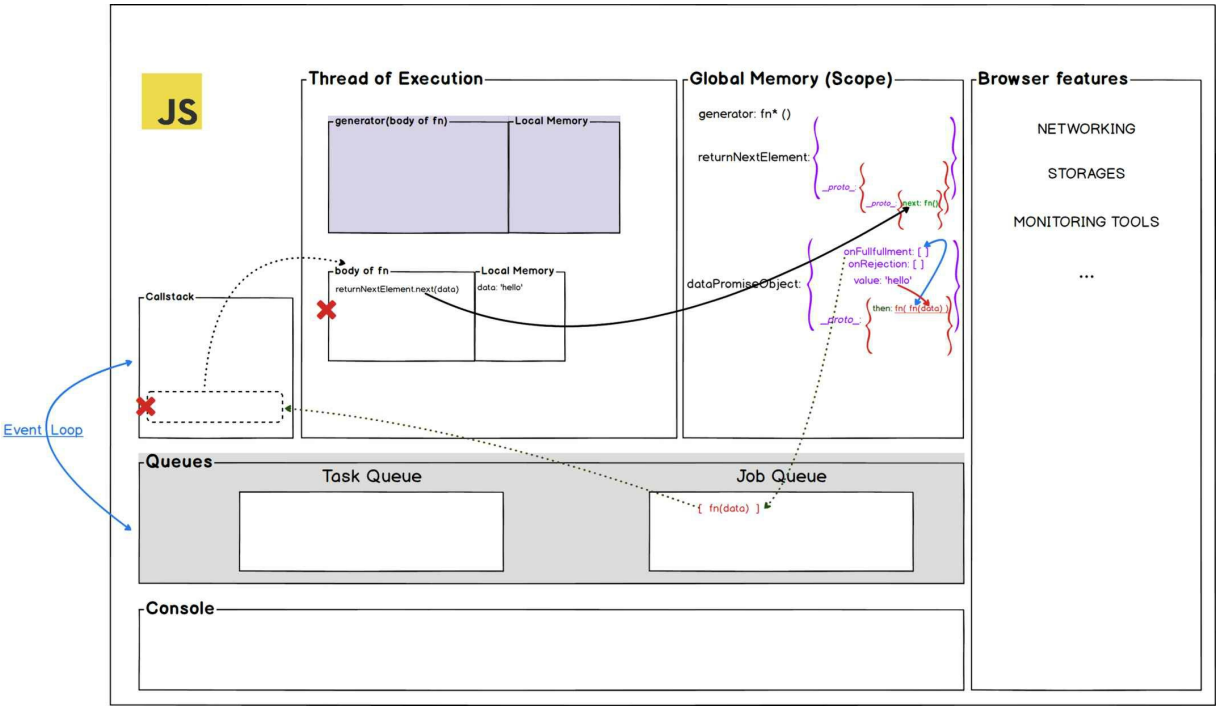
- Just follow the diagrams and you will get it☺.






JS
Thread of Execution
generator(body of fn)
Local Memory
Global Memory (Scope)
generator: fn* ()
returnNextElement:
proto
next: fn()
dataPromiseObject:
onFullfullment: [ ]
onRejection: [ ]
value: 'hello'
then: fn(fn(data))
Browser features
NETWORKING
STORAGES
MONITORING TOOLS
...
Callstack
Event Loop
Queues
Task Queue
Job Queue
{ fn(data) }
Console

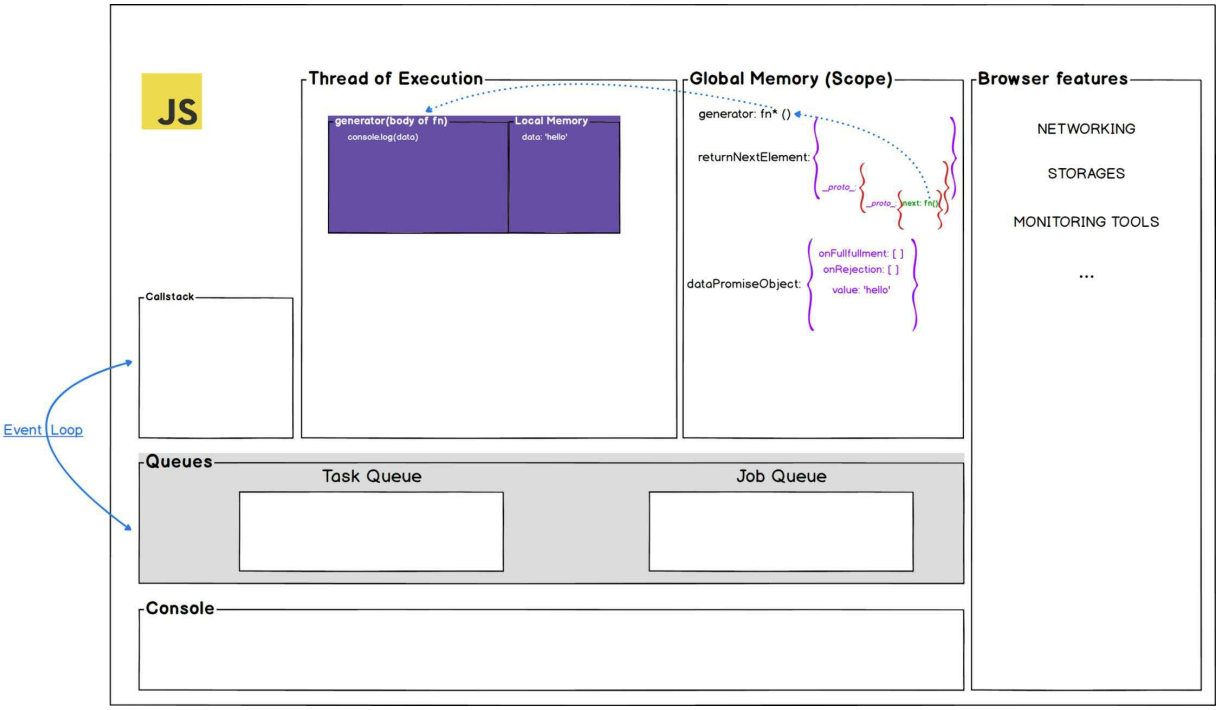

JS
Thread of Execution
generator(body of fn)
Local Memory
Global Memory (Scope)
generator: fn* ()
returnNextElement:
proto
next: fn()
dataPromiseObject:
onFullfullment: [ ]
onRejection: [ ]
value: 'hello'
then: fn(fn(data))
Browser features
NETWORKING
STORAGES
MONITORING TOOLS
...
Callstack
body of fn
returnNextElement.next(data)
Local Memory
data: 'hello'
Event Loop
Queues
Task Queue
Job Queue
{ fn(data) }
Console



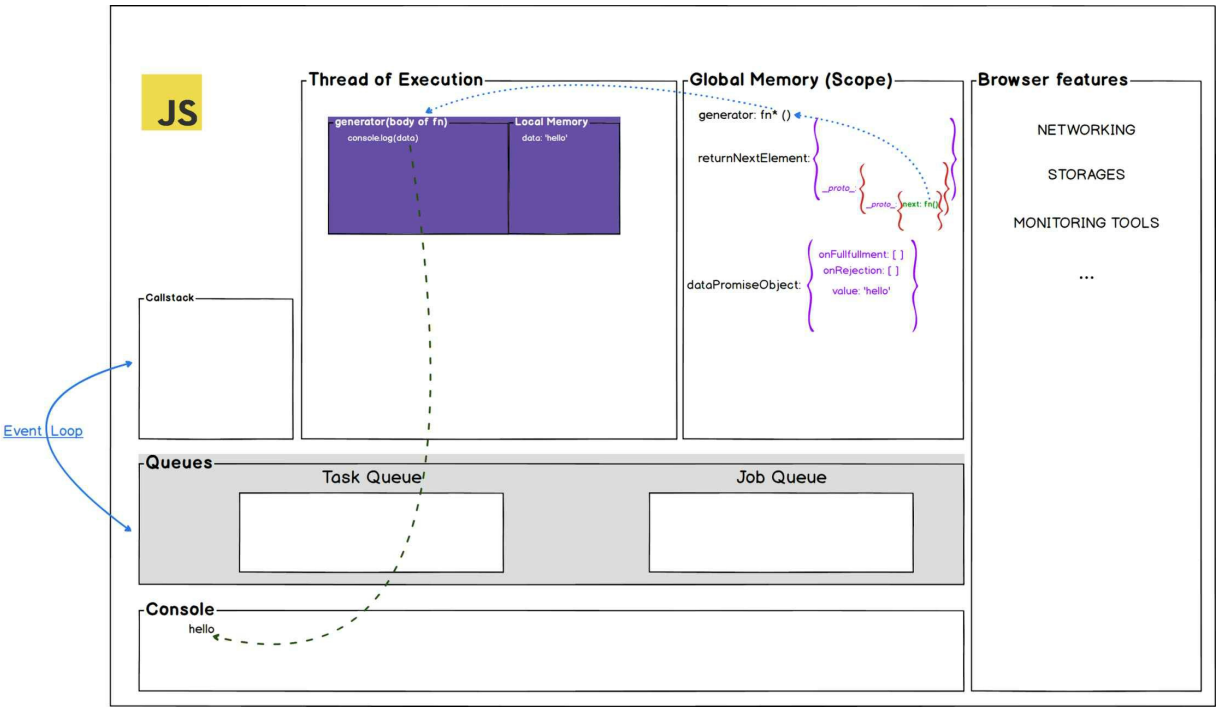
- After finishing the promise object work, now we are calling again the next() method of **returnNextElement** by providing the **data** argument to the next() method. see our example code.

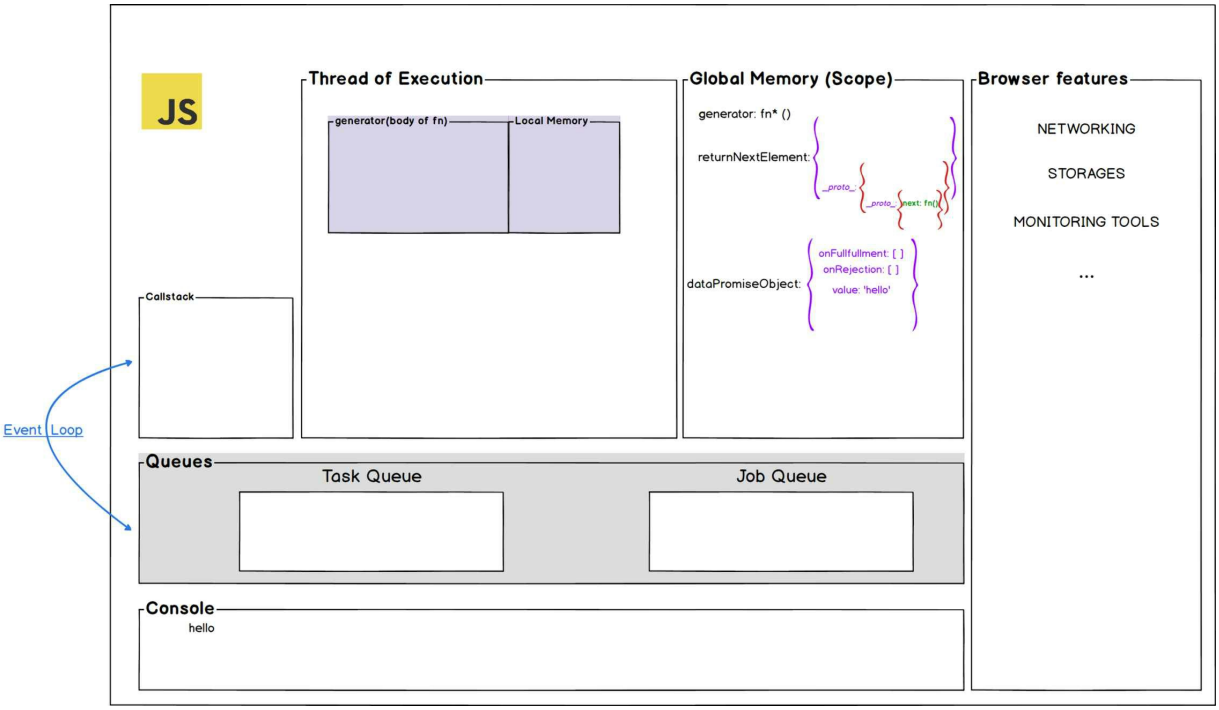
```
182
183  // ------------------------------------------------------------------------------
184  // assume here we are getting back 'hello' string back from https://test.com
185  // ------------------------------------------------------------------------------
186  function* generator() {
187      const data = yield fetch('https://test.com');
188      console.log(data);
189  }
190
191  const returnNextElement = generator();
192  const dataPromiseObject = returnNextElement.next();
193
194  dataPromiseObject.then((data) => {
195      returnNextElement.next(data);
196  });
197
```

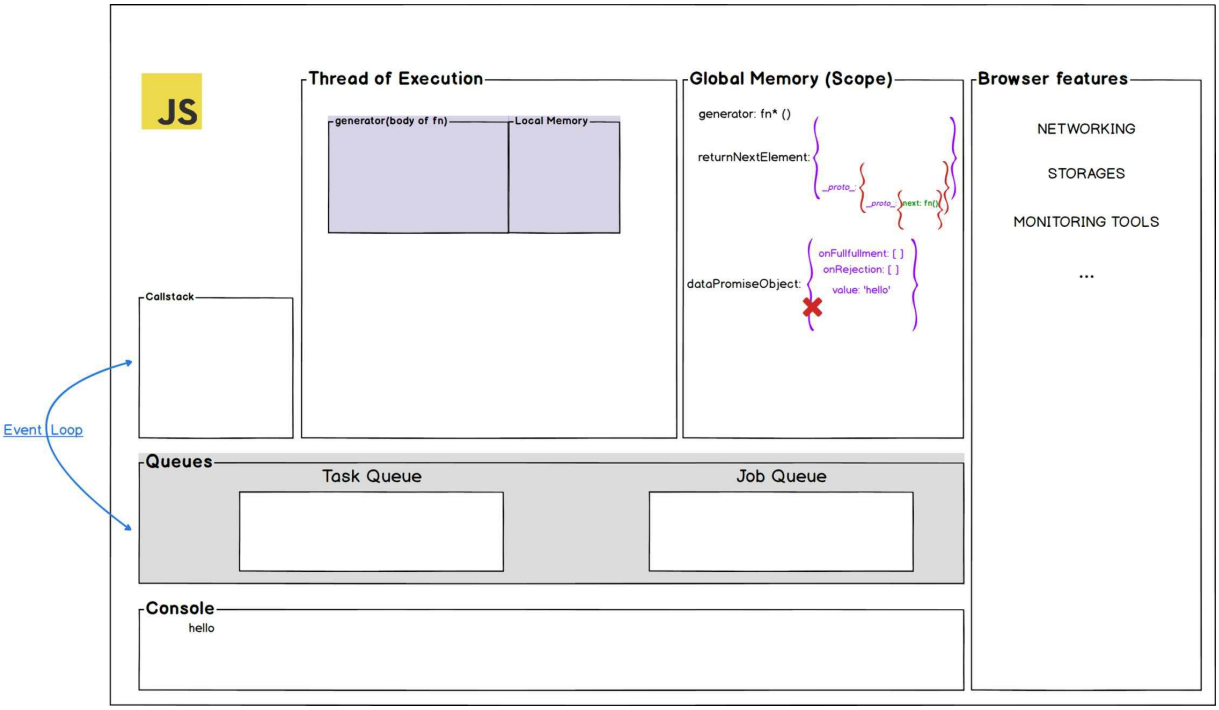
- Let’s see how it goes in further execution.

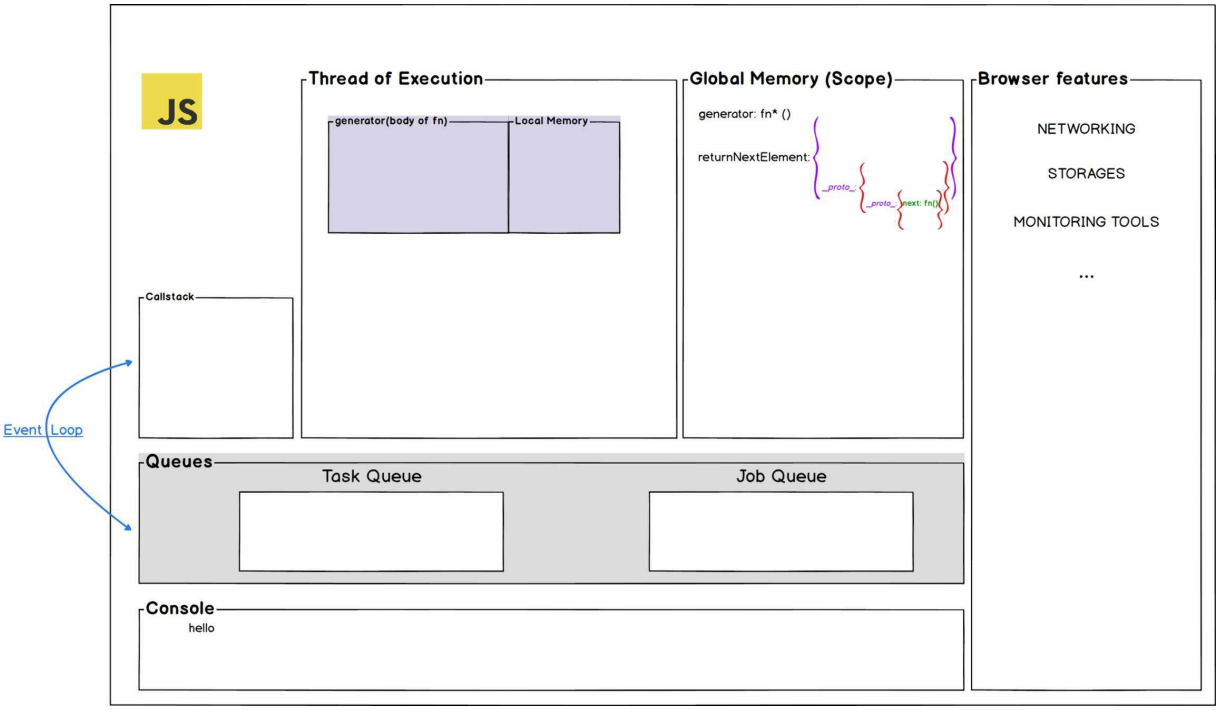


- Because we called the next() method with **data** which we got from the promise object's **value,** we will make available that value to the local variable of the next() iterator.




JS
Thread of Execution
generator(body of fn)
Local Memory
Global Memory (Scope)
generator: fn* ()
returnNextElement:
proto
proto
next: fn()
dataPromiseObject:
onFullfullment: [ ]
onRejection: [ ]
value: 'hello'
Browser features
NETWORKING
STORAGES
MONITORING TOOLS
...
Callstack
Event Loop
Queues
Task Queue
Job Queue
Console
hello


JS
Thread of Execution
generator(body of fn)
Local Memory
Global Memory (Scope)
generator: fn* ()
returnNextElement:
proto
proto
next: fn()
dataPromiseObject:
onFullfullment: [ ]
onRejection: [ ]
value: 'hello'
Browser features
NETWORKING
STORAGES
MONITORING TOOLS
...
Callstack
Event Loop
Queues
Task Queue
Job Queue
Console
hello



There you have it, my friends. we have made our own custom async-await from scratch 😎.(you can do the same with a catch scenario as well, which I am not going to show 😉)

Before we close this chapter let's just see the shorten ES7+ syntax of async-await with example.

```
198  // ------------------------------------------------------------------------------
199  // assume here we are getting back 'hello' string back from https://test.com
200  // ------------------------------------------------------------------------------
201  async function generator() {
202      const data = await fetch('https://test.com');
203      console.log(data);
204  }
205
```

- As you can see, the **async** keyword will behave like a generator and the **await** keyword will behave like the **yield** keyword from our example. That been said this is all under the hood and deep dive for this chapter ☺.

* * *

6

NodeJS, C++, Queues & Servers

If someone would ask me, what is the best thing that happens in the javascript community?, my immediate answer will be it is none other than NodeJS!.NodeJS is an amazing javascript runtime and also a little misunderstood tool. In this chapter, I will try to make it as easy as possible and explain the core architecture and fundamentals of the nodejs.

What node is not?

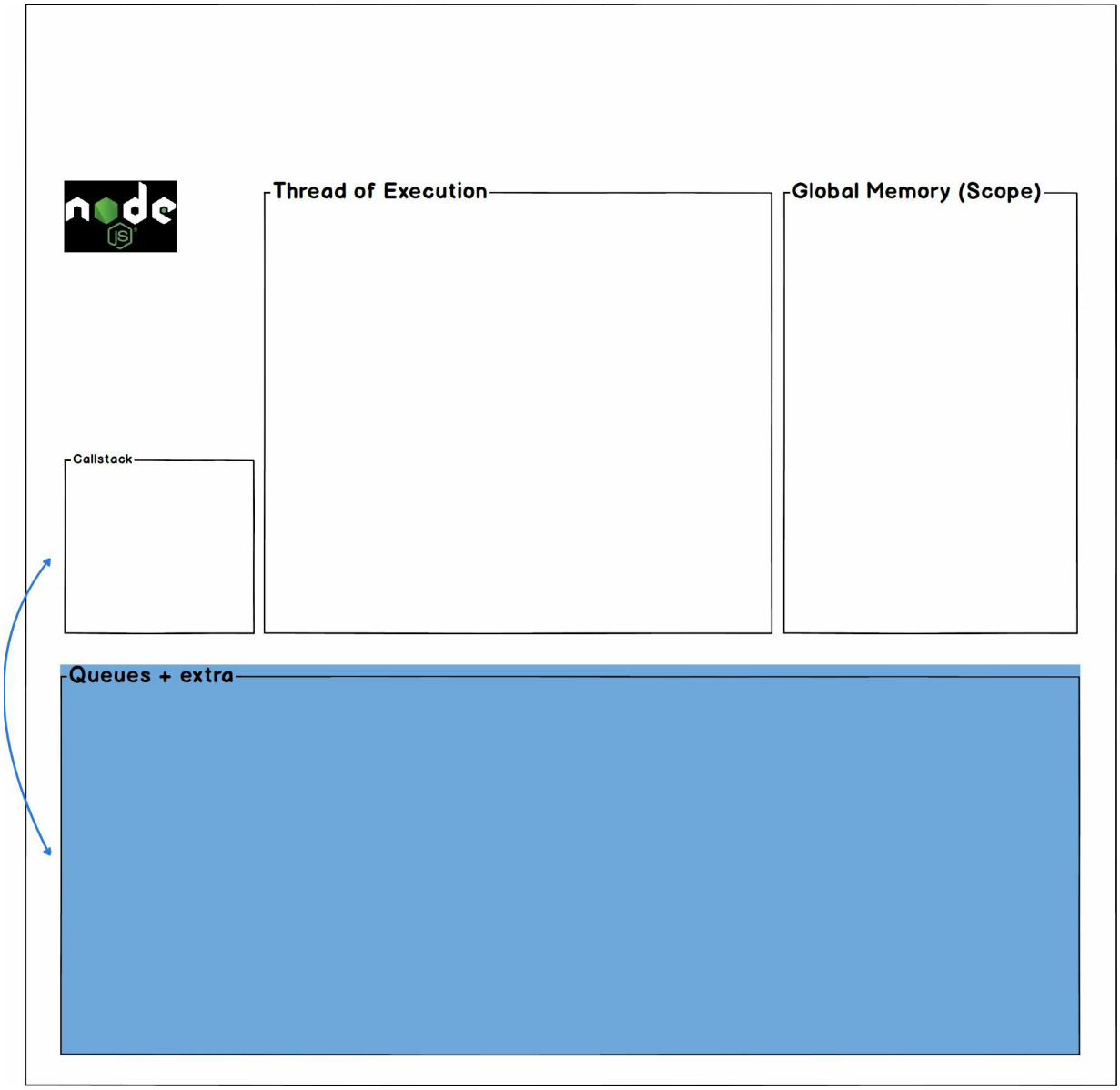
1. It’s not a programming language.
2. It’s not a framework/tool for server-side coding (Not like Php).
3. JavaScript is a programming language, **the node is not**.

What node actually is?

1. It’s a low-level c++ delegated code made available to us by making JavaScript APIs (AKA JavaScipt version of a c++ features(**sometimes even “c” bound c++ code**)).
2. **It’s a literally C++ program** that takes JavaScript code as a string and does all the work!
3. Node is a single c++ program, which relies on a lot of other c++ libraries like **libUV, V8, c-ares, OpenSSL,** etc.
4. We call is nodejs because c++ written features are made available to us by exposing equivalent JavaScript APIs. so, Node(C++ features) + JavaScript == NodeJS😎.

```
NodeJS's core architecture is way more complicated than a browser or any other environment. Not only that, on the internet 90% of information related to nodejs are partially false or misguided😔.
```

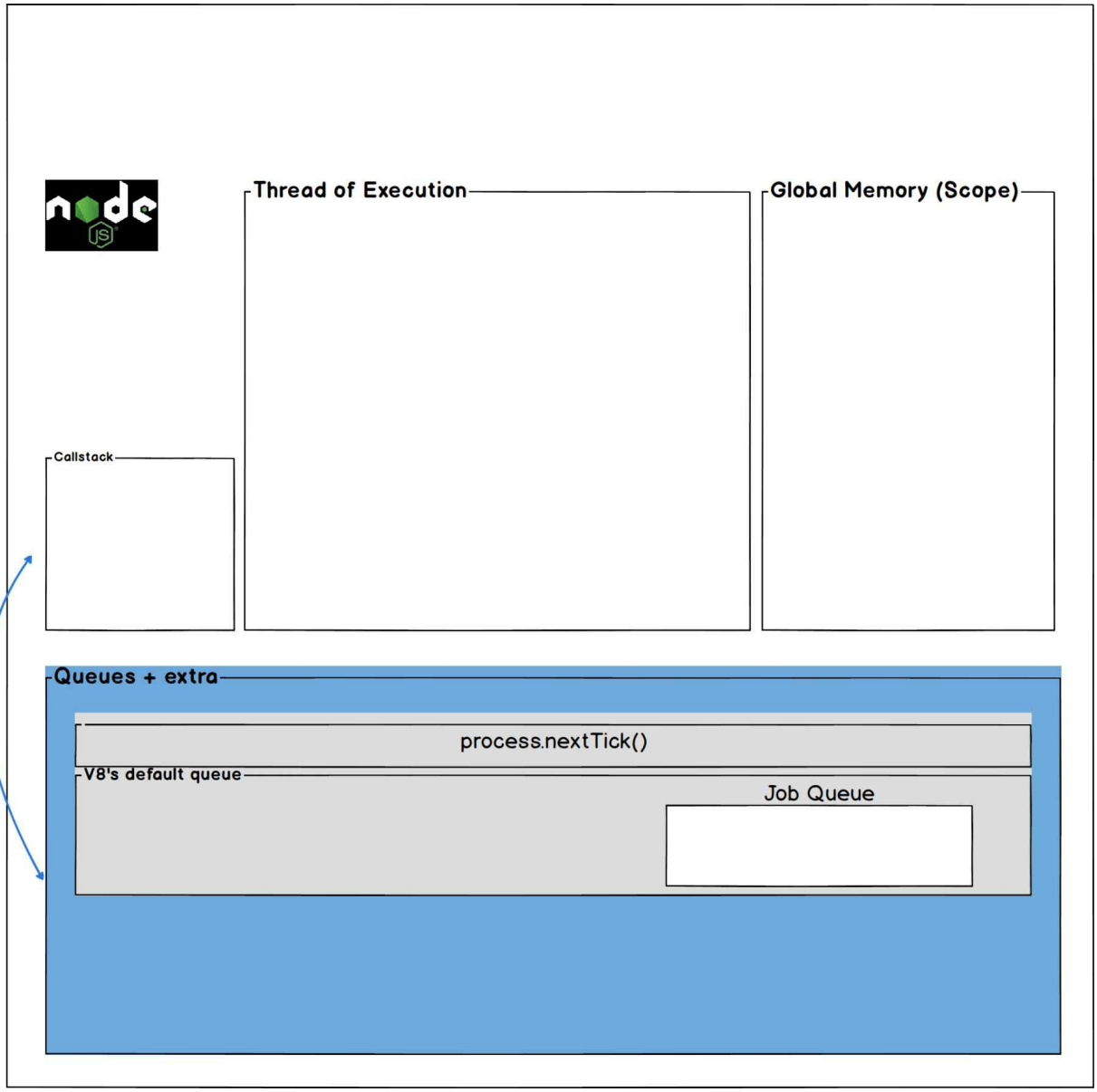
- I will try to go very gradually on the core architecture and will split pieces for you, such that you don’t have to go anywhere to understand anymore. Let’s start with our basic version of the javascript diagram.



- As we can see we have a similar model that we know, queues, callstack, and our memory space.
- Because we are going to perform a lot of system operations on our computer, the node actually uses a lot of third-party c/c++ libraries which does all kinds of stuff.
- There are major 2 components that we should understand first.

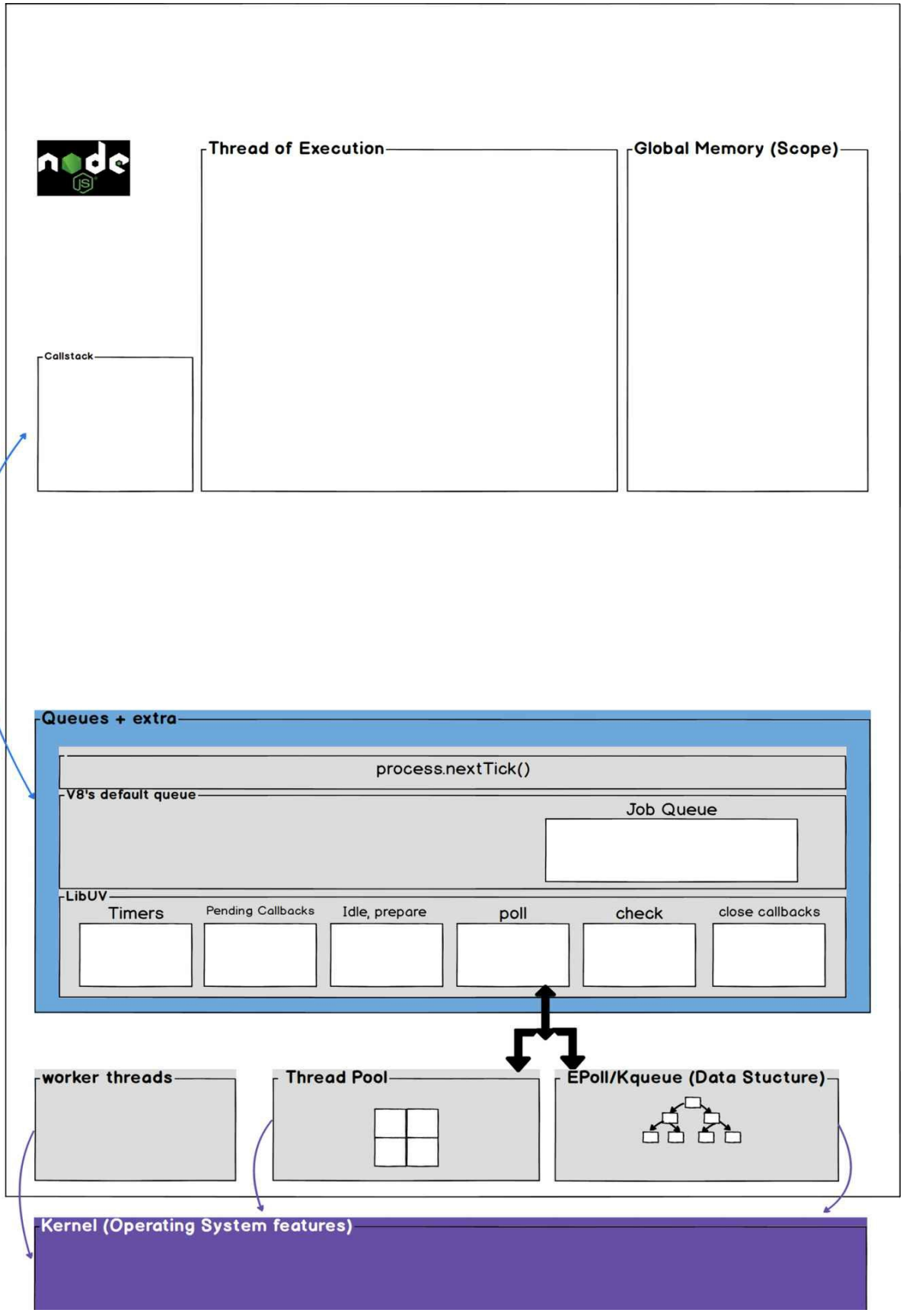
V8 engine

- The V8 engine has a fundamental 1 job to perform which is to parse the JavaScript and convert it into **optimized machine code**.
- However V8 just not convert our javascript code into machine code, but it does much more than that. It provides the default version of the event loop(wait what).
- The default event loop given to us by v8 can be extended or replaced by javascript runtime creators.
- Node actually does not use the default v8's event loop. Node uses the event loop of LibUV's instead of v8!
- Because v8 has a default event loop, the barebone code of v8's event loops queue remains there and **Promise** uses that queue instead of any other queues.
- I have removed the **'Task Queue'** of v8. The main reason because the node doesn't use it or even care about it. Node uses a little different approach to tackle asynchronous **timers**.
- There is **'process.nextTick()'** which is provided by node and it always runs ahead of every single queue by design.


Thread of Execution
Global Memory (Scope)
Callstack
Queues + extra
process.nextTick()
V8's default queue
Job Queue

LibUV

- This library is the heart of the node. LibUV does not just provide an event loop but it has a really good mechanism to handle a lot of I/O operations.
- The I/O operation could be anything like a lot of connection requests, sockets, File systems, etc.
- LibUV provides or sometimes uses a lot of queues which we are going to learn soon.
- Let’s just see first our mental model diagram with **v8 and libUV** combined😊.


Thread of Execution
Global Memory (Scope)
Callstack
Queues + extra
process.nextTick()
V8's default queue
Job Queue
LibUV
Timers
Pending Callbacks
Idle, prepare
poll
check
close callbacks
worker threads
Thread Pool
EPoll/Kqueue (Data Stucture)
Kernel (Operating System features)

- I left out some space above the blue-colored box intentionally and I will come to that.
- As from the diagram you can see we have v8 + LibUV's queues.
- LibUV has major 7 queues(drawn only 6 because Idle and prepare are combined in one line).

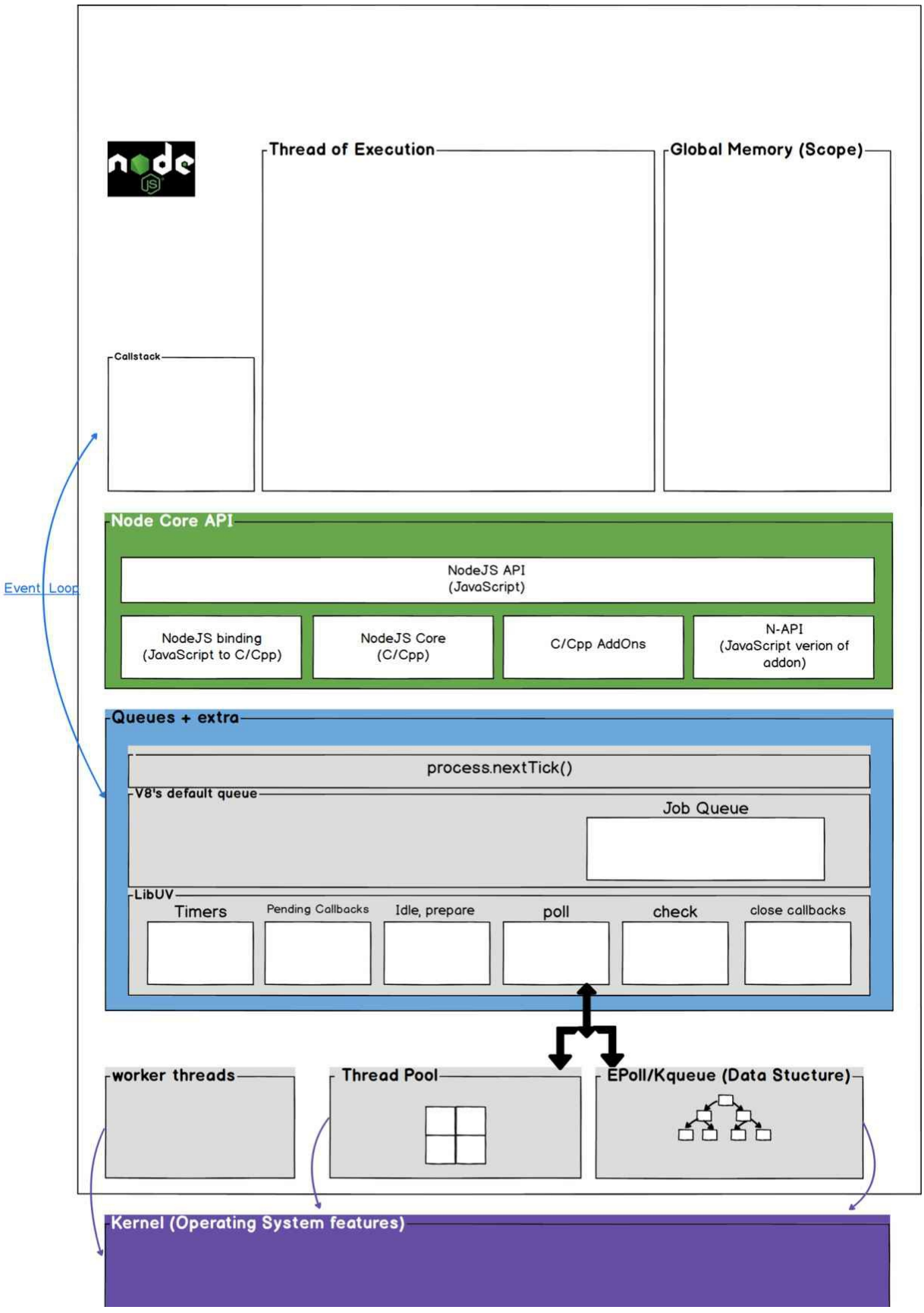
1. **Timers:** setTimeout() and setInterval().
2. **Pending callbacks:** Imagine parent process is **Nginx server which** is already running on PORT 3000, and you want to start node server on PORT 3000. So, it will throw an error(system errors). All those kinds of system call's or error callbacks are given to this queue.
3. **Idle and prepare:** we don't interact with it.😇 (2 queues drawn in one as we don't actually use, libUV uses it internally)
4. Poll: all I/O operations like files handling, client request handling, sockets, crypto, zlib(compression) operations, etc handled via this queue.
5. **Check:** for the setImmediate() callback/s ,we use this queue.
6. **Close:** All close events from a system like a socket close or clean up events are added in this queue (but why ? because these are system calls, not incoming requests, so😁).

The default libUV and v8 combined mental model is not so hard to grasp. The problem is a node internally does way much more stuff than just using these queues and most important part is ***poll queue*****😉.**

- From the diagram, we can see we have some strange stuff like **Thread Pool and Epoll/KQueue(data structure), which is used to do work**

delegated from the poll queue.

- Not only that we have, **Worker Threads** which is also a different thing.
- At the bottom, we have our operating system feature or you can say that heart of the system **‘Kernel’**.
- Before we move further and explain how all things work combined, let’s just finish out node architecture.


Thread of Execution
Global Memory (Scope)
Callstack
Event Loop
Node Core API
NodeJS API
(JavaScript)
NodeJS binding
(JavaScript to C/Cpp)
NodeJS Core
(C/Cpp)
C/Cpp AddOns
N-API
(JavaScript verion of
addon)
Queues + extra
process.nextTick()
V8's default queue
Job Queue
LibUV
Timers
Pending Callbacks
Idle, prepare
poll
check
close callbacks
worker threads
Thread Pool
EPoll/Kqueue (Data Stucture)
Kernel (Operating System features)

- As we know we are going to write ‘JavaScript’, so any system-level code needs to be written in JavaScript which will be delegated to our c++ code. That’s the reason why we have **Node Core APIs.**
- It has major all core modules logic written in c++ and exposed javascript API which we use as a developer.
- **C++ Addon and N-API** are special features or APIs available to us that we can use to extends nodejs using c++ (this requires c/c++ knowledge so I am not going to tell you about this😊).

Normal Flow of execution inside the node.

step 1: Scan the entire code and hoist all possible variables. Do garbage collection of unused references(optimization).

step 2: Complete all synchronous execution from top to bottom.

step 3: All the asynchronous code's event handlers or callbacks are scanned and stored in memory as a future placeholder.

step 4: Start executing all finished background code via a queue, and keep going until we don't finish all background tasks or kill the process (Event Loop😁!)

The order of Queue in the node environment is as shown in the figure.

1. execute all process.nextTick()'s callback/s.
2. execute all promises of the **job queue.**
3. enter into LibUV's queues and start from right to left for execution
   3.1 complete the timers via the **Timer queue**.
   3.2 complete error handling callback of system calls
   via **Pending queue**.
   3.3 complete internally used **Idle, prepare queue.** 3.4 wait for some amount of period to try to finish as much as callbacks of the **poll queue.** (little complicated, we will understand this deeply later). 3.5 complete the **check queue** (setImmediate). 3.6 complete the execution of close callbacks via the **close queue.**

step 5: repeat the loop until the node process gets killed.

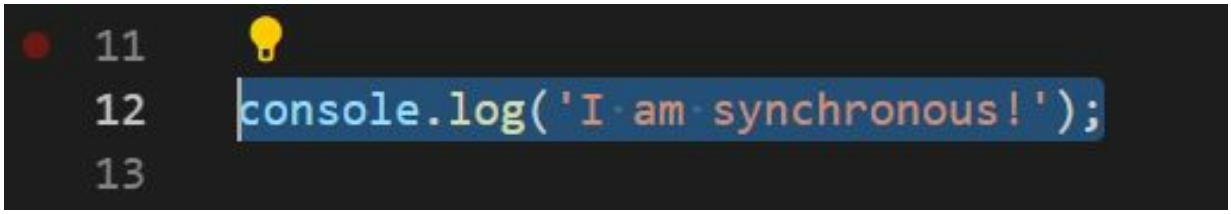
```
It's time to understand everything combined with an example and will understand
deeply what happens in each step.
```

```
const fs = require('fs');
const http = require('http');

fs.readFile('./file1.txt', 'utf8', (err, data) => {
    console.log(data); // 'Hello guys 1'
});
fs.readFile('./file2.txt', 'utf8', (err, data) => {
    console.log(data); // 'Hello guys 2'
});

console.log('I am synchronous!');
process.nextTick(() => {
    console.log('nextTick');
});

setImmediate(() => {
    console.log('immediate');
});

http.get('http://dummy.restapiexample.com/api/v1/employee/1', (res) => {
    console.log(res.statusMessage, '1');
});
http.get('http://dummy.restapiexample.com/api/v1/employee/2', (res) => {
    console.log(res.statusMessage, '2');
});

Promise.resolve('Success Promise').then(function(value) {
    console.log(value); // "Success1"
})

setTimeout(() => {
    console.log('timeout');
}, 0);

```

Ignore how the ***require() and how the module system works right now and only focus on our example.*** *Also, I wrote a javascript code*

in a way that execution of code will have completely different unexpected output than logically shown in code.

- In our starting, we are reading the content of text files that is file1.txt and file2.txt.
- file1.txt has ‘Hello guys 1’ and file2.txt has ‘Hello guys 2’ in it as content.



- The next statement is the pretty simple synchronous log.

- The next thing, we are using process.nextTick() as shown above.

```
setImmediate(() => {
    console.log('immediate');
});
```

- After that, we are calling setImmediate() as above.

```
http.get('http://dummy.restapiexample.com/api/v1/employee/1', (res) => {
    console.log(res.statusMessage, '1');
});
http.get('http://dummy.restapiexample.com/api/v1/employee/2', (res) => {
    console.log(res.statusMessage, '2');
});
```

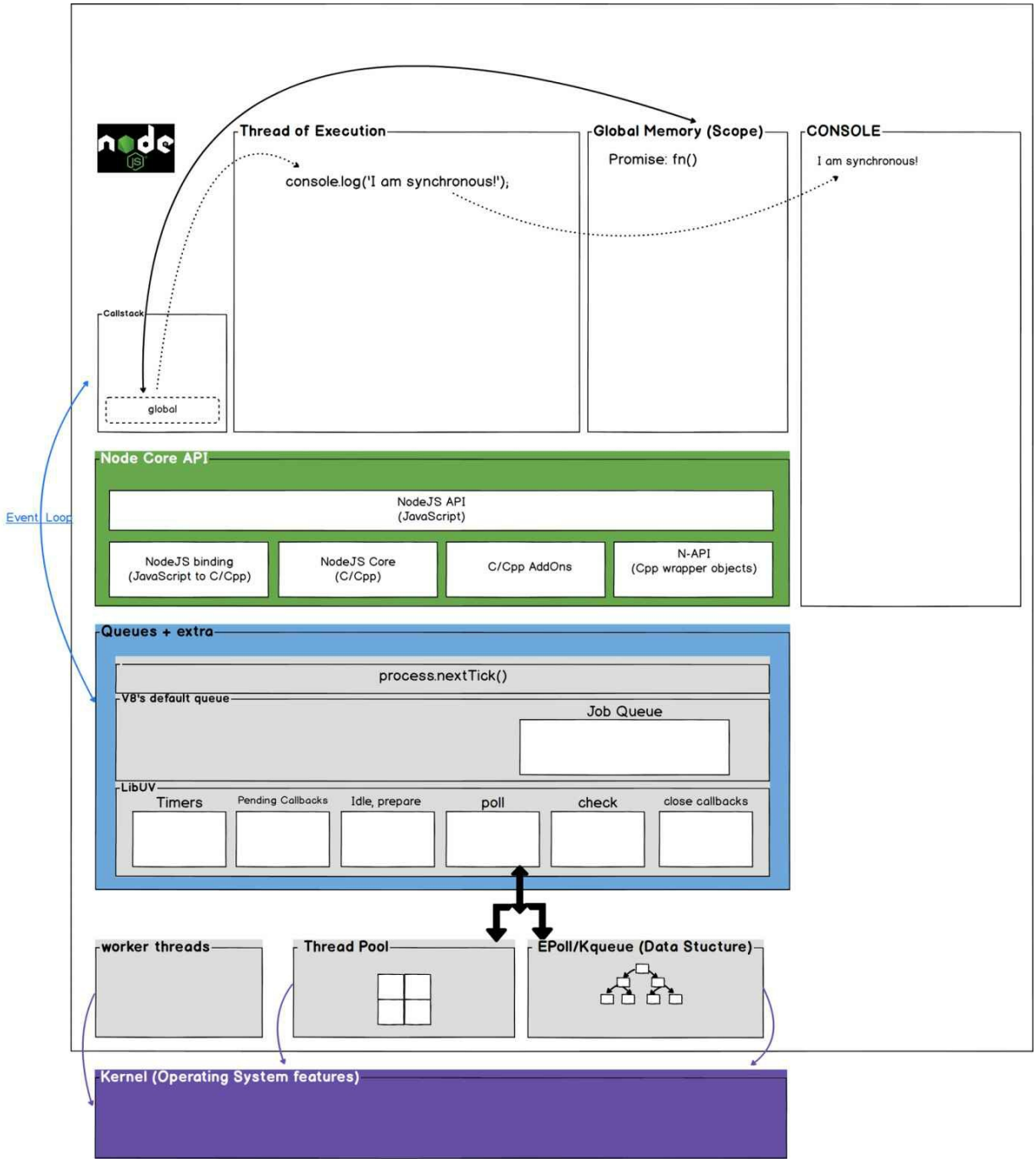
- Then after, we are making **2 HTTP GET API** calls and logging simply **statusMessage** from the response.

```
Promise.resolve('Success Promise').then(function(val) {
    console.log(val); // "Success Promise"
})
```

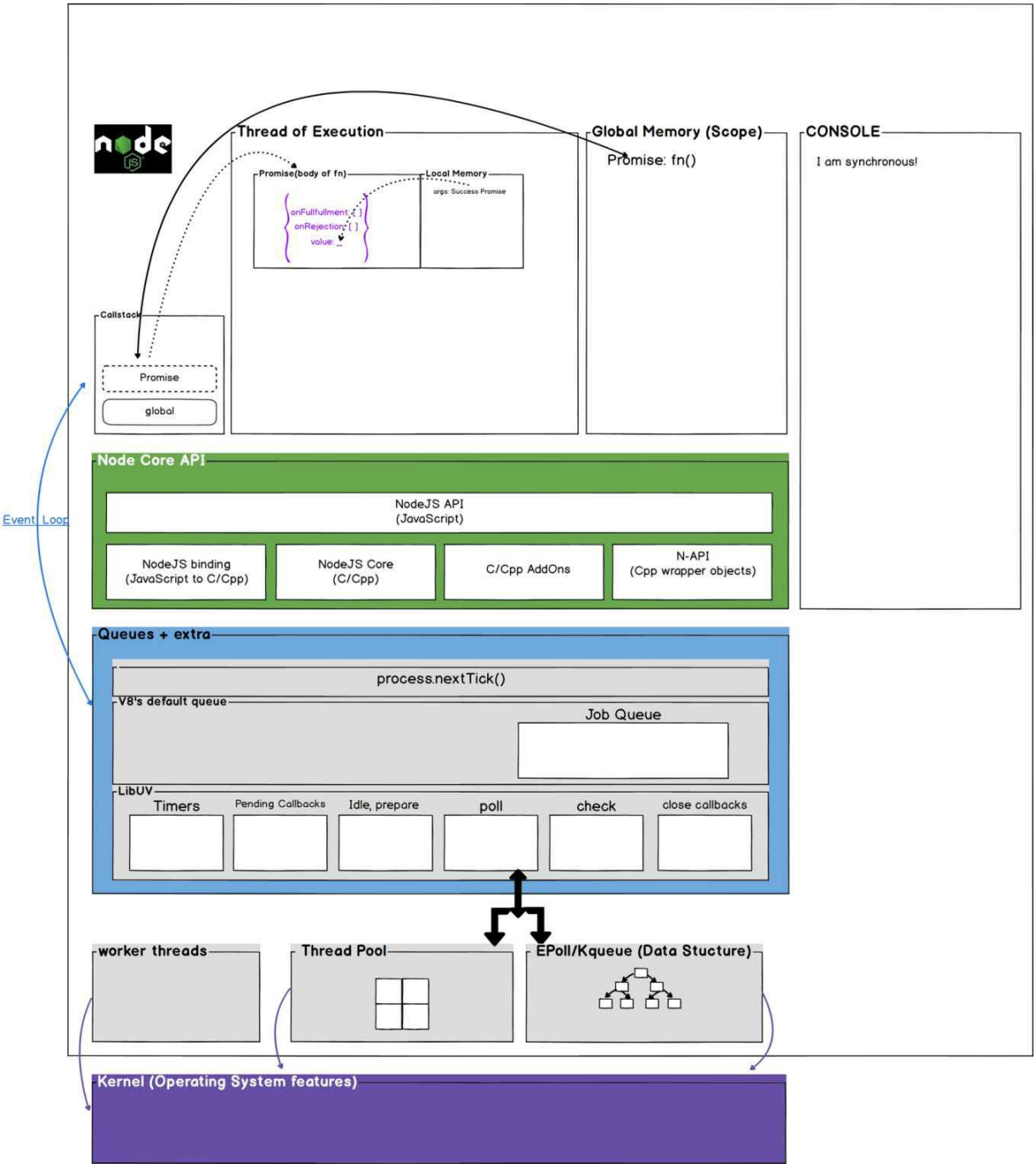
- After that, we are just resolving Promise and have **then()** call of that resolved promise.

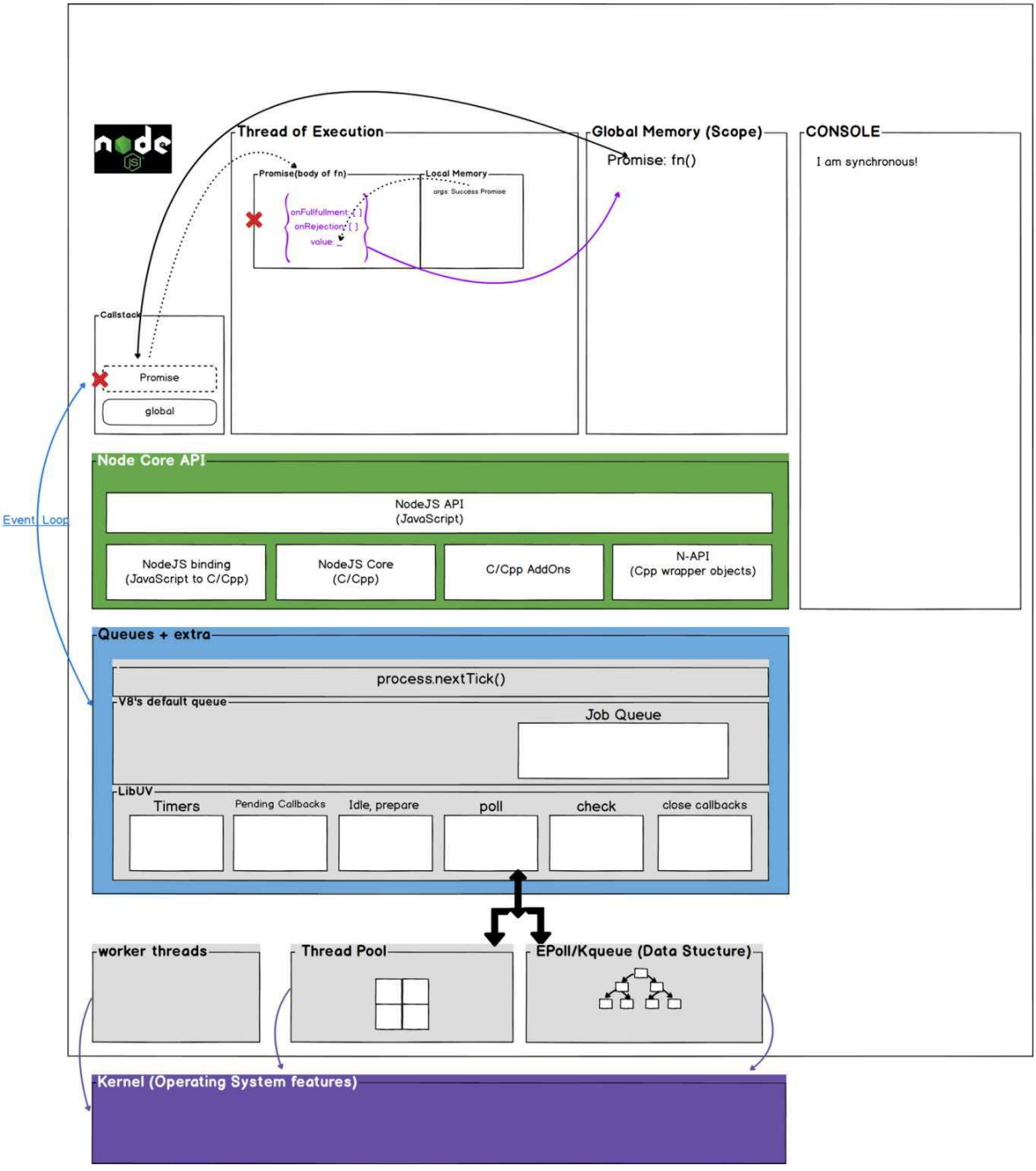
- At last in our code, we have setTimeout() call which has **0** milliseconds timeout.

```
It's time to start our visualization of code and see how code actually works under the hood 😎!
```

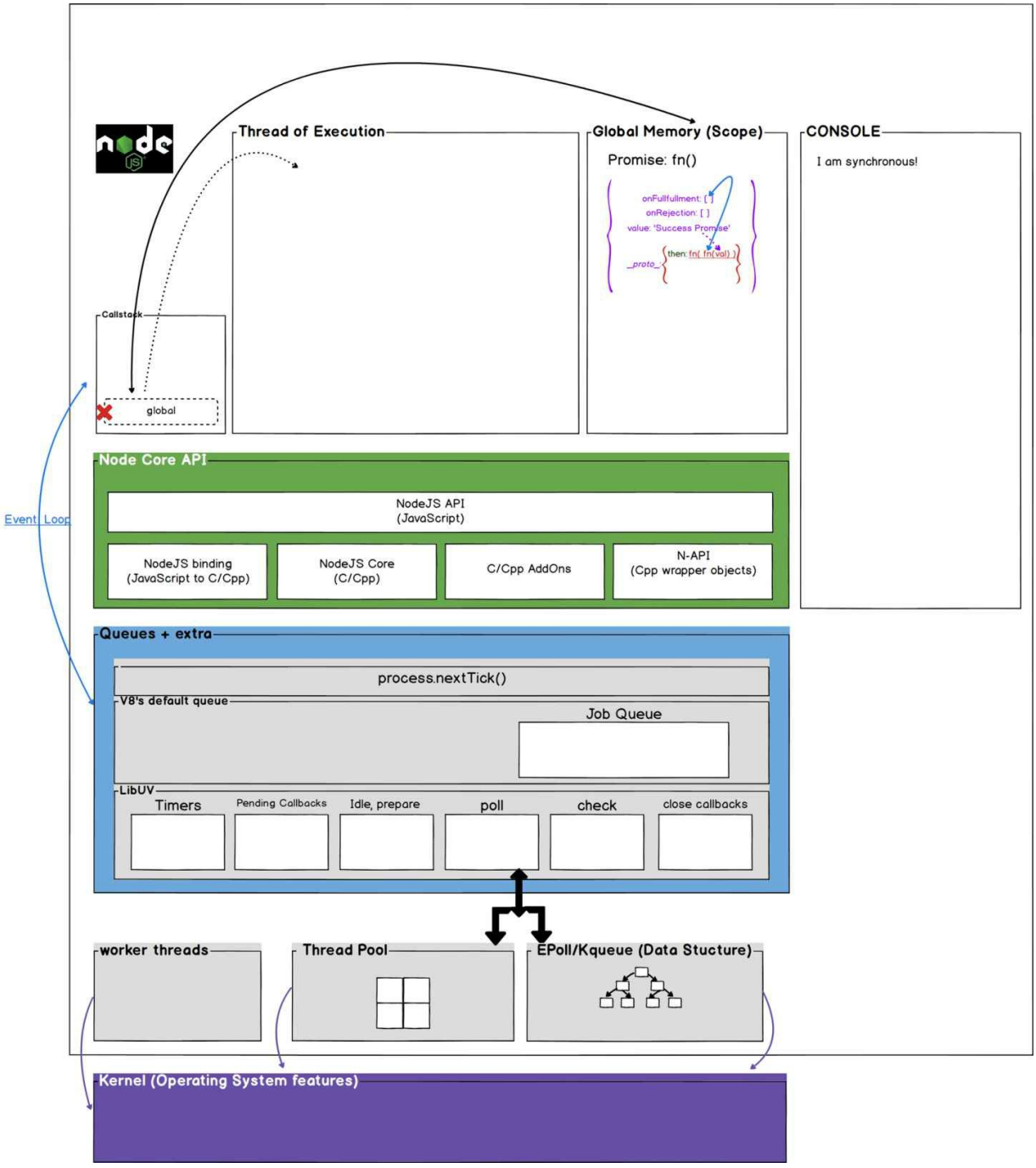


- Initially, we will complete our all synchronous code which in our case, console statement and Promise function constructor (note here, not the finished promise object, only **Promise** constructor execution).

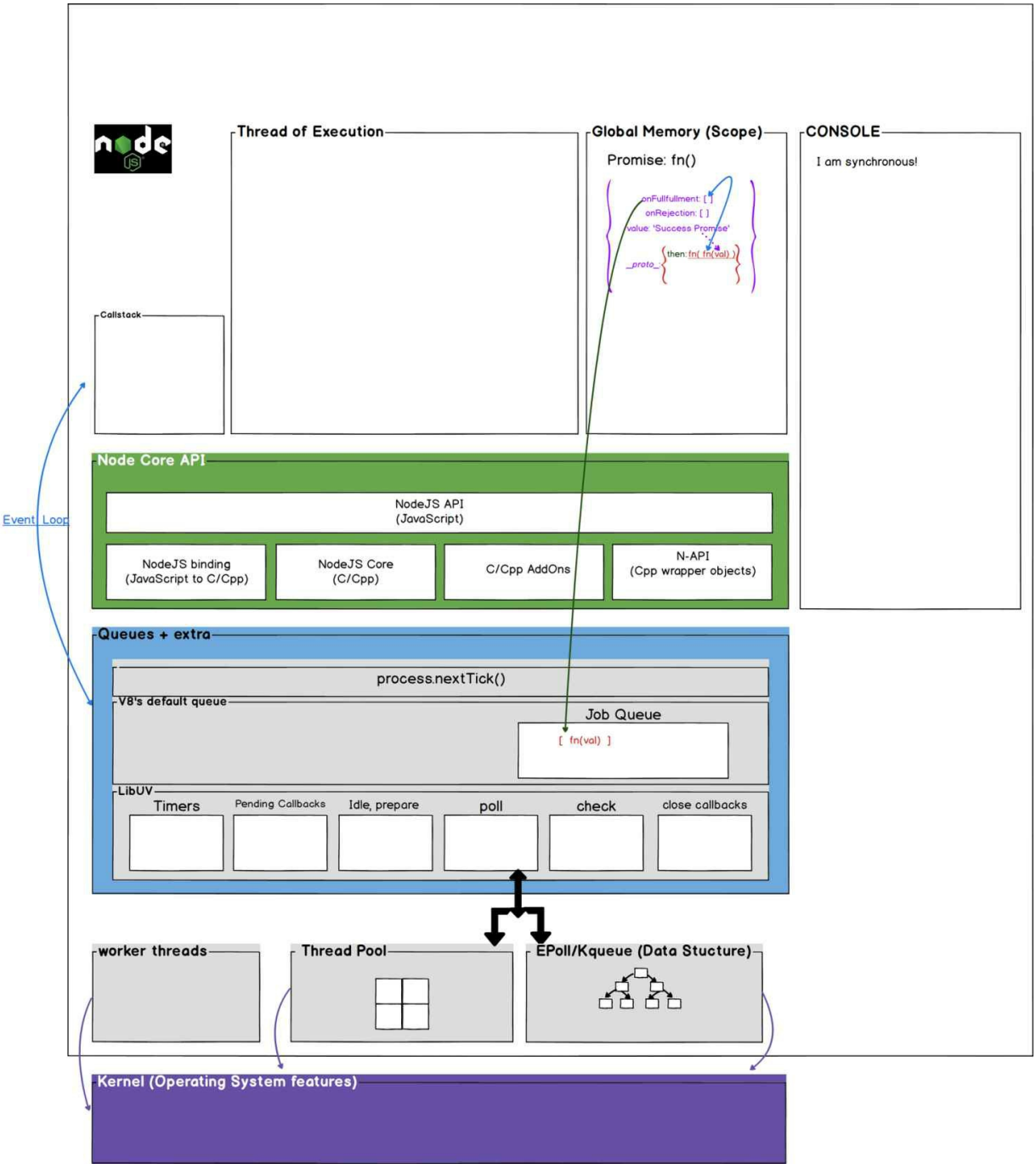

Thread of Execution
Promise(body of fn)
Local Memory
args: Success Promise
onFullfullment: [ ]
onRejection: [ ]
value:
Global Memory (Scope)
Promise: fn()
CONSOLE
I am synchronous!
Callstack
Promise
global
Event Loop
Node Core API
NodeJS API
(JavaScript)
NodeJS binding
(JavaScript to C/Cpp)
NodeJS Core
(C/Cpp)
C/Cpp AddOns
N-API
(Cpp wrapper objects)
Queues + extra
process.nextTick()
V8's default queue
Job Queue
LibUV
Timers
Pending Callbacks
Idle, prepare
poll
check
close callbacks
worker threads
Thread Pool
EPoll/Kqueue (Data Stucture)
Kernel (Operating System features)

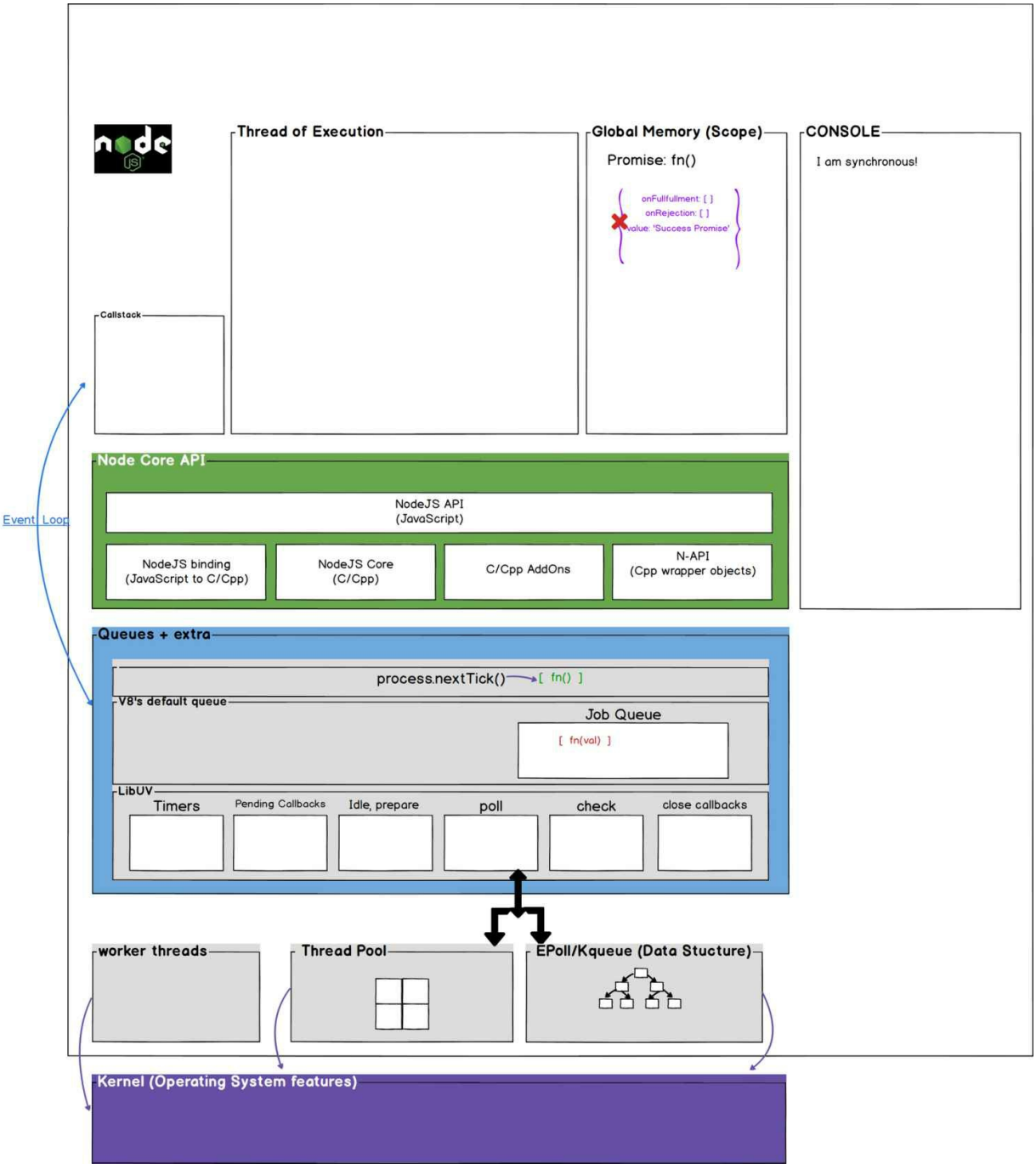


- Here we are returning a promise object but we are not storing it in any variable, so immediately **then()'s callback** is put into **Job Queue** and the object gets garbage collected as it has zero references in memory. let's see it below**!**



- **Working of Promises does not changes in browser or node environment as both uses V8 and promises are part of V8 engine itself.**


Thread of Execution
Global Memory (Scope)
Promise: fn()
onFullfullment: [ ]
onRejection: [ ]
value: 'Success Promise'
then: fn(fn(val))
__proto__
CONSOLE
I am synchronous!
Callstack
Node Core API
NodeJS API
(JavaScript)
NodeJS binding
(JavaScript to C/Cpp)
NodeJS Core
(C/Cpp)
C/Cpp AddOns
N-API
(Cpp wrapper objects)
Event Loop
Queues + extra
process.nextTick()
V8's default queue
Job Queue
[fn(val)]
LibUV
Timers
Pending Callbacks
Idle, prepare
poll
check
close callbacks
worker threads
Thread Pool
EPoll/Kqueue (Data Stucture)
Kernel (Operating System features)

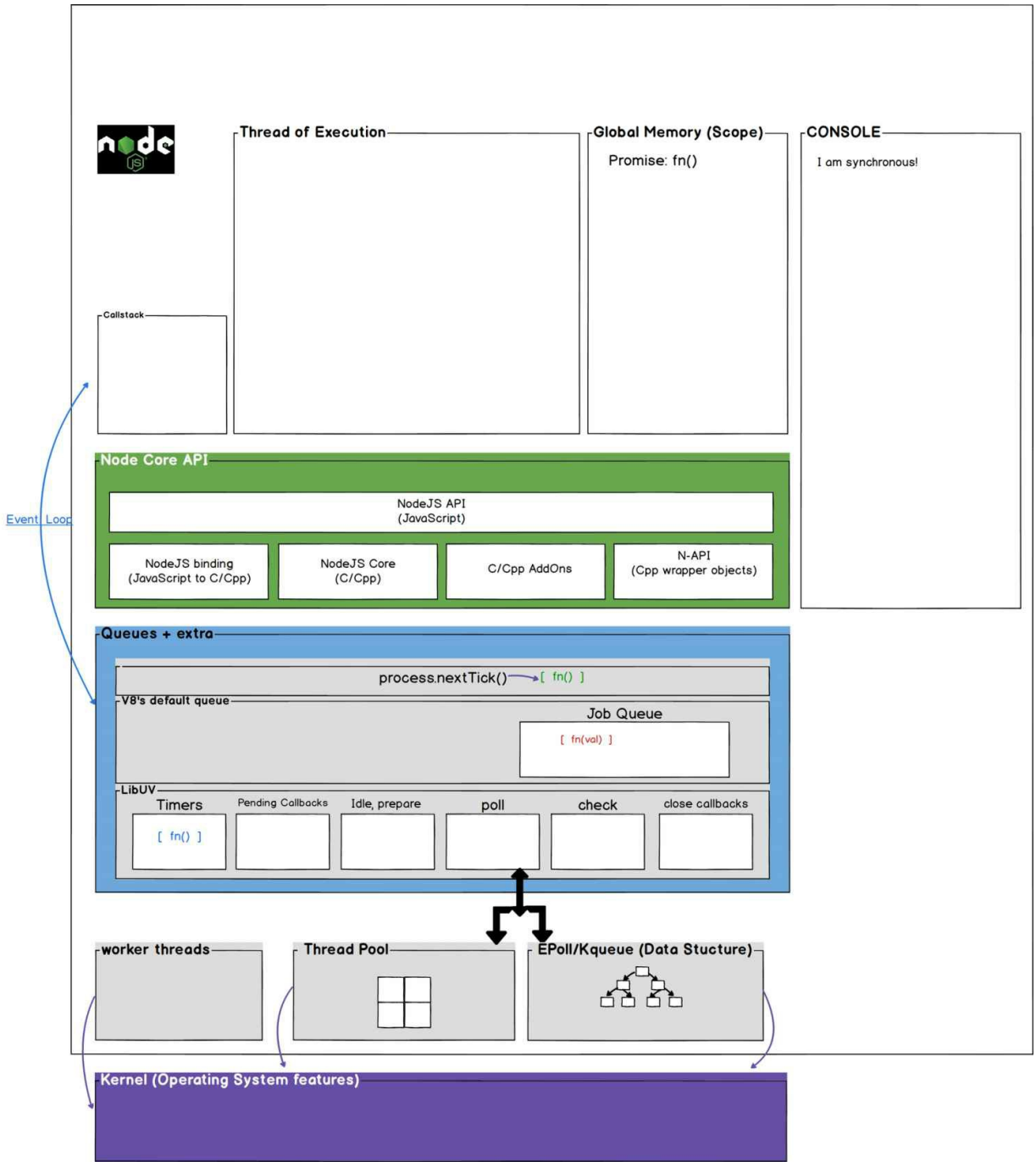


- As you can see, the callback of **then()** is added to the **job queue** and the promise object gets garbage collected. The next thing we did is continued our execution.

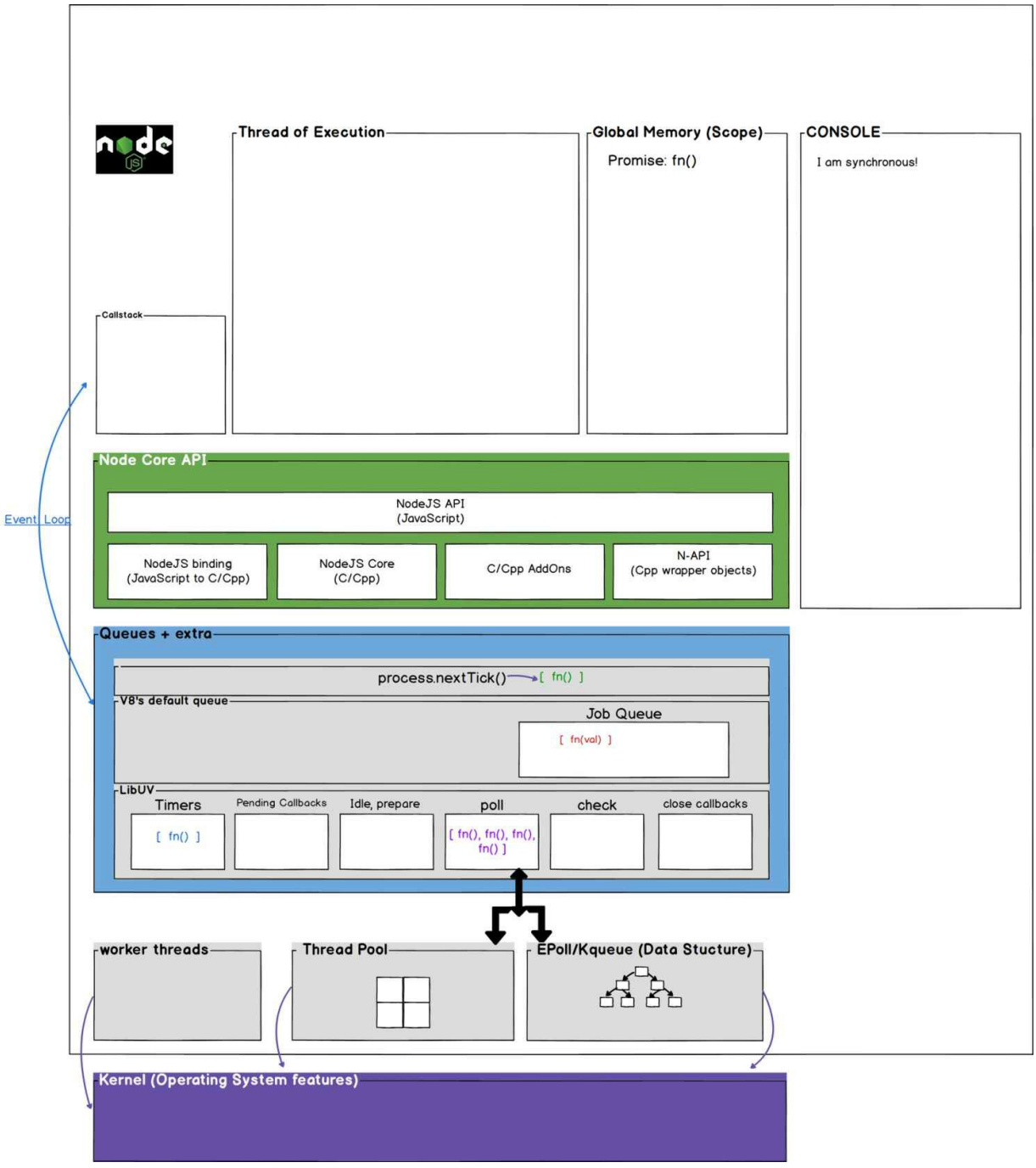
  As node made available **process.nextTick()**, **<u>all callback of</u>**

process.nextTick() after finishing synchronous execution, we add in one separate queue as shown in the figure.

- so callback of process.nextTick() is stored in the queue which is on our topmost hierarchy of queues.



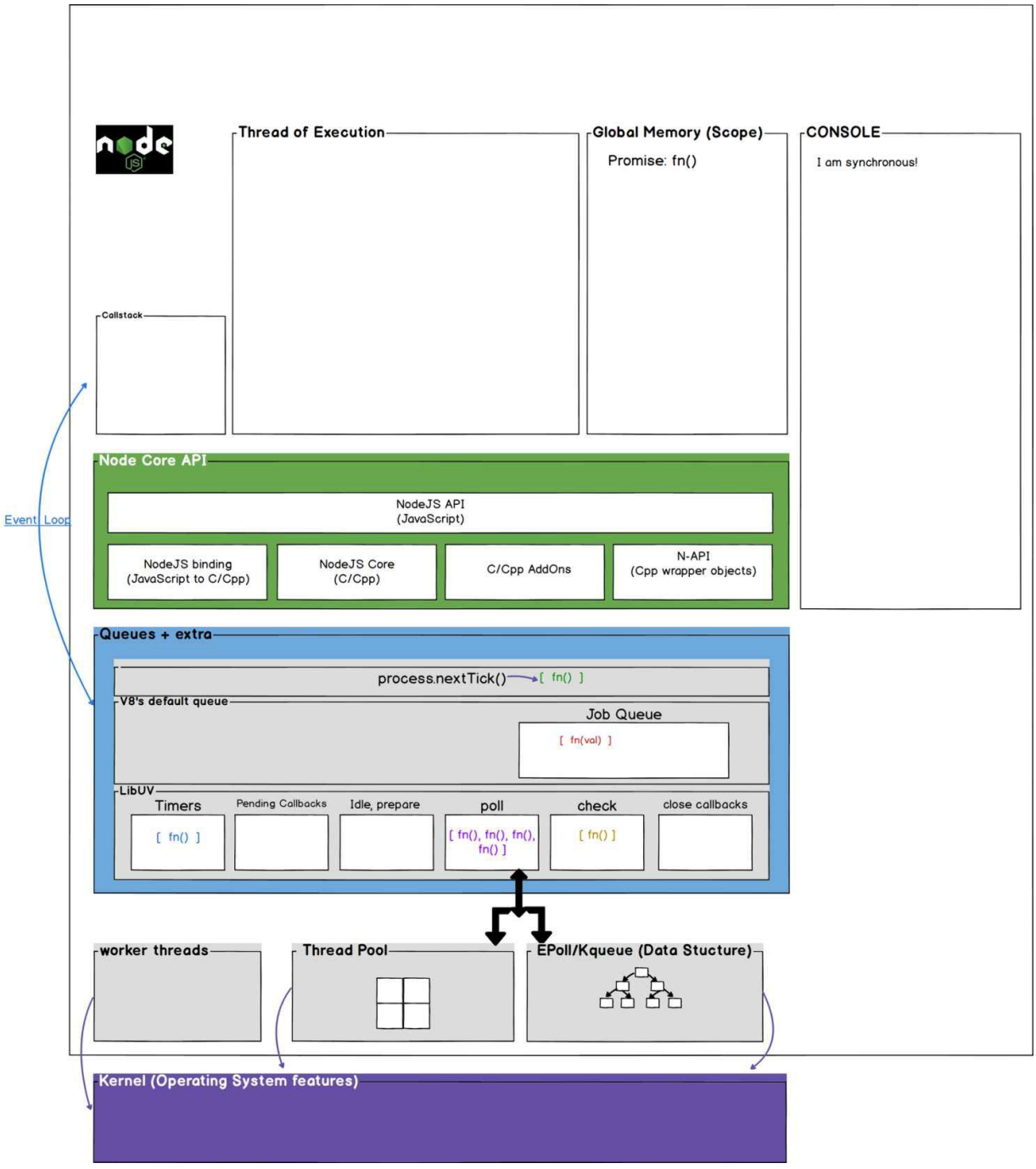
- We further go through code and store callbacks into their respective queue which node environment decided. in the above, we added a timer callback to the timer queue.



- As we can see above, all the I/O callbacks are in the poll queue. Both

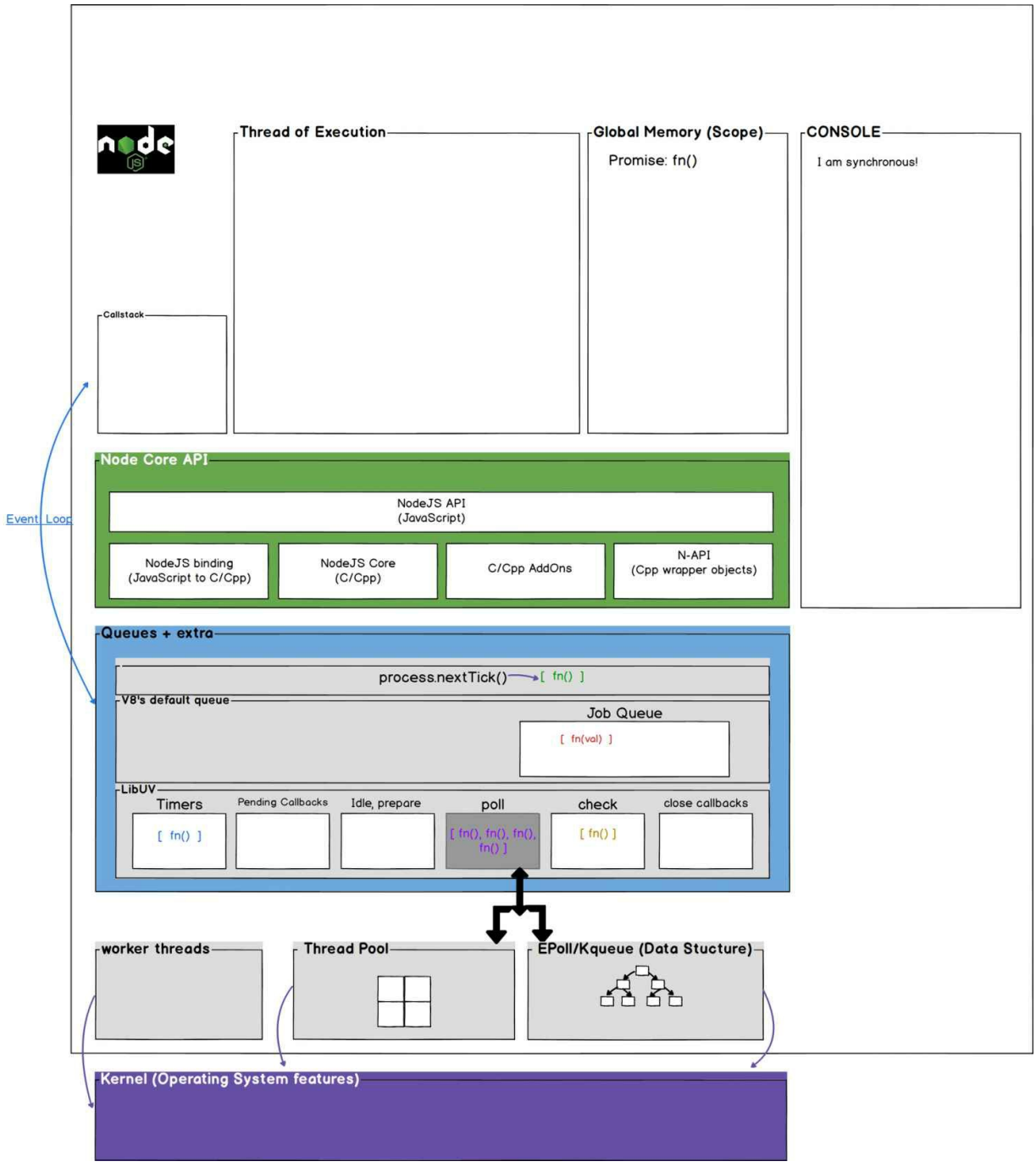
files read operation and HTTP get API call are part of I/O operation.

- Internally poll queue used different approaches based on what operation is performed on the poll queue(will see soon).



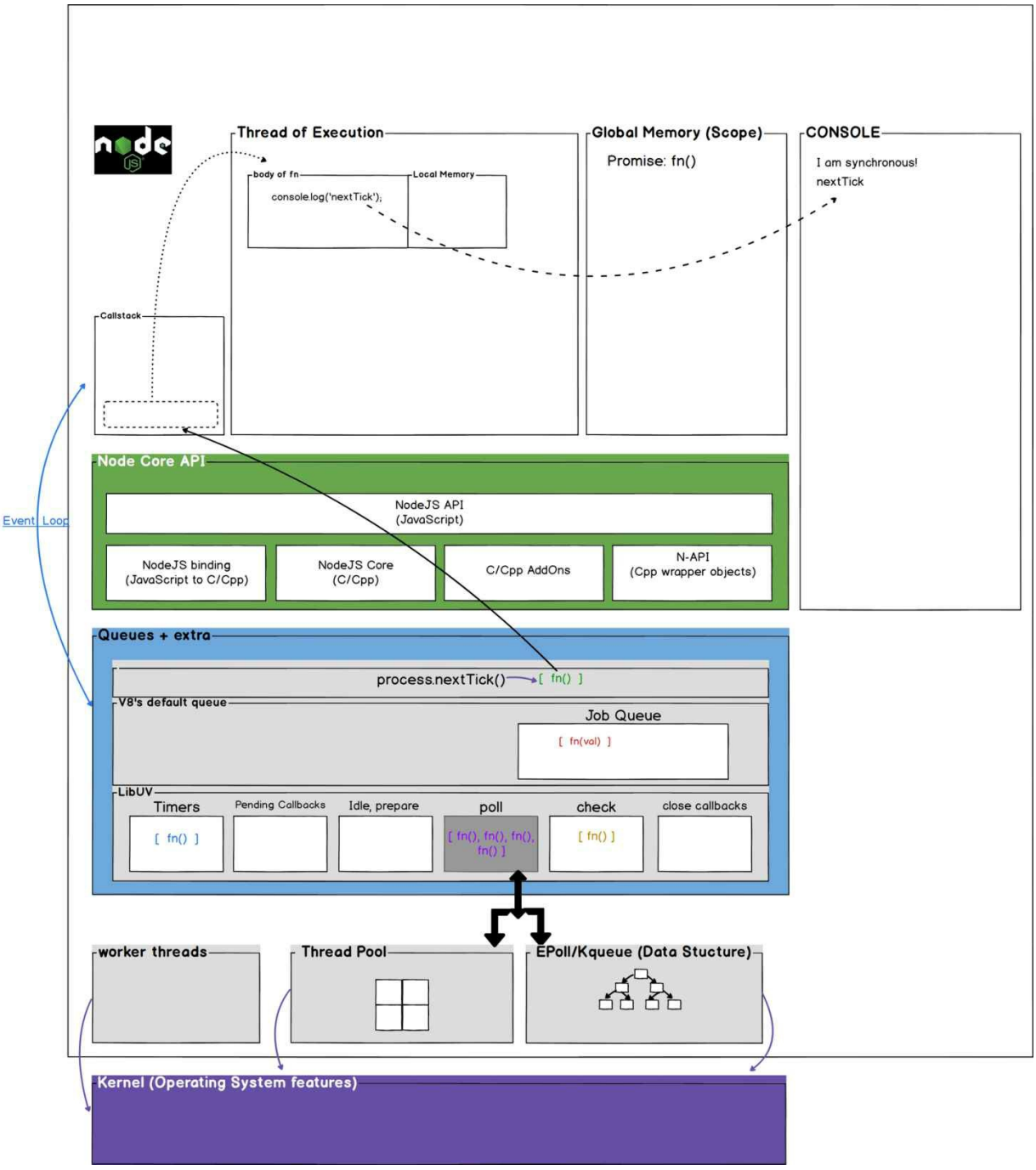
- At last, we have setImmediate() which will go into the **check queue.**

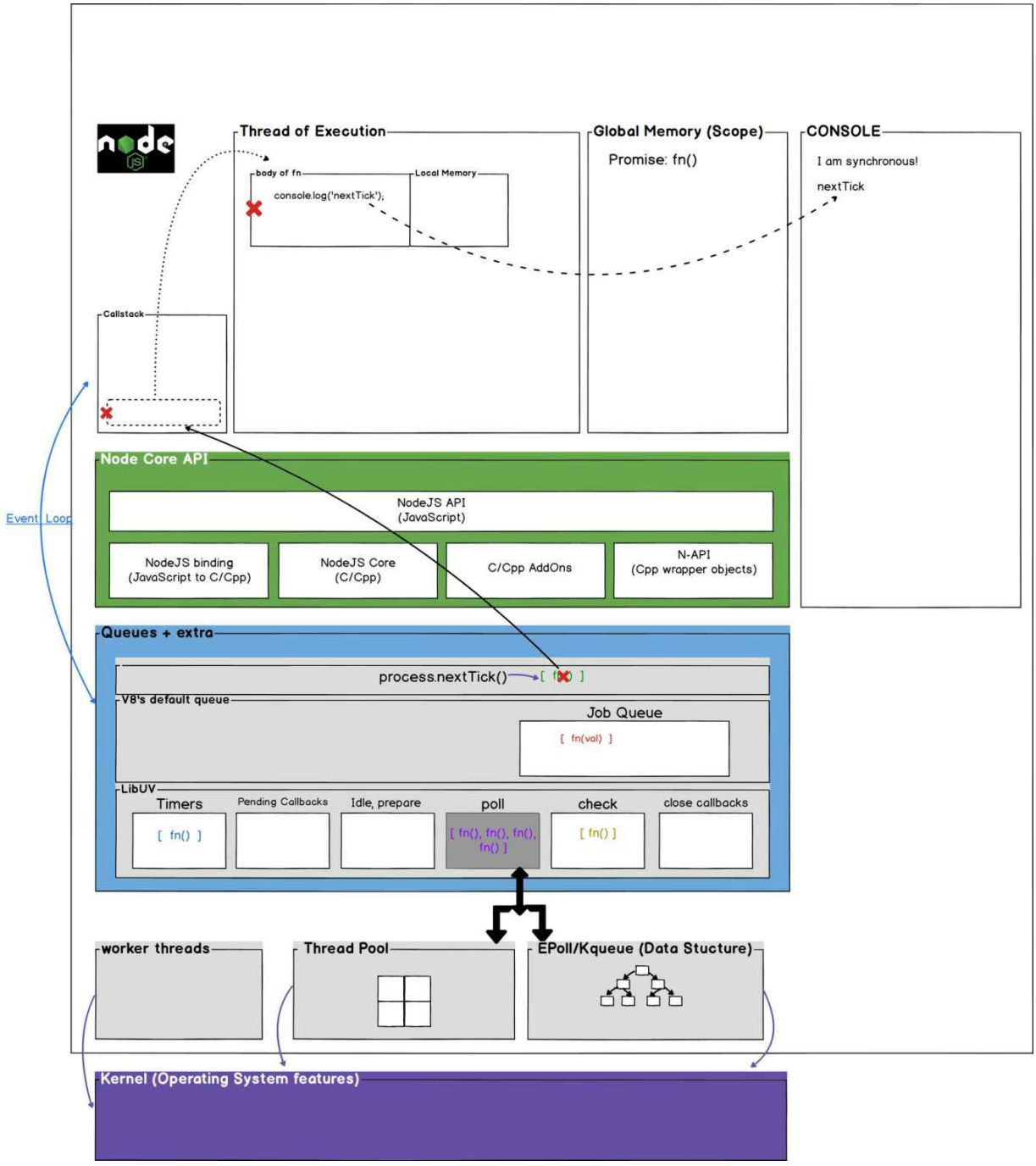
- The poll queue actually starts background work and does not take participation in the first iteration(the first iteration of the event loop) of the queues call. So, I have made a grey color for that.



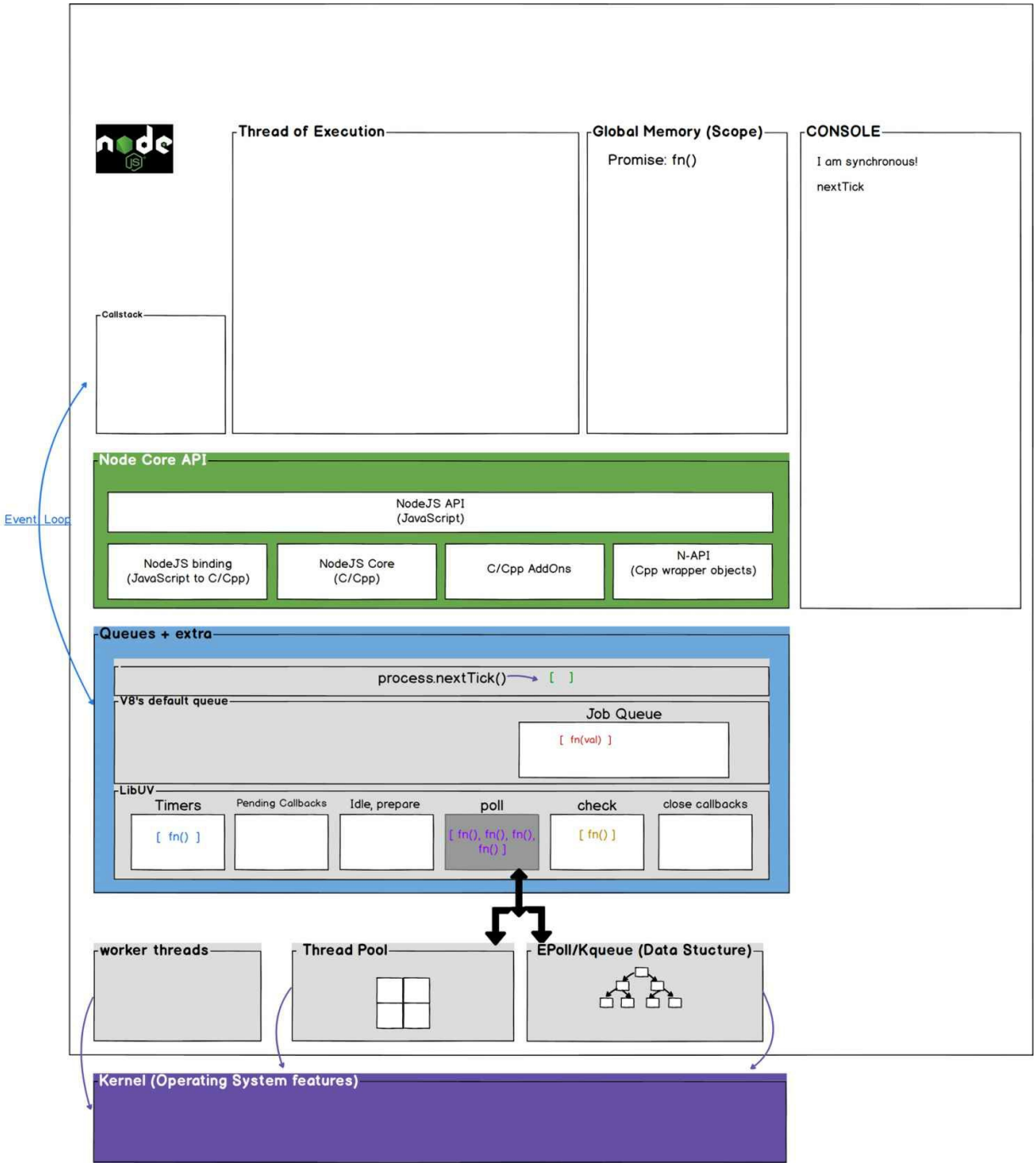
- Now, let's start executing our queues. Remember the order for queues

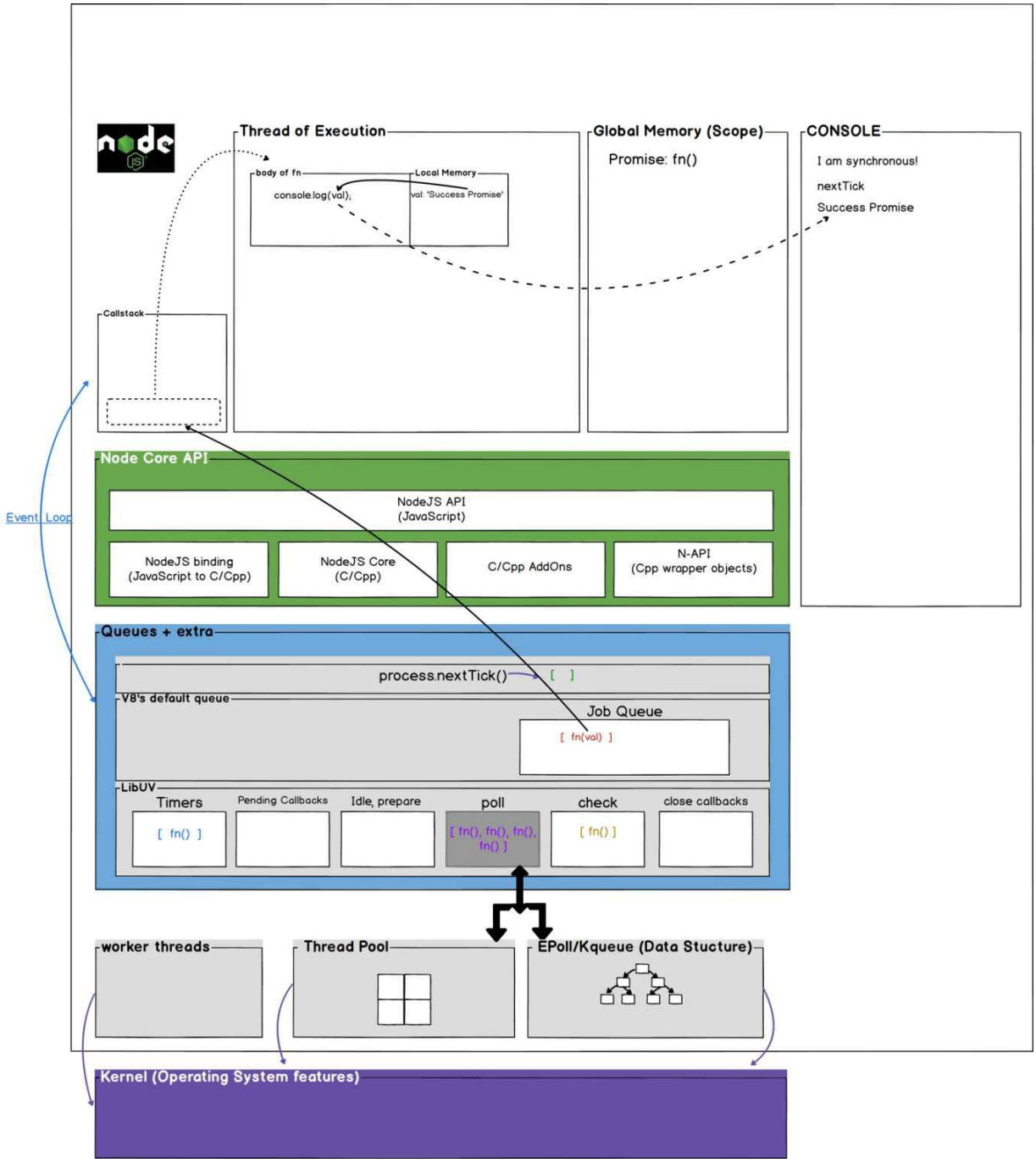
which is top to bottom and right to left.

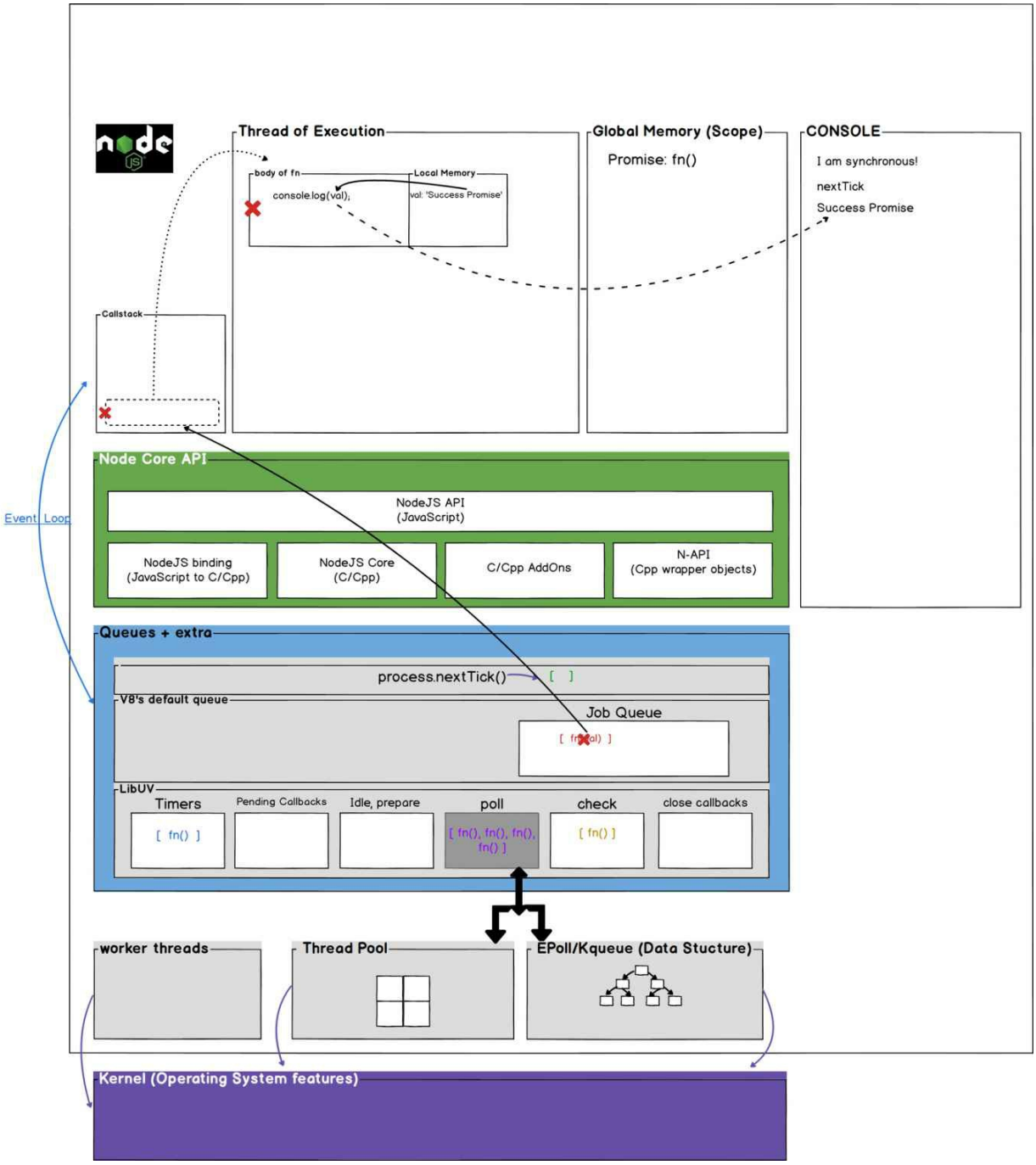


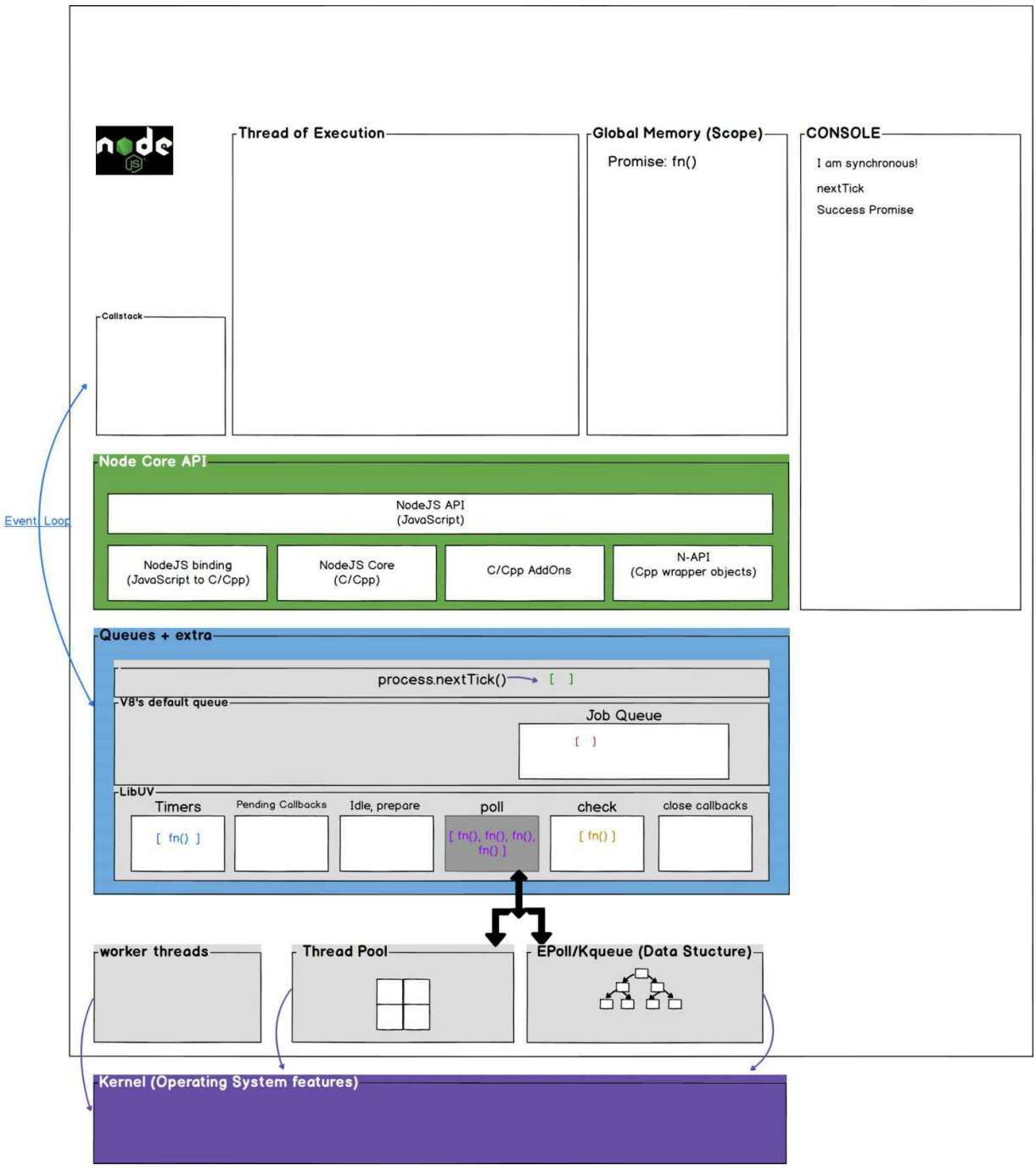


The next turn is for our job queue (promises).

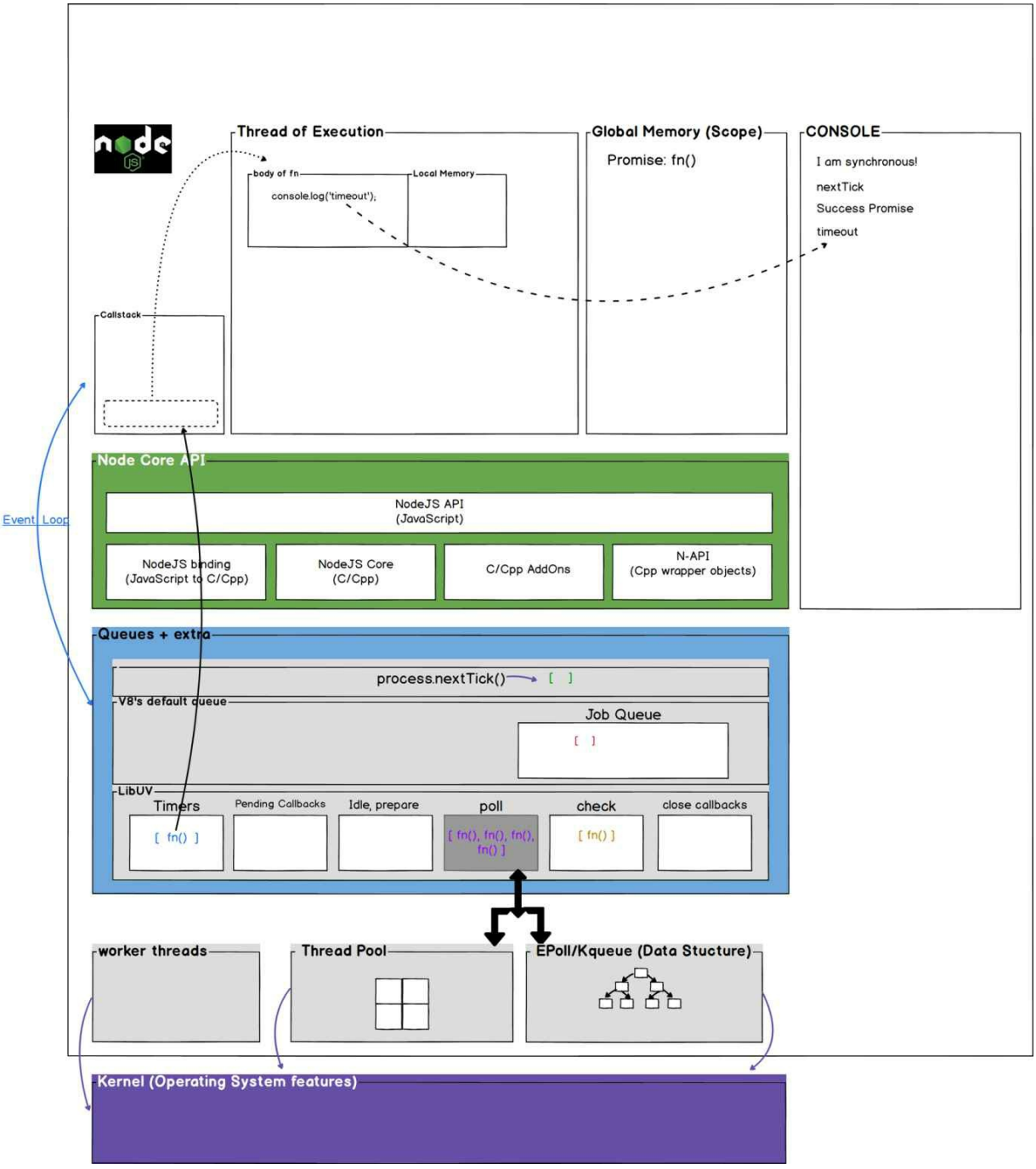

Thread of Execution
Global Memory (Scope)
Promise: fn()
CONSOLE
I am synchronous!
nextTick
Callstack
Event Loop
Node Core API
NodeJS API
(JavaScript)
NodeJS binding
(JavaScript to C/Cpp)
NodeJS Core
(C/Cpp)
C/Cpp AddOns
N-API
(Cpp wrapper objects)
Queues + extra
process.nextTick()
[ ]
V8's default queue
Job Queue
[fn(val)]
LibUV
Timers
Pending Callbacks
Idle, prepare
poll
check
close callbacks
[fn()]
[fn(), fn(), fn(), fn()]
[fn()]
worker threads
Thread Pool
EPoll/Kqueue (Data Stucture)
Kernel (Operating System features)

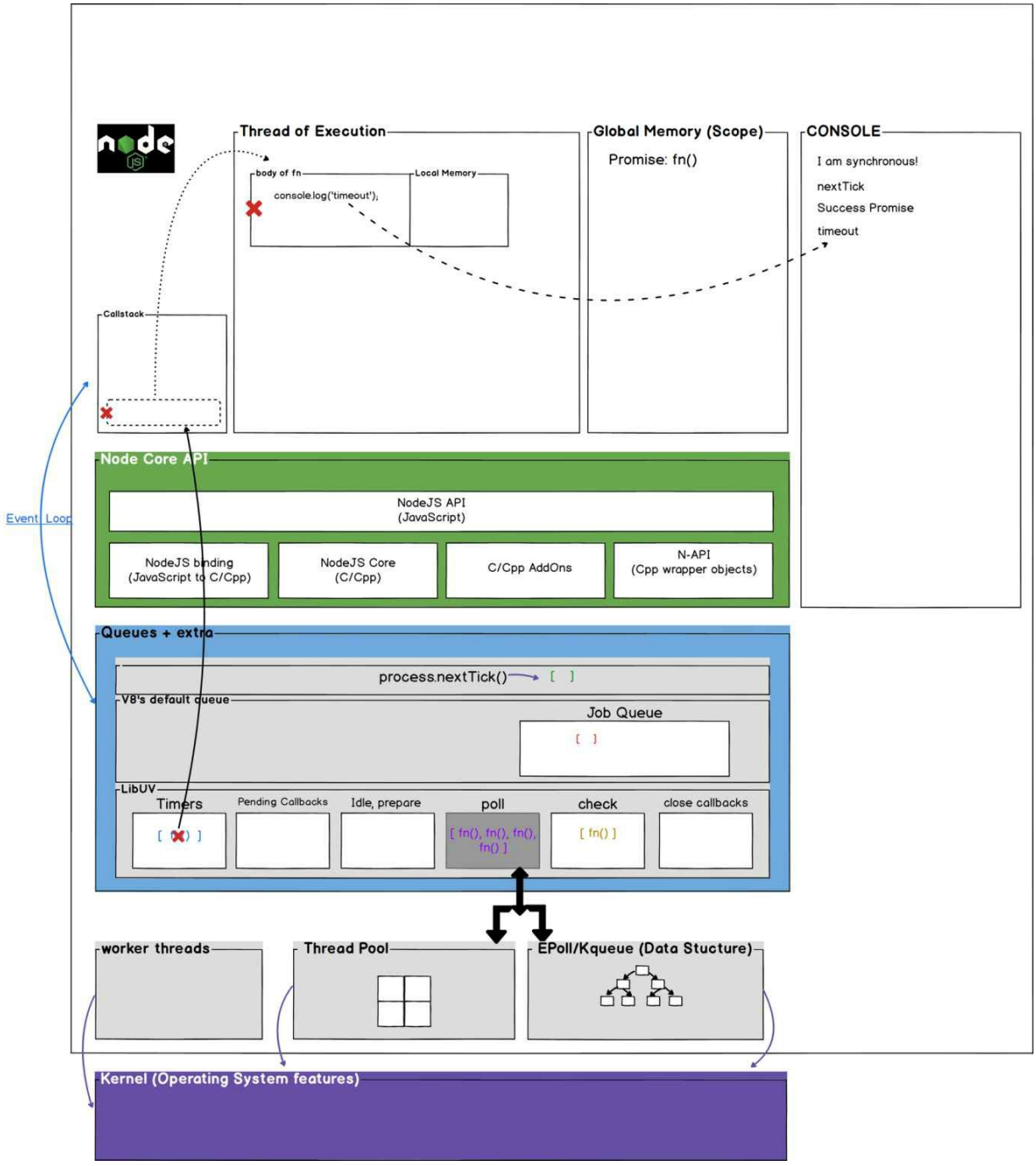

Thread of Execution
body of fn
Local Memory
console.log(val);
val: 'Success Promise'
Global Memory (Scope)
Promise: fn()
CONSOLE
I am synchronous!
nextTick
Success Promise
Callstack
Event Loop
Node Core API
NodeJS API
(JavaScript)
NodeJS binding
(JavaScript to C/Cpp)
NodeJS Core
(C/Cpp)
C/Cpp AddOns
N-API
(Cpp wrapper objects)
Queues + extra
process.nextTick()
[ ]
V8's default queue
Job Queue
[fn(val)]
LibUV
Timers
Pending Callbacks
Idle, prepare
poll
check
close callbacks
[fn()]
[fn(), fn(), fn(), fn()]
[fn()]
worker threads
Thread Pool
EPoll/Kqueue (Data Stucture)
Kernel (Operating System features)

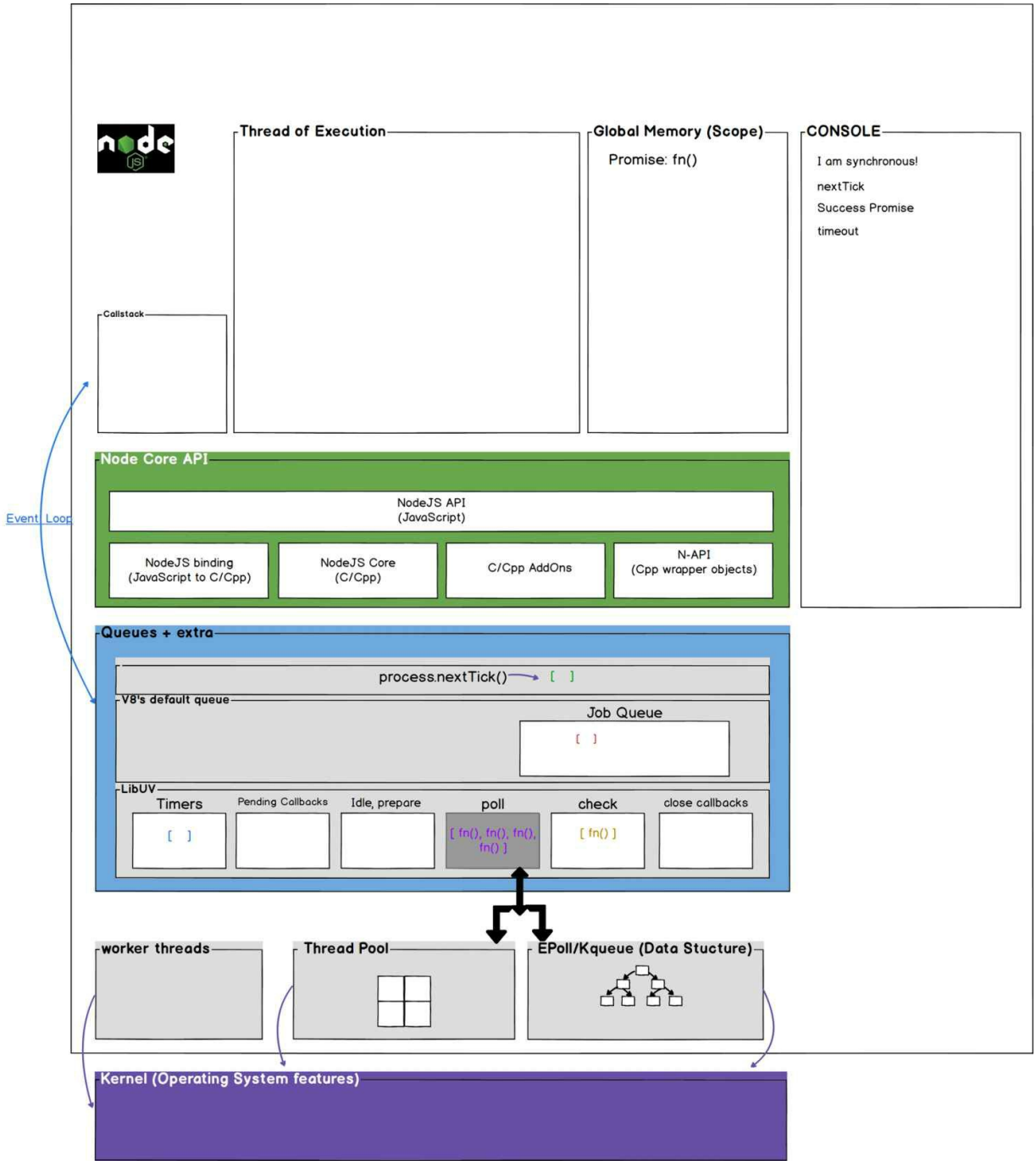

Thread of Execution
body of fn
Local Memory
console.log(val);
val: 'Success Promise'
Global Memory (Scope)
Promise: fn()
CONSOLE
I am synchronous!
nextTick
Success Promise
Callstack
Event Loop
Node Core API
NodeJS API
(JavaScript)
NodeJS binding
(JavaScript to C/Cpp)
NodeJS Core
(C/Cpp)
C/Cpp AddOns
N-API
(Cpp wrapper objects)
Queues + extra
process.nextTick()
[ ]
V8's default queue
Job Queue
[fn(val)]
LibUV
Timers
Pending Callbacks
Idle, prepare
poll
check
close callbacks
[fn()]
[fn(), fn(), fn(), fn()]
[fn()]
worker threads
Thread Pool
EPoll/Kqueue (Data Stucture)
Kernel (Operating System features)



- Now the next turn is for our timers😉.

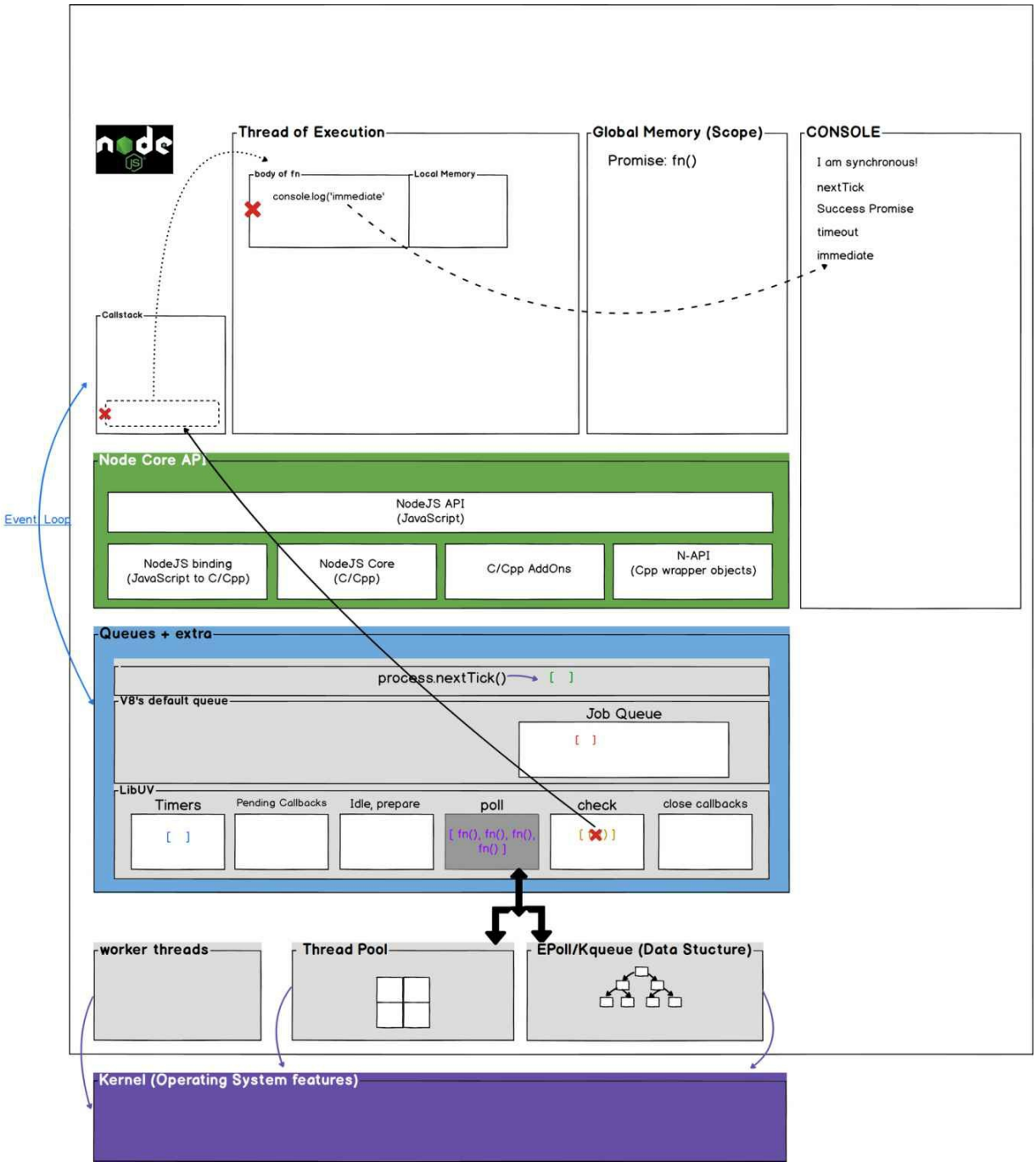

Thread of Execution
body of fn
Local Memory
console.log('timeout');
Global Memory (Scope)
Promise: fn()
CONSOLE
I am synchronous!
nextTick
Success Promise
timeout
Callstack
Node Core API
NodeJS API
(JavaScript)
NodeJS binding
(JavaScript to C/Cpp)
NodeJS Core
(C/Cpp)
C/Cpp AddOns
N-API
(Cpp wrapper objects)
Event Loop
Queues + extra
process.nextTick()
[ ]
V8's default queue
Job Queue
[ ]
LibUV
Timers
Pending Callbacks
Idle, prepare
poll
check
close callbacks
[fn()]
[fn(), fn(), fn(), fn()]
[fn()]
worker threads
Thread Pool
EPoll/Kqueue (Data Stucture)
Kernel (Operating System features)

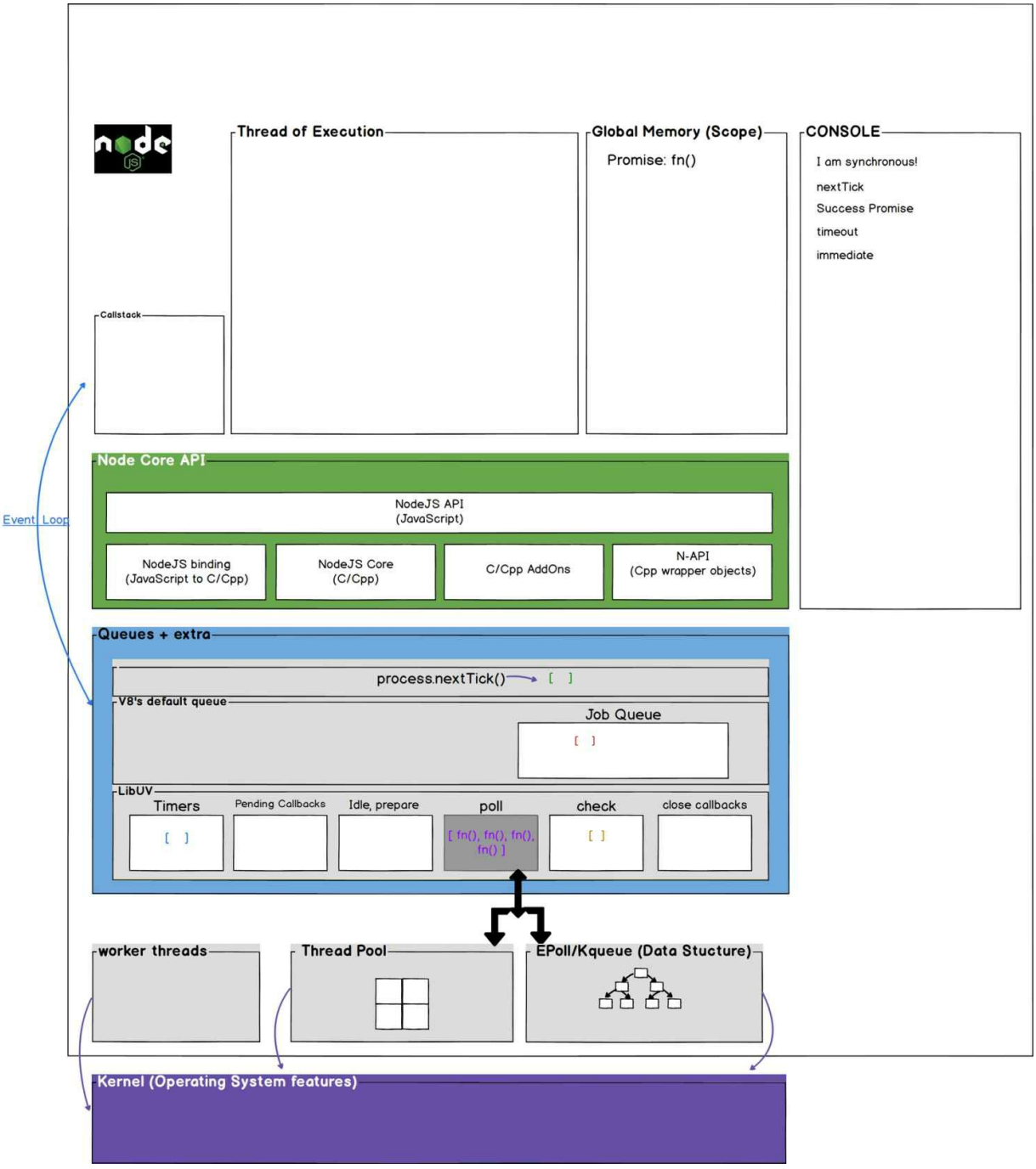

Thread of Execution
body of fn
Local Memory
console.log('timeout'),
Global Memory (Scope)
Promise: fn()
CONSOLE
I am synchronous!
nextTick
Success Promise
timeout
Callstack
Node Core API
NodeJS API
(JavaScript)
NodeJS binding
(JavaScript to C/Cpp)
NodeJS Core
(C/Cpp)
C/Cpp AddOns
N-API
(Cpp wrapper objects)
Event Loop
Queues + extra
process.nextTick()
[ ]
V8's default queue
Job Queue
[ ]
LibUV
Timers
Pending Callbacks
Idle, prepare
poll
check
close callbacks
[ ]
[fn(), fn(), fn(), fn()]
[fn()]
worker threads
Thread Pool
EPoll/Kqueue (Data Stucture)
Kernel (Operating System features)



- This time turn is for the poll queue, but the poll queue has 4 operations -> 2 file reads and 2 GET API calls. so, internally timeout is calculated by libUV whether these operations are going to be finished quickly in a certain period of time (usually a very few seconds or not) or not. If not,

then immediately we move to check queue. That is why here we are quickly going to move for setImmediate()'s callback from the check queue without waiting for 4 operations(2 file reads + 2 GET API calls).

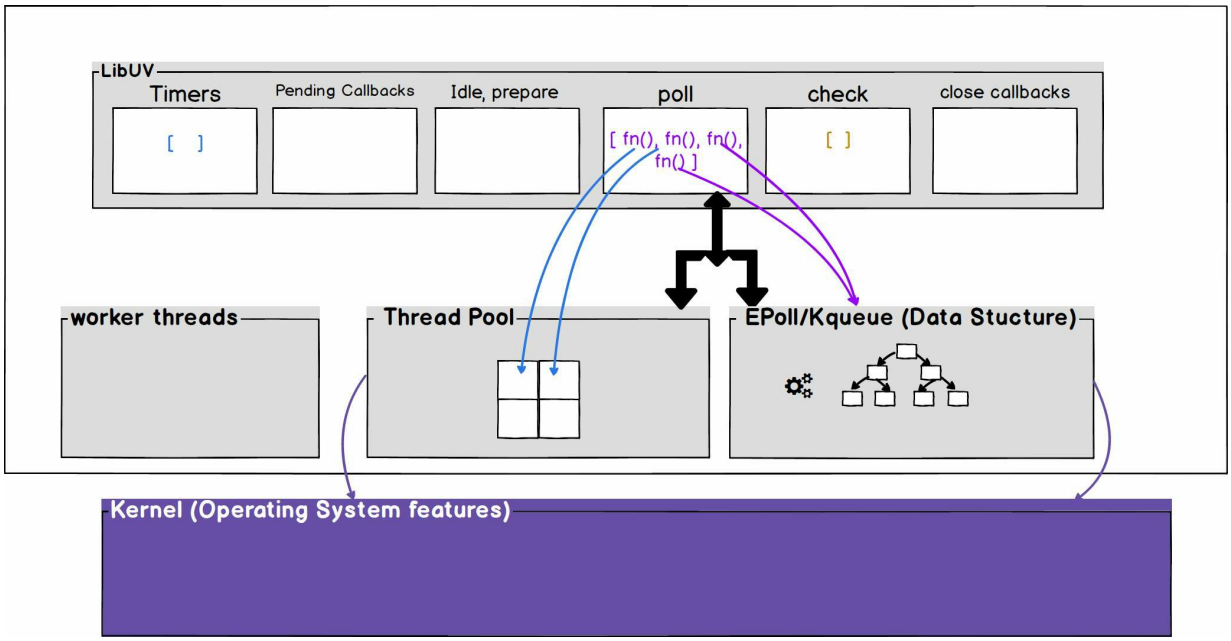




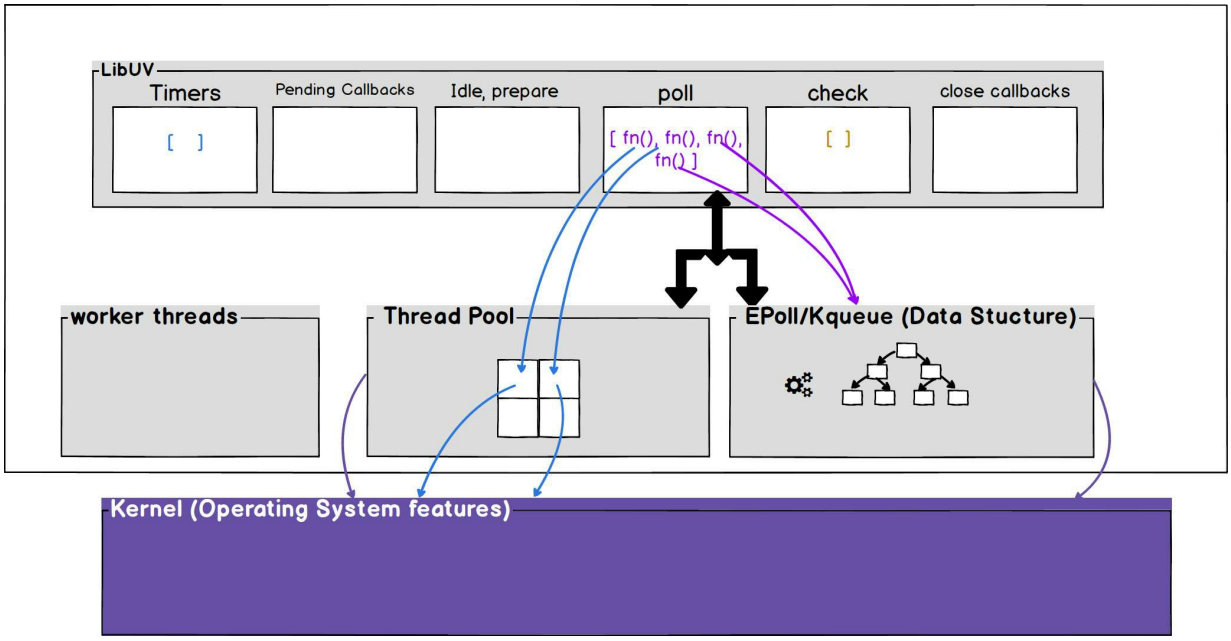
- Now, all queues are empty and all pending task has finished except poll queue. Let's see how the **poll queue** is going to perform these tasks.
- Poll queue delegate task in major 2 categories.
- All operations like file read, crypto and zLib are done through the

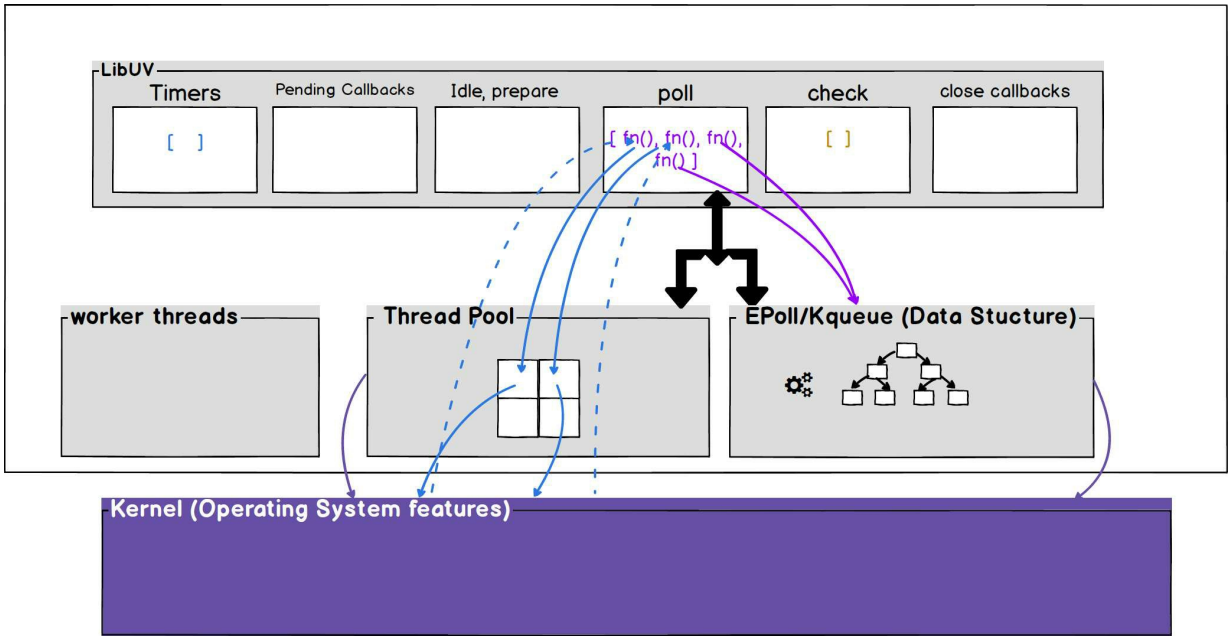
Thread pool.

- **A Thread pool is a collection of worker threads.** (it's different from worker threads provided by node's default module). The default size of the thread pool is 4 but you can overwrite using the **UV_THREADPOOL_SIZE** global variable provided by the node. I have drawn 4 small boxes for thread inside the thread pool.
- One thread can perform one task, so 4 threads can perform 4 tasks simultaneously.
- However, as I said file read, crypto, and zLib like tasks only perform in thread pool by design. So in our case, both files read operation is done through thread pool and 2 file operation handle through 2 separate threads and other 2 GET APIs calls are done by some other mechanism like Epoll/Kqueue(will see soon).

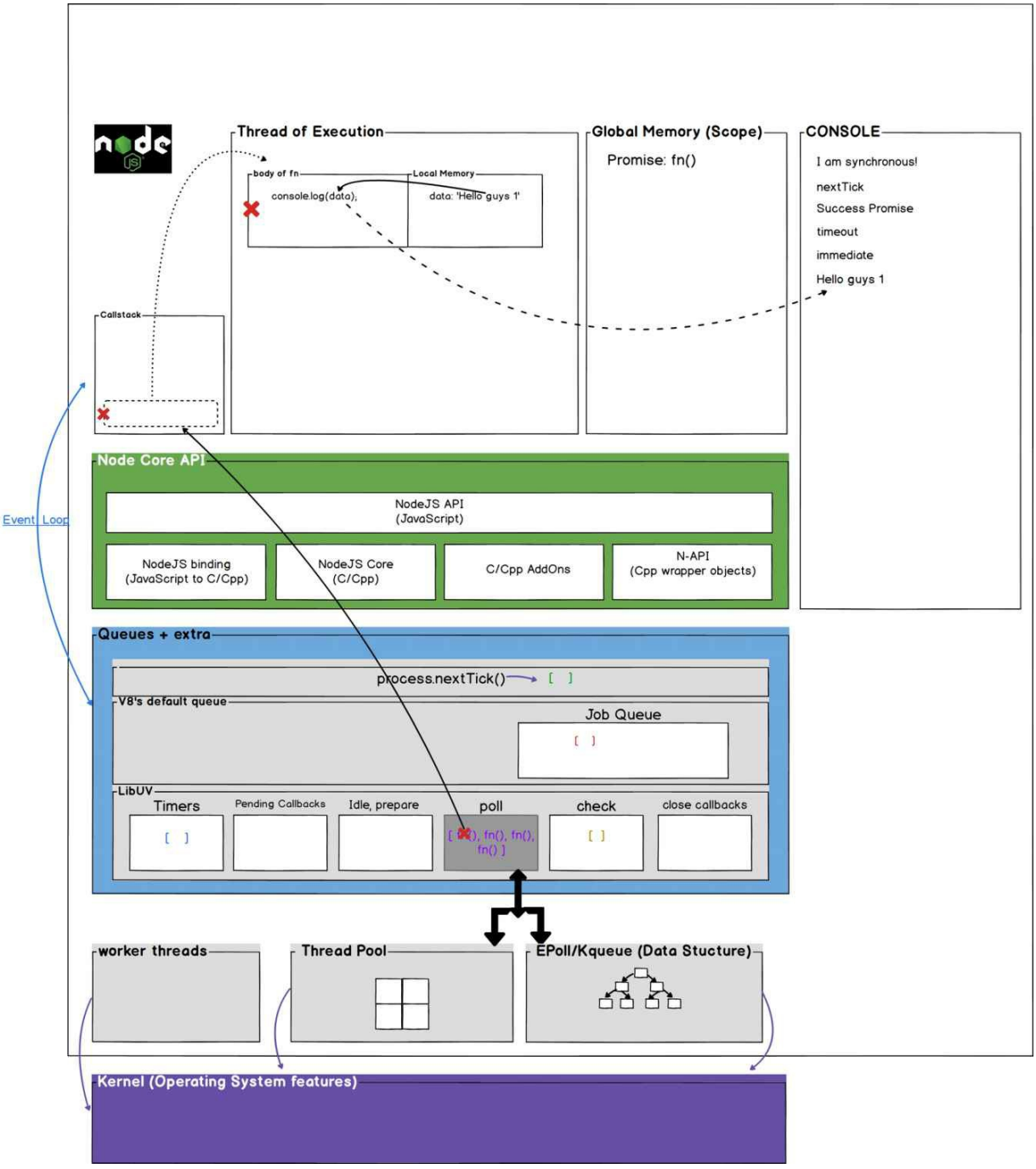


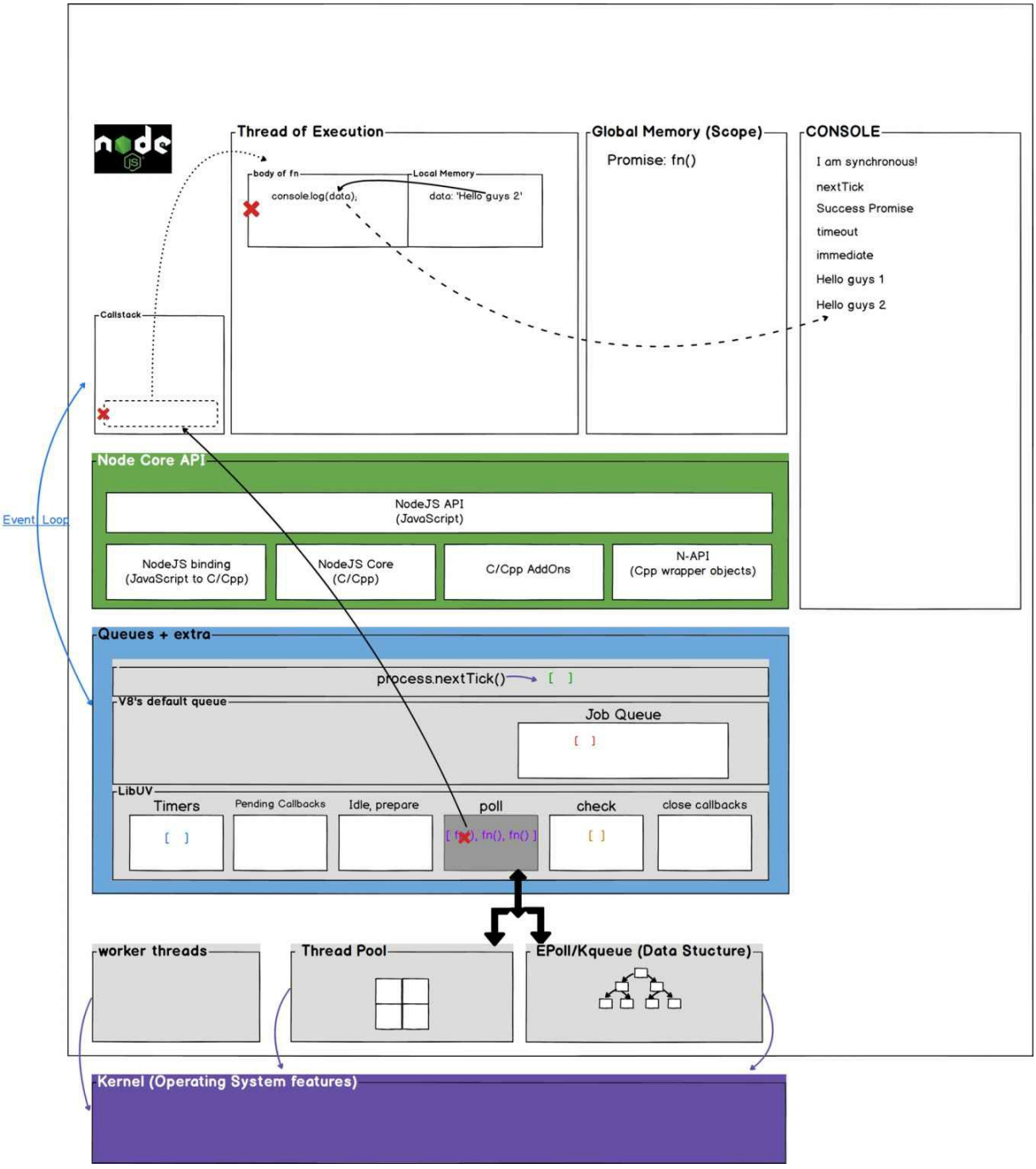
- As you can see we are processing 2 file read operations in 2 threads and when they finish reading result is given back to respective callbacks.



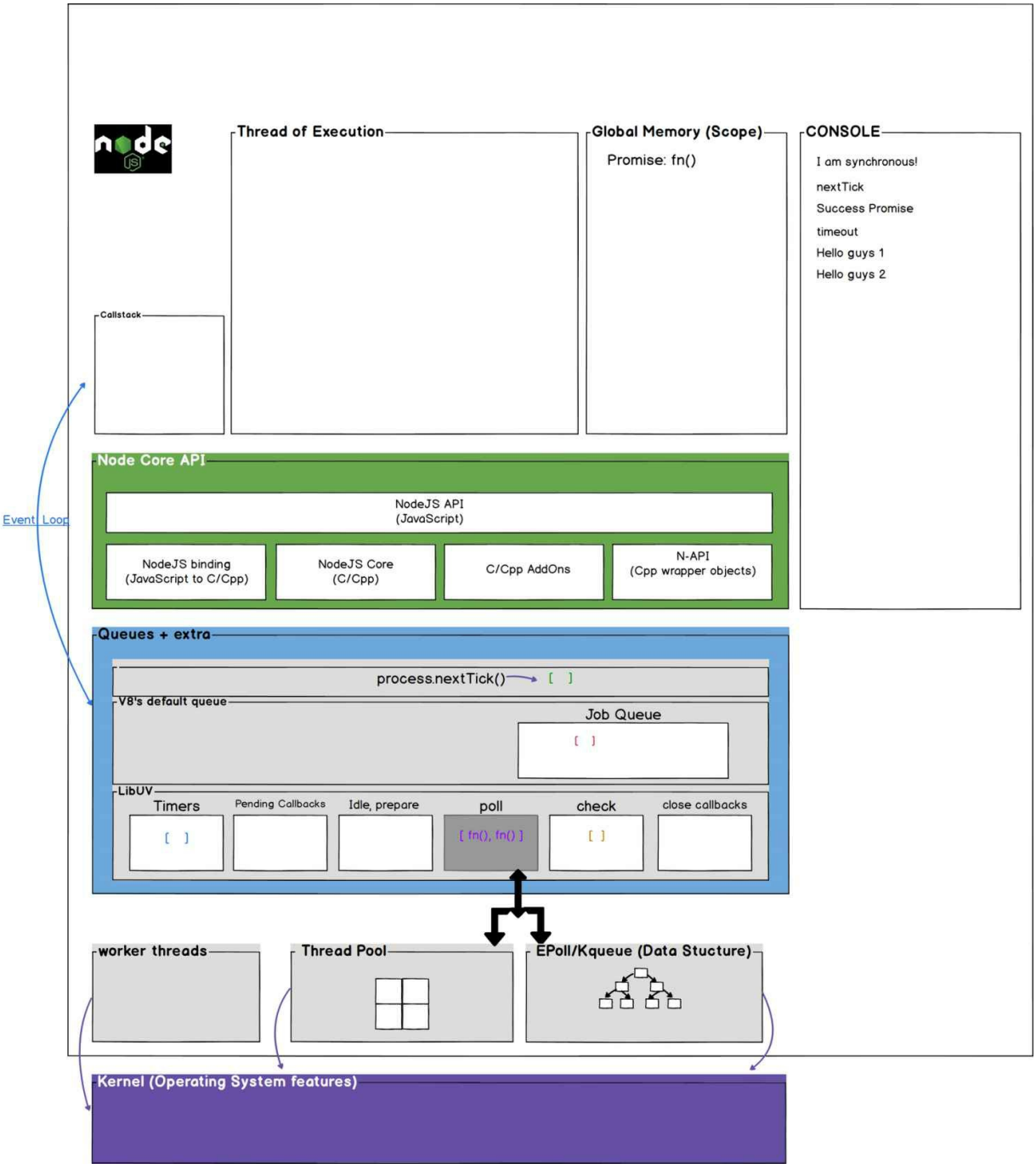


- Let’s finish the execution of the completed 2 read operations and run callbacks of it(Because we are not going to block the poll queue while waiting to finish API calls).


Thread of Execution
body of fn
Local Memory
console.log(data);
data: 'Hello guys 1'
Global Memory (Scope)
Promise: fn()
CONSOLE
I am synchronous!
nextTick
Success Promise
timeout
immediate
Hello guys 1
Callstack
Event Loop
Node Core API
NodeJS API
(JavaScript)
NodeJS binding
(JavaScript to C/Cpp)
NodeJS Core
(C/Cpp)
C/Cpp AddOns
N-API
(Cpp wrapper objects)
Queues + extra
process.nextTick()
[ ]
V8's default queue
Job Queue
[ ]
LibUV
Timers
Pending Callbacks
Idle, prepare
poll
check
close callbacks
[ ]
[fn(), fn(), fn(), fn()]
[ ]
worker threads
Thread Pool
EPoll/Kqueue (Data Stucture)
Kernel (Operating System features)



- Great, now we have finished file read operations as well, but still, we have 2 more operations to complete which are our 2 API calls. So, again we go through the next iteration of the event loop.

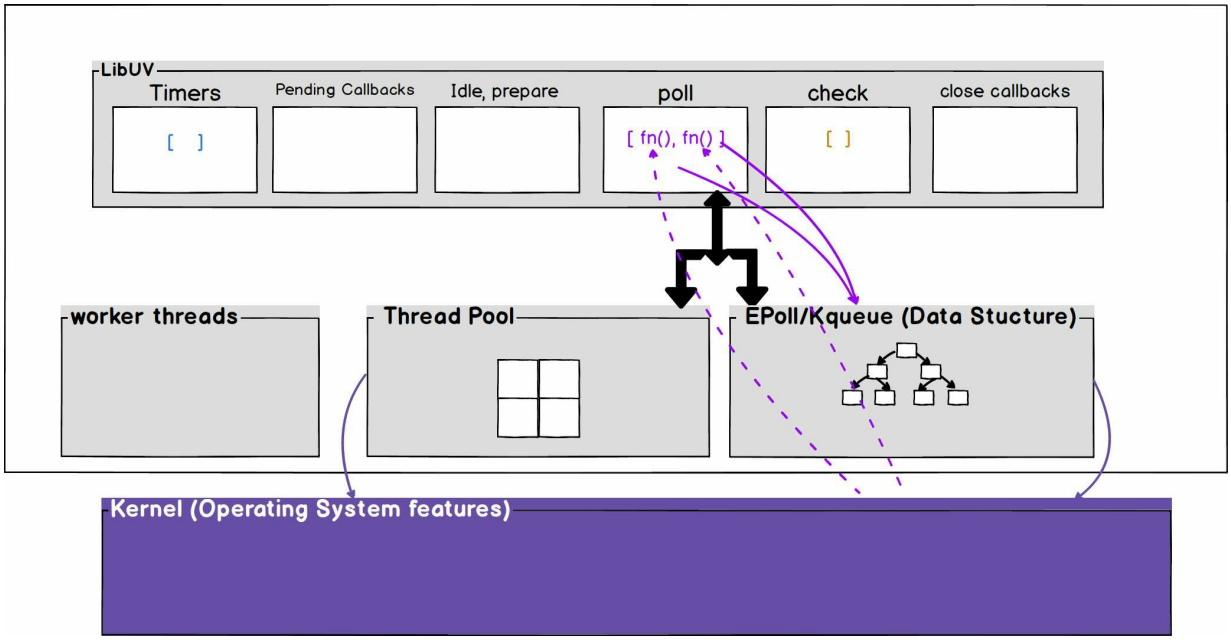


- Now the time has come how **epoll/Kqueue** and **poll queue** work together. I will explain epoll, not other mechanisms because all other mechanisms are almost the same only difference is it’s based on Operating System Architecture.

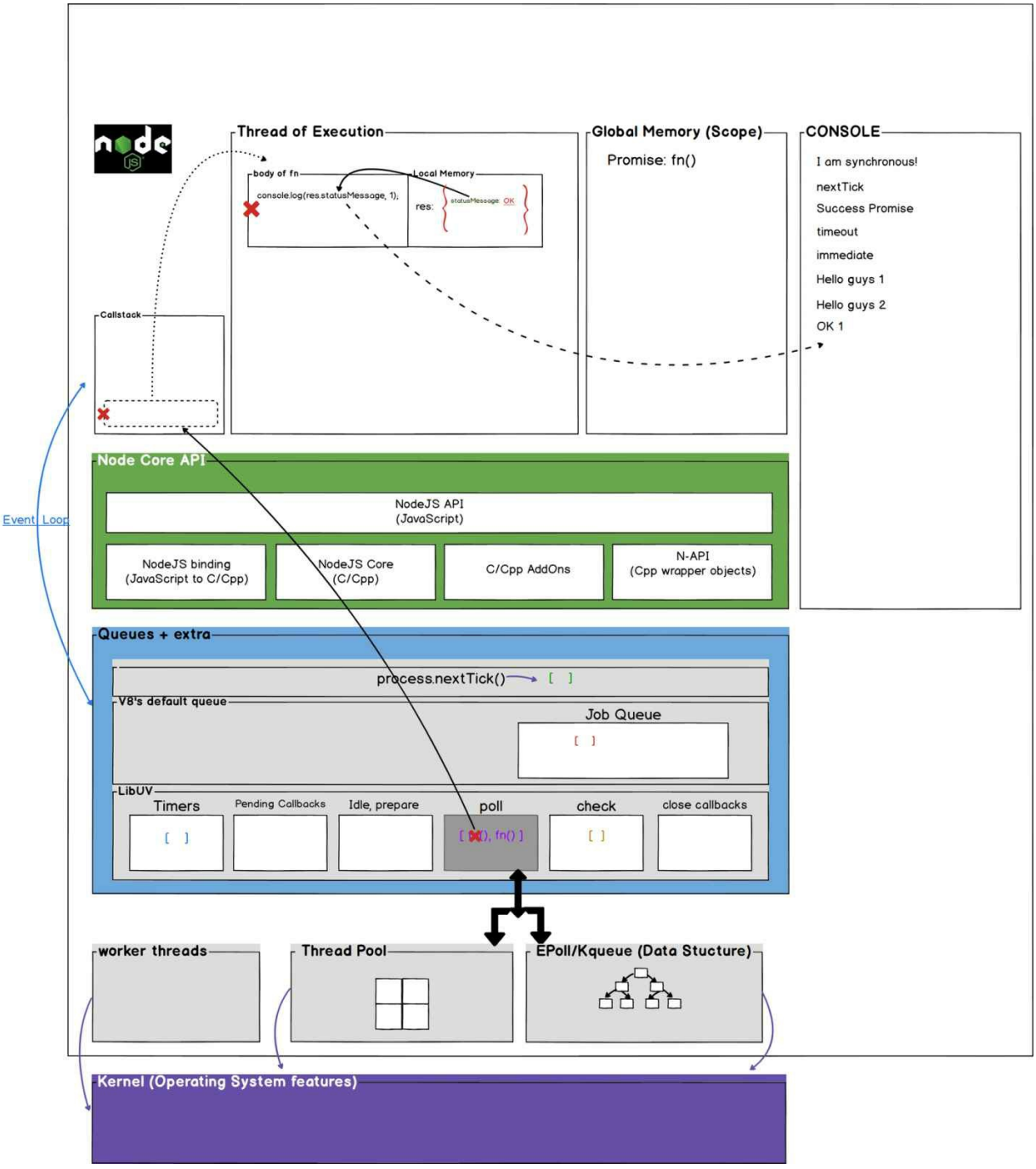
epoll(Assuming we are on Linux):

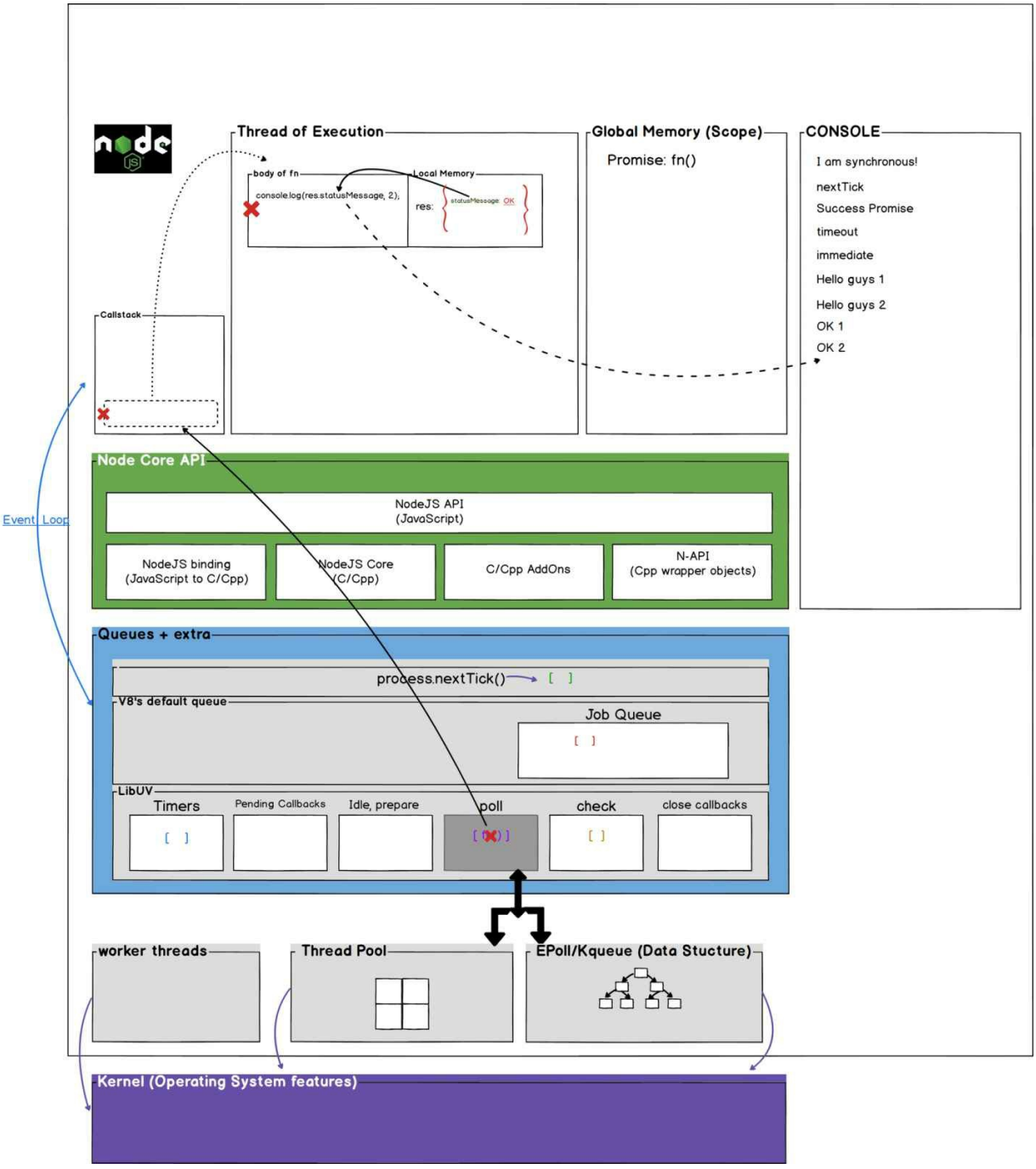
- It’s a self-balancing binary tree data structure.
- We attach task/s to a node of this data structure and loop through each task until some timeout.
- Let’s assume we have 10 API requests that come to the running node process at the same time, all are actually given to epoll.
- Before giving all requests to epoll, requests are opened in a non-blocking mode (It’s a Linux feature that allows you to open the files/fulfill requests in non-blocking mode and wait for them until it’s done, and then the result is given back).
- Now in our assumption, we said we have 10 simultaneous requests which we are going to open in non-blocking mode and its id(they call it file descriptor) is stored in epoll. each id will have events attached to it. (here, our callback functions).
- Generally, we loop over epoll for a certain time period. let’s say epoll is looping for 2 seconds, so out of 10 requests, 4 requests are done in 2 seconds. so, the result of it given back to epoll and from epoll to our **poll queue.**
- After finishing 4 requests, we move immediately to the **check queue**.
- On the next iteration of the event loop, let’s say in our epoll we completed 3 more requests(remember here we are literally looping in epoll for every 2 seconds to check whether work is done or not in the background). These 3 requests with data given back to the **poll queue.**
- On the next iteration, let’s say we fulfill all remaining 3 requests, and the result is given back to the **poll queue.**

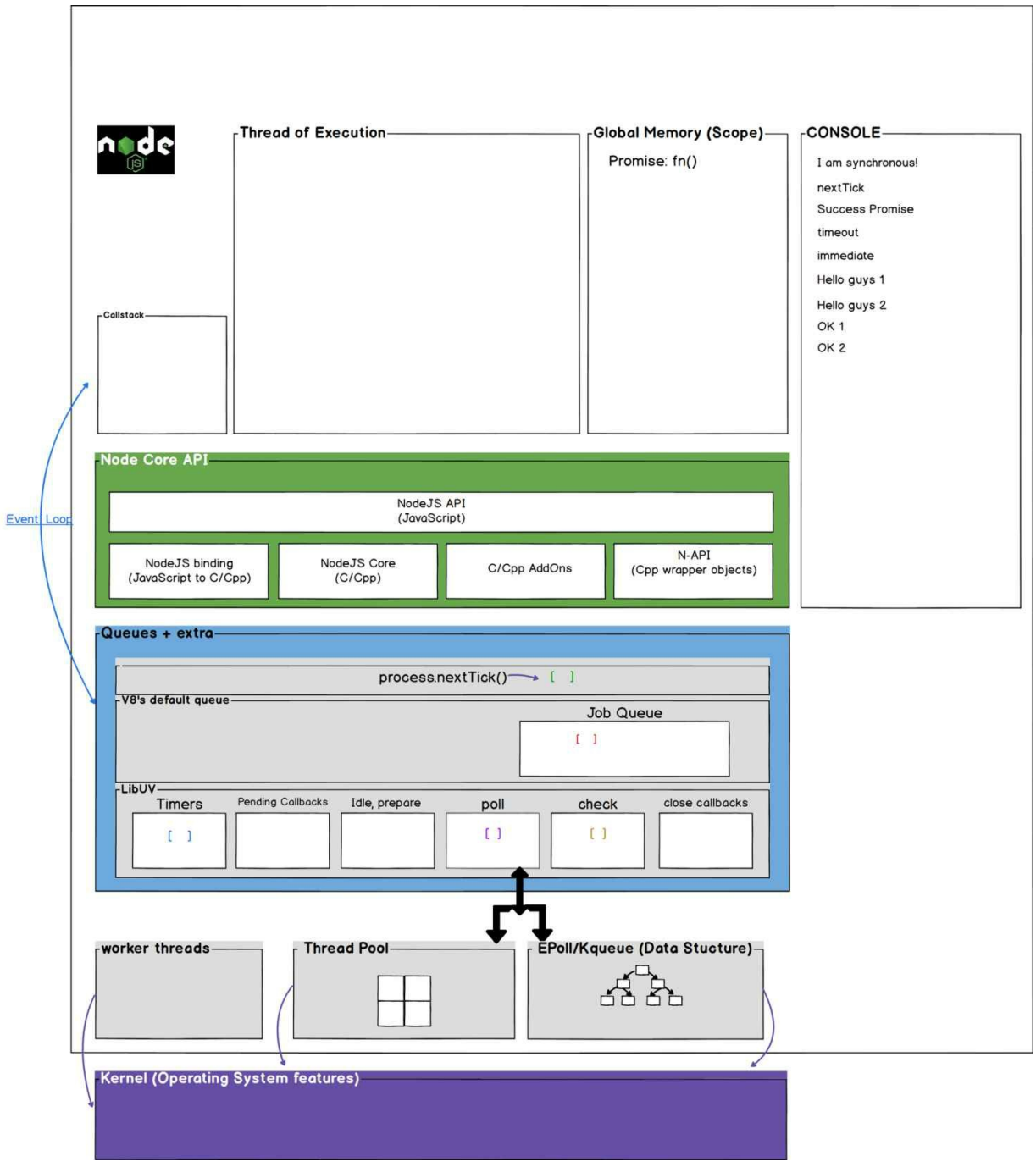
```
Putting requests in non-blocking mode via kernel's feature and handling stuff
through Epoll data structure will give amazing concurrency without blocking main
javascript thread as well as without blocking poll queue and spinning new threads.
This is the reason why nodejs is so popular and amazing.
```



- In our case, we have 2 GET API calls. We will process them through epoll as we just discussed and the result is given back to the poll queue. Let’s finish our visualization of the remaining finished 2 API calls.


Thread of Execution
body of fn
console.log(res.statusMessage, 1);
Local Memory
res:
statusMessage: OK
Global Memory (Scope)
Promise: fn()
CONSOLE
I am synchronous!
nextTick
Success Promise
timeout
immediate
Hello guys 1
Hello guys 2
OK 1
Callstack
Event Loop
Node Core API
NodeJS API
(JavaScript)
NodeJS binding
(JavaScript to C/Cpp)
NodeJS Core
(C/Cpp)
C/Cpp AddOns
N-API
(Cpp wrapper objects)
Queues + extra
process.nextTick()
[ ]
V8's default queue
Job Queue
[ ]
LibUV
Timers
Pending Callbacks
Idle, prepare
poll
check
close callbacks
[ ]
[ ]
[ ]
worker threads
Thread Pool
EPoll/Kqueue (Data Stucture)
Kernel (Operating System features)


Thread of Execution
body of fn
console.log(res.statusMessage, 2);
Local Memory
res:
statusMessage: OK
Global Memory (Scope)
Promise: fn()
CONSOLE
I am synchronous!
nextTick
Success Promise
timeout
immediate
Hello guys 1
Hello guys 2
OK 1
OK 2
Callstack
Event Loop
Node Core API
NodeJS API
(JavaScript)
NodeJS binding
(JavaScript to C/Cpp)
NodeJS Core
(C/Cpp)
C/Cpp AddOns
N-API
(Cpp wrapper objects)
Queues + extra
process.nextTick()
V8's default queue
Job Queue
LibUV
Timers
Pending Callbacks
Idle, prepare
poll
check
close callbacks
worker threads
Thread Pool
EPoll/Kqueue (Data Stucture)
Kernel (Operating System features)



Now, our all execution has finished as above and all queue became empty so our program will exit. That's all my friends this is how node actually works 😎. you learned complex node architecture with the proper mental model😁.

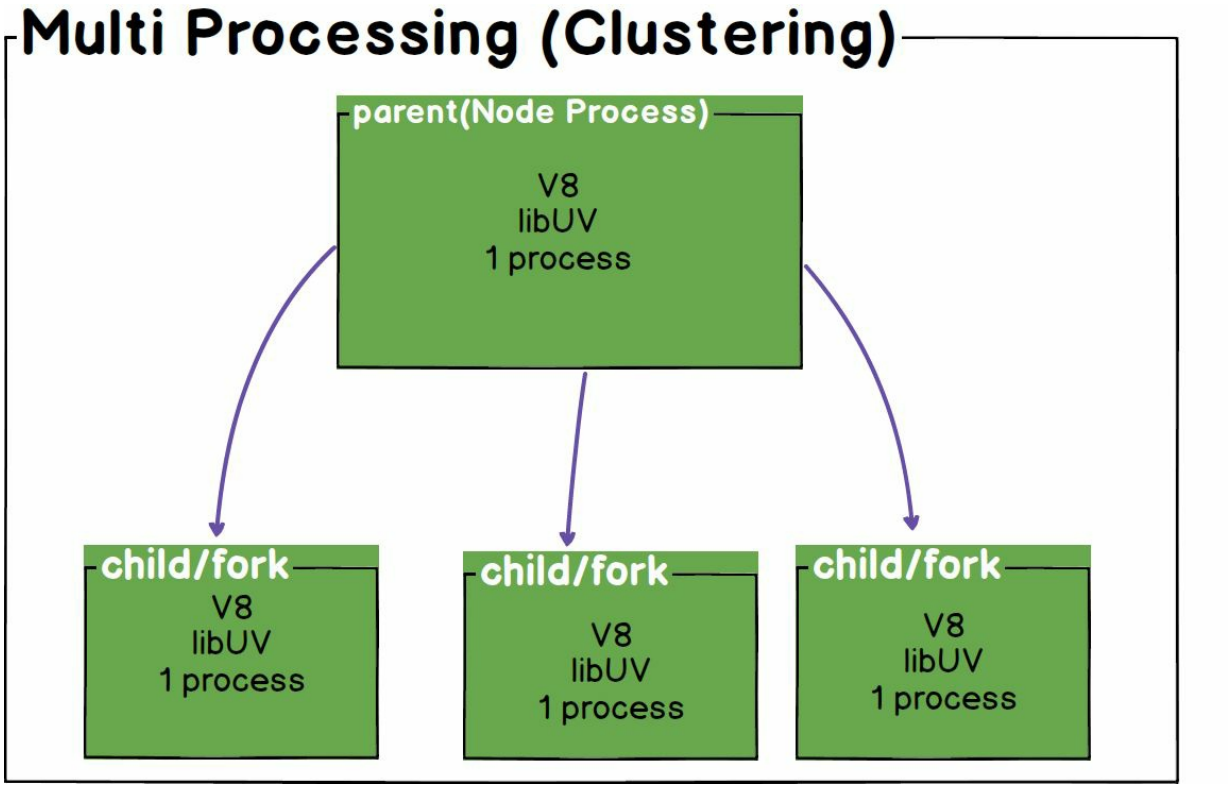
```
Note: Here I have assumed that file1.txt read operation done before file2.txt and
the same way the first API call finished before the second API call. Which
```

```
operation out of 2 files read will complete before it depends on your OS and
processer. The same goes for API calls.
```

Now let’s talk about some confusing stuff like **multi-threading, worker threads, and multi-processing** from the node’s perspective.

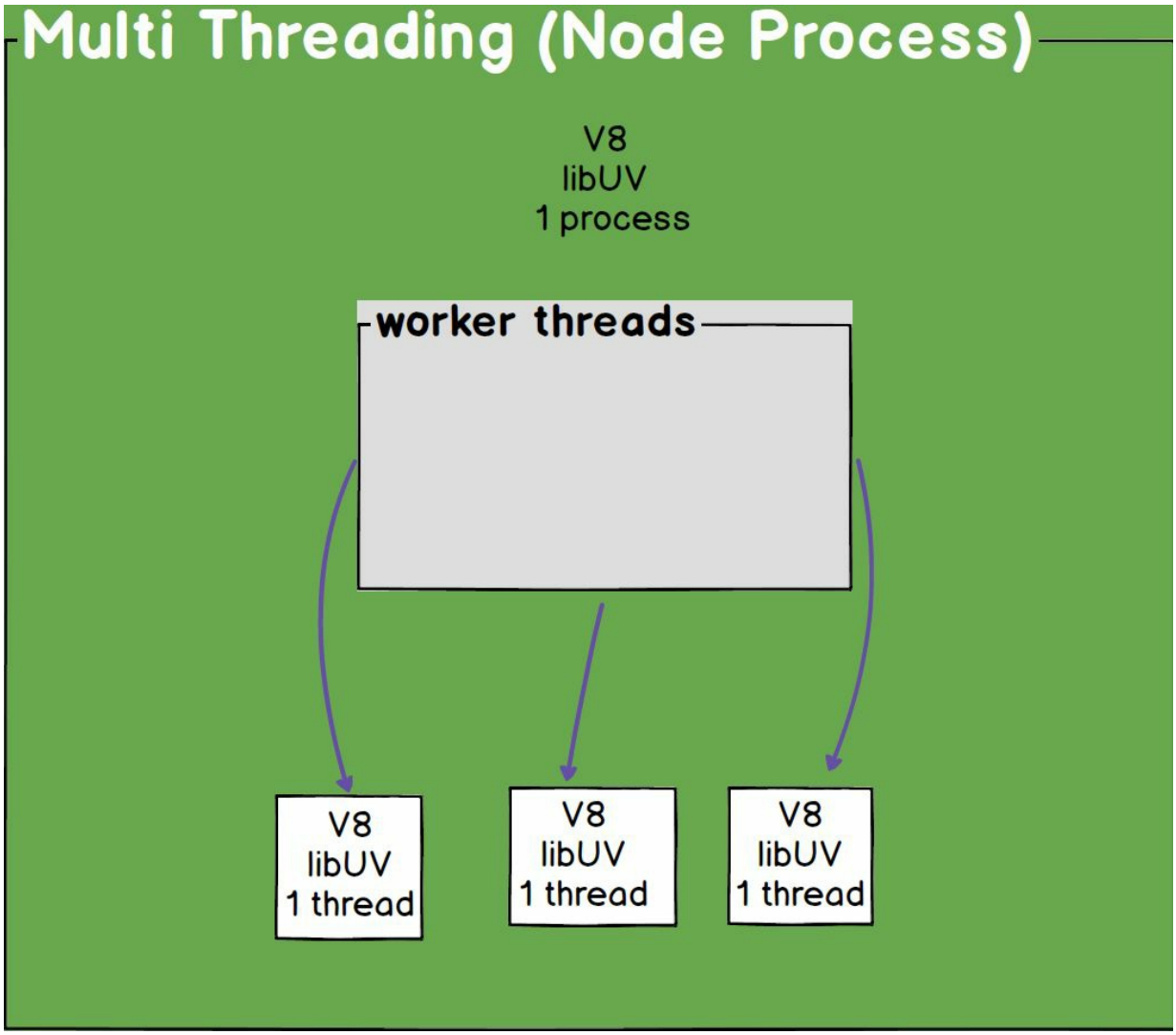
Multi-processing

- We know, when we start the node program we essentially start the node process in which we will have 1 instance of v8, 1 instance of event loop via libUV, and node APIs core layer.
- Now, when we use **child_process,** we create a new separate process from the parent process(from our node process). we can do anything in the newly created process.
- The important thing to note here, each newly created process will have its own 1 instance of v8, 1 instance of event loop via libUV as well. That is the reason why we can parse javascript inside child_process as well if we want.
- If you are a node developer then you most likely know the module called **‘Cluster’,** which is fundamentally used to create a fork/s of our running node process and each process will be put in the logical core of your processor!
- Let’s visualize it.


Multi Processing (Clustering)
parent(Node Process)
V8
libUV
1 process
child/fork
V8
libUV
1 process
child/fork
V8
libUV
1 process
child/fork
V8
libUV
1 process

Worker Threads

- This is a separate module available in the node which allows you to create real threads and we can run javascript code in that newly created thread as well.
- each thread will have its own v8 and libUV instance in it.



- The worker thread can communicate to the parent process's main thread via a communication channel and not only that we can pass data back

and forth to different threads as well😉.

- Threading in the node is a little different from other programming languages threading concepts. as we can see we can run CPU-intensive tasks in a separate thread but each thread will have its v8 and libUV instances. Because each thread needs v8 and libUV to perform the task and parse javascript code threads in the node are a little more memory consuming.
- So, even create a tone of threads in a node we will surely run out of memory in our system or we will overload our system with resource-hungry threads.
- So, it’s better to use worker threads only for big CPU-intensive tasks in the node. Never use worker threads for serving API requests or serving files.

Multi-processing vs worker threads

- It’s best to use multi-processing for serving node processes on each CPU-core. Also, inside each node process, we will have enough memory and storage such that multiple API requests can be accumulated and later delegate to epoll and thread pool via poll queue to give us really good concurrency.
- Worker threads are can be useful to do CPU-intensive tasks like calculating highly mathematical formulas in a separate thread and giving back results to the parent thread (our node main process’s thread).
- We can combine the best of both worlds with creating multiple node processes and for each process for specific CPU-intensive tasks we use worker threads😎.

Worker threads vs Thread pool

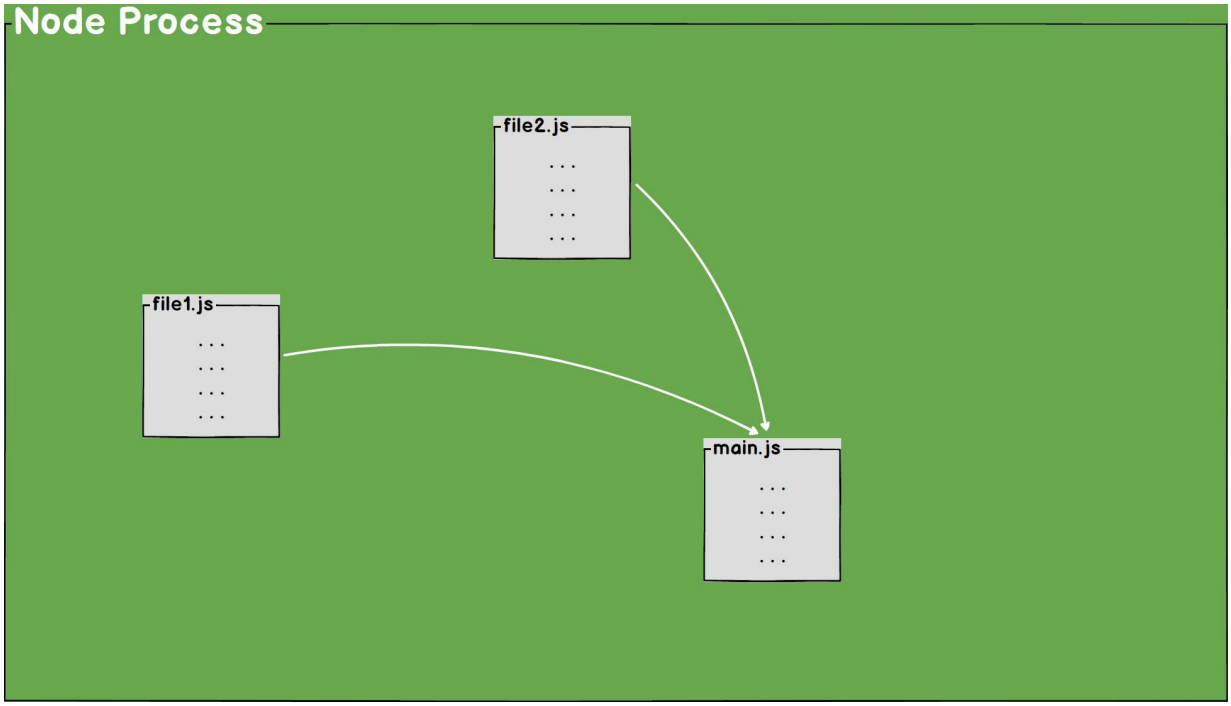
- It’s pretty simple, Thread pool has a worker thread(threads which can perform specific task/s directly) that is used automatically by libUV whenever we do File I/O, crypto, and zlib operations.
- **‘Worker threads’**(module provided by node) are created by us to do a specific task/s in a completely different thread/s.
- Now, **thread pool’s** threads are easy to maintain and fast as they don’t need v8 and libUV instances in it because tasks have already been known to thread pool ahead of time.
- **‘Worker threads’** are little resource takers so we don’t create too much of them. (here, we don’t create threads for serving each client request as we do in other traditional programming languages like JAVA, because if we do that then it’ll make our system more resource hungry and unstable. it’s totally different concept compare to them. we should always rely on node’s non-blocking I/O for these kinds of purposes).

```
I hope all the above information has cleared all wrong assumptions and
misconceptions regarding nodejs☺.
```

- **Now, before we close our journey of the node, I like to explain how the node’s module system works.**
  JavaScript is run in different environments, and we don’t have a native module system by default available to us (ignore ES6+ native modules which came recently, will discuss later).
- Now, when nodejs developed they need to figure out some way how to give a modularity system to developers, so they created their own

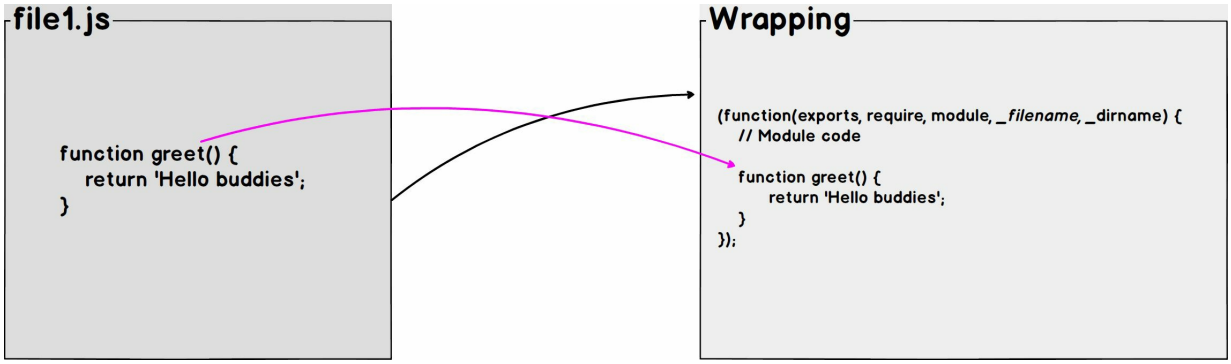
module system which they call **“commonjs”**. Let’s start understanding how it works with visualization.

- In Nodejs, each javascript file that we are going to use in our project is **treated as a “module” which means they are not just normal javascript code sitting inside the file, but they have some extra wrapper and pieces of stuff.**
- Also, nodejs provides us some global variables which are available to all “modules”. we call them **global variables**. Out of those 2 variables, the **“require” function and “module” object**(Note that our file “module” and global keyword “module” are 2 different things here😄)**.**
- Let’s visualize whatever we have learned so far.

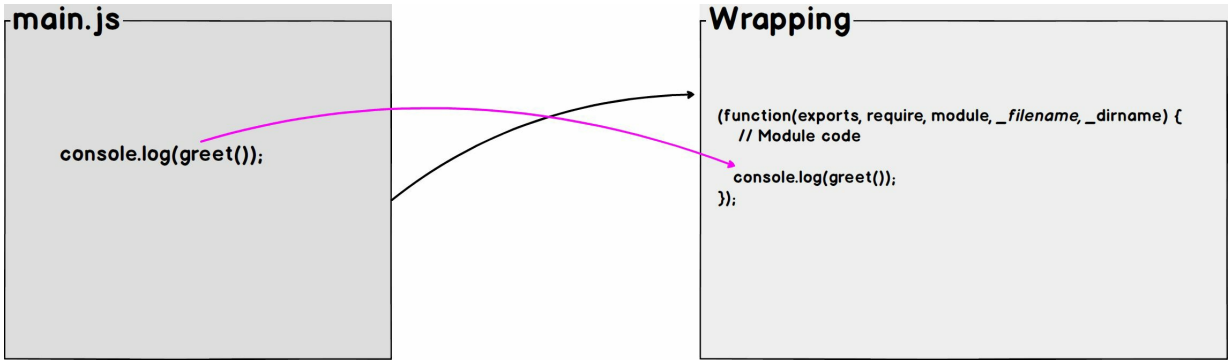


- Suppose, we want to add some content of other files(file1.js & file2.js) to our main file(main.js).

- As I said earlier, in running the node process we have some global variables available to us by default. So, here we are going to need one of those variables to accomplish our needs.
- The **require() function** is made available to us in each file(module) which means I can use that helper function in the **main.js** as well.
- But nodejs put something call **IIFE(Immediately invoked function expression)** to all our modules(js files) before it even going to required by another module. let's see that.

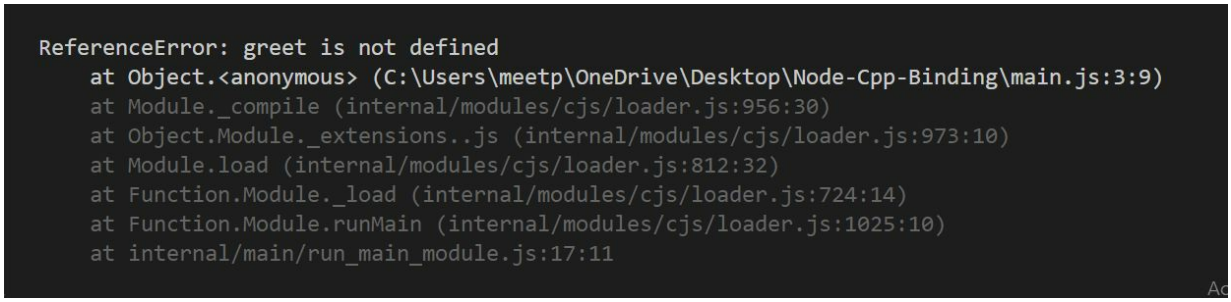


- As we can see, in our file1.js we have simple **greet() function,** which will be wrapped around by the IFEE function as shown in the figure(only if we require file1.js in another module then and only of course).
- In the same way, our main.js(main module) also get wrapped around as well. Because when we are going to start our node process we are going to start from our main module.



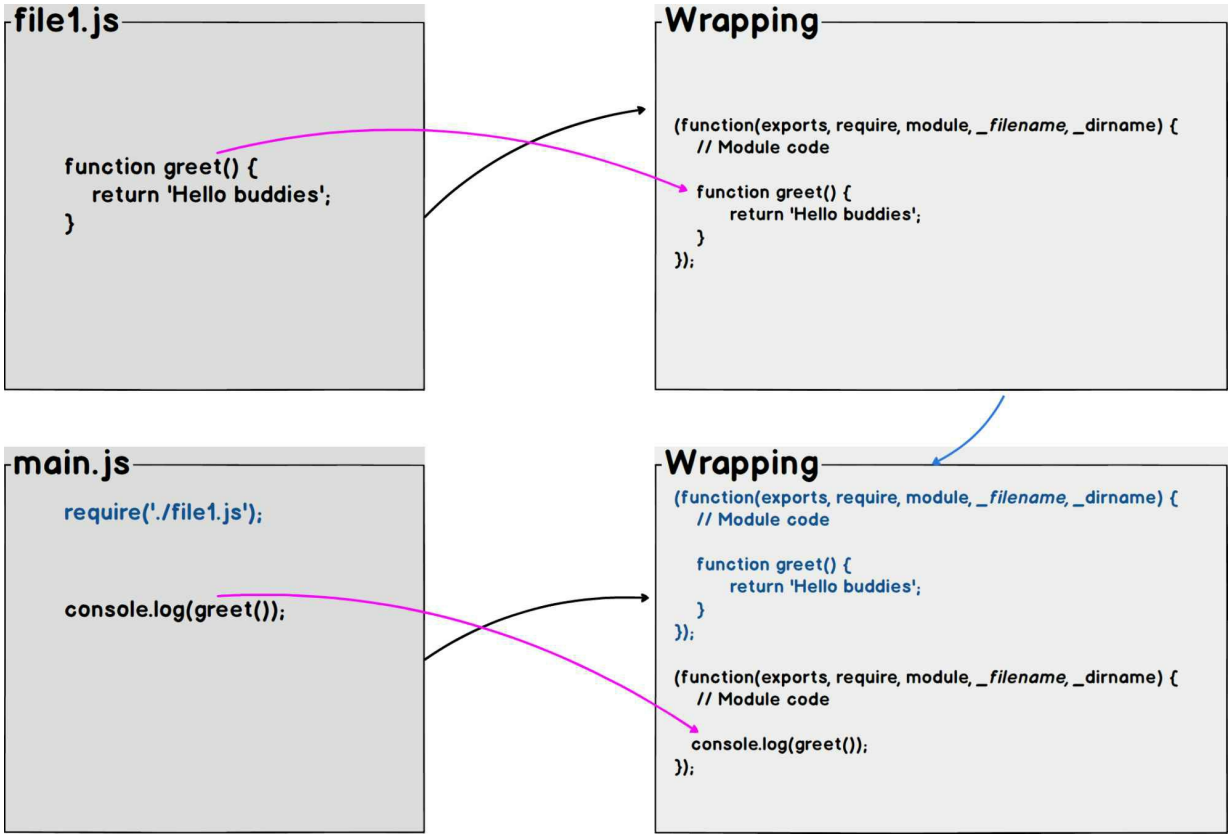
- One thing if you have noted then, we have a bunch of arguments/variables available to wrapper IFEE and one of them is **“require”.** let’s see what happens when we use require in our main module.
- The **require()** function takes the file path as an argument to find another module.

- If you try to run the above code then it will give a “module” not found error.



- But why?
- Remember I said every module is wrapped with IFEE, so when we

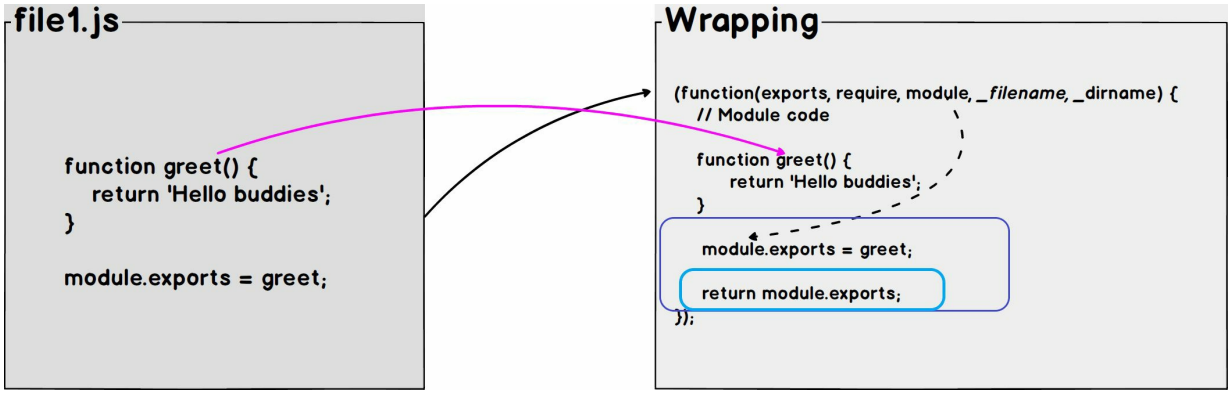
required a module we actually grab the IFEE not the code of the module 😮. let's visualize it.



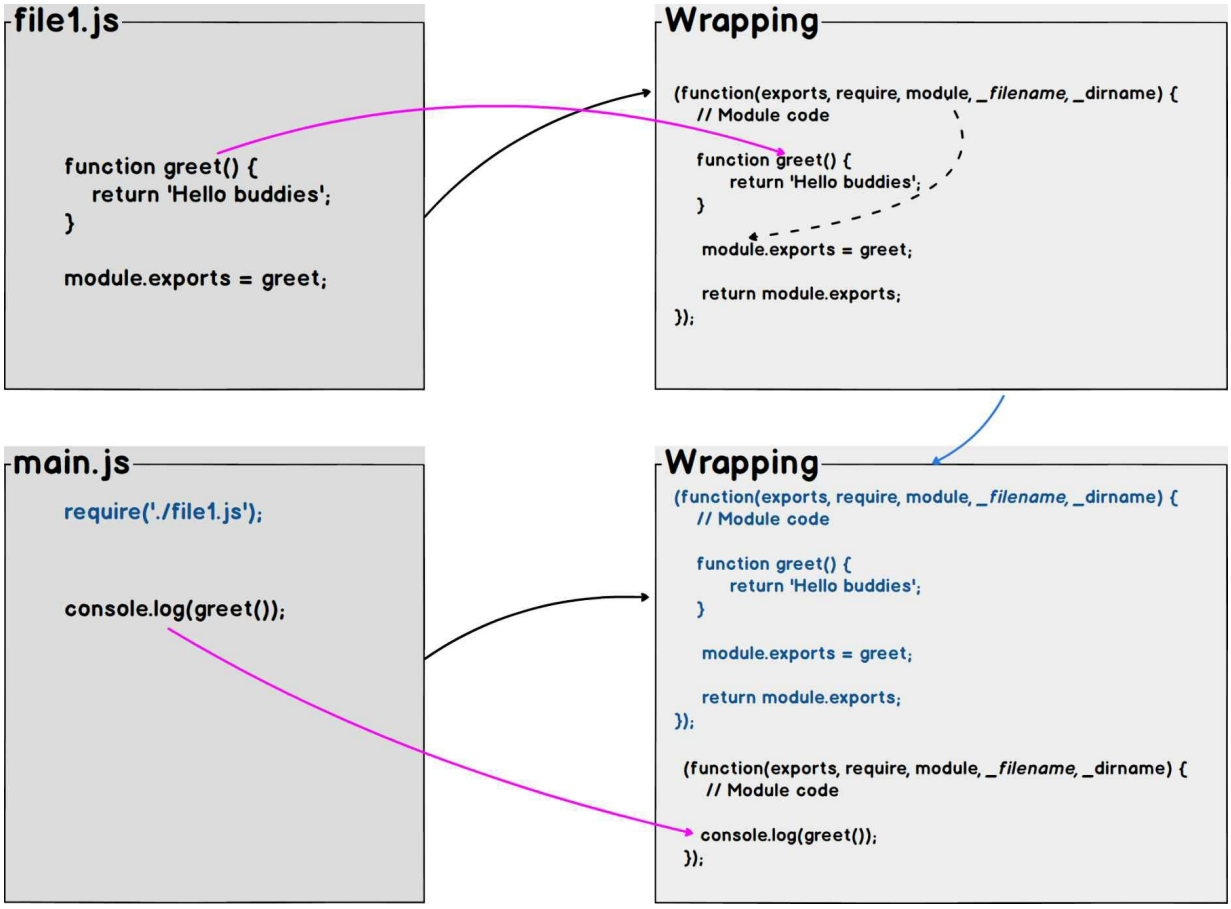
- As we can blue color content is actually what **"require()"** does. so, it's wrapped function and all variables and function inside that are locally scoped. That's is the reason why we were not able to execute the **greet()** function and got the error.
- But how then we are able to get this work?
- That is where our **arguments** passed to IFEE are come into the picture.
- If we want to use the **greet()** function then we need to somehow take it outside of the IFEE scope when we require it.
- Nodejs's IFEE has a **"module" object** available to us via argument and

there is a “**exports”** key available to us on the **“module” object** (see diagram, the 3rd argument of IFEE).

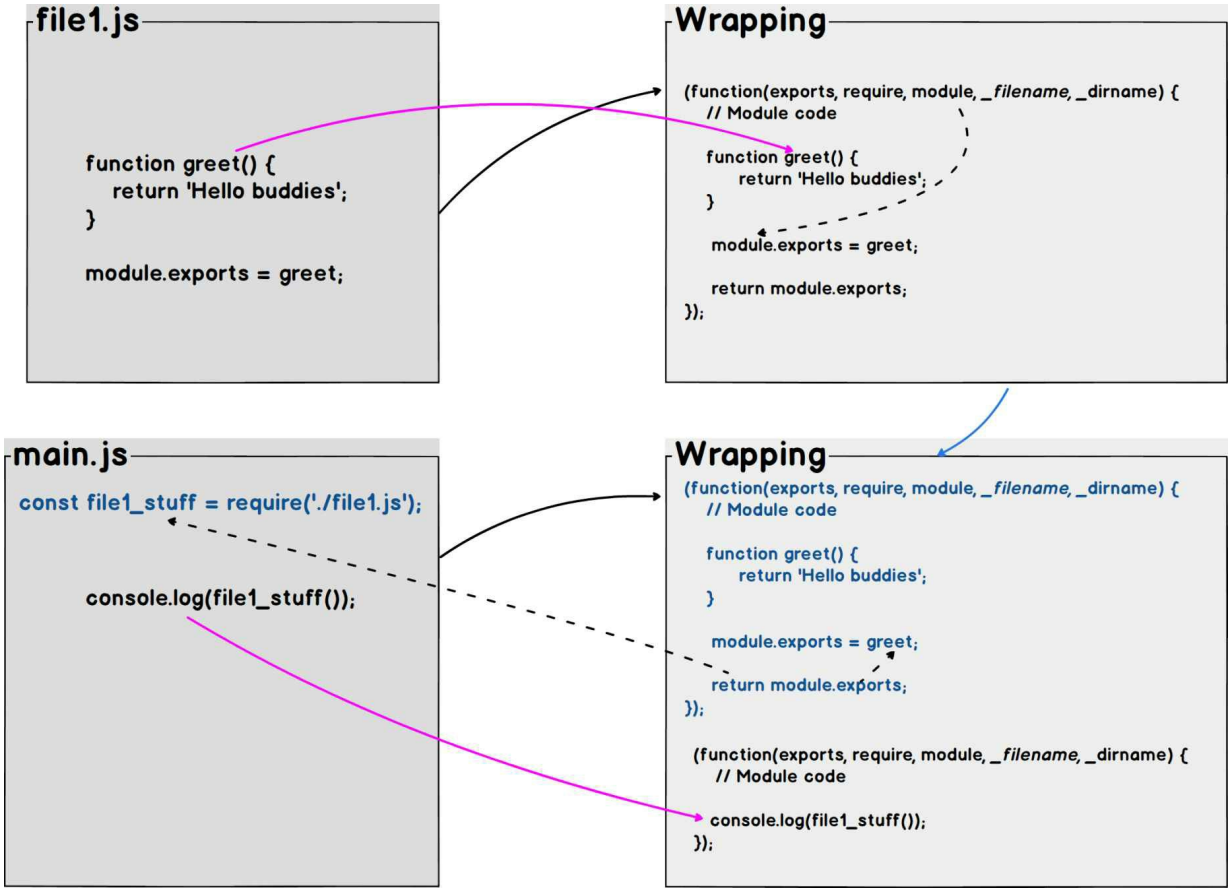
- So, if I use a **“module.exports” and add properties to it,** then when we require that it in another module it will have access to it. let’s visualize it because it’ll easy to grasp.



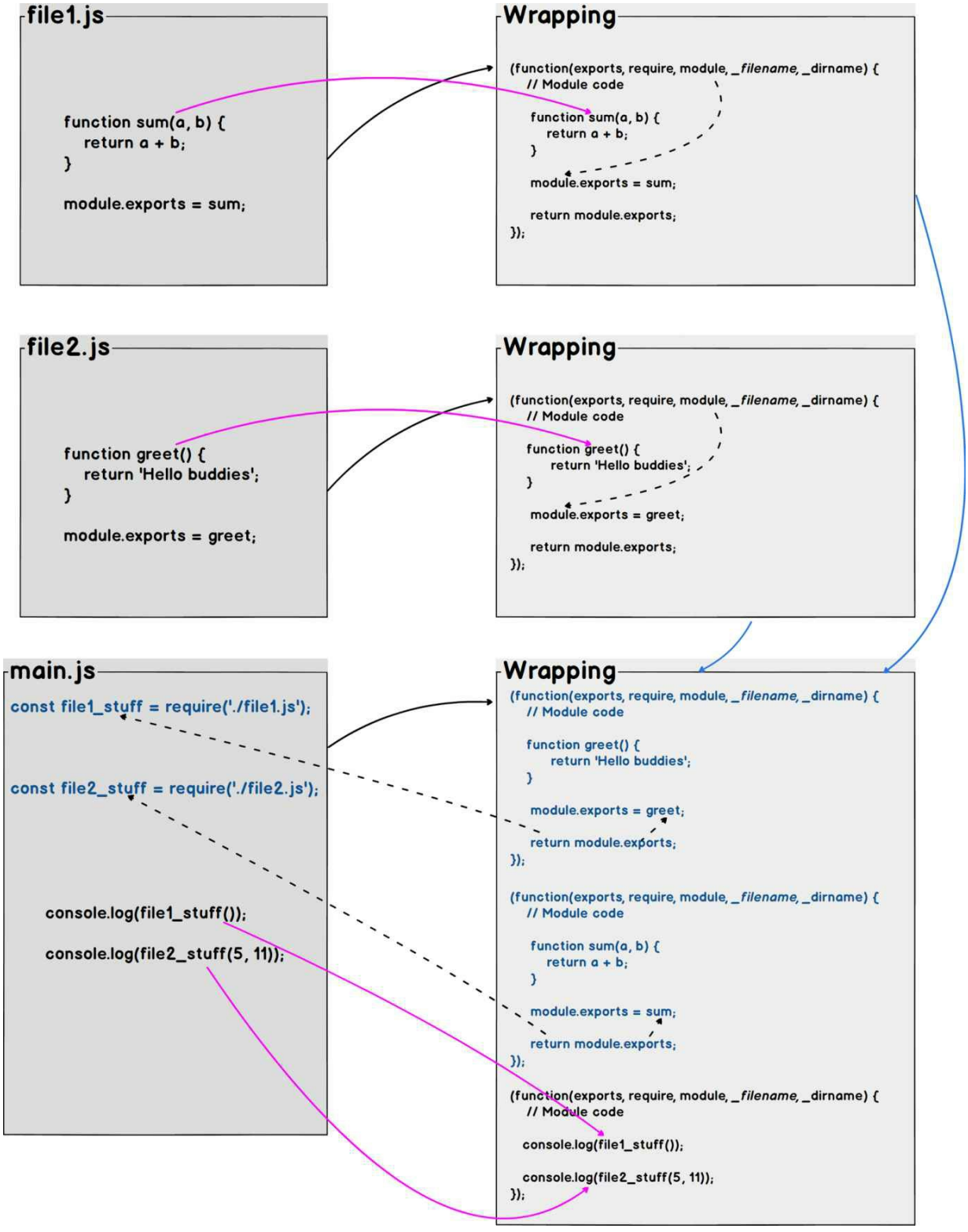
- As we can see from the diagram wrapper of file1.js returns the module.exports object and we have attached the **greet()** function to the **module.exports object**.
- Now, let’s see what happens in our main.js where we required this module(file1.js).



As per visualization, we can see if we extract whatever is returned from IFEE then we can get access to **greet** function via **“module.exports” because “module.exports” is what is returned from that IFEE.** Here we are returning the stuff from IFEE but not stored in a variable such that we can use it.



- Finally, we can see whatever we returned from “module.exports” is actually stored in the variable called “file1_stuff”. Because “module.exports” point to greet function, **“file1_stuff”** will point to greet function😁.
- There you have my friends how **commonjs module system** works under the hood!
- Same way let’s just finish our visualization for file2.js as well😊.


file1.js
function sum(a, b) {
return a + b;
}
module.exports = sum;
Wrapping
(function(exports, require, module, _filename, _dirname) {
// Module code
function sum(a, b) {
return a + b;
}
module.exports = sum;
return module.exports;
});
file2.js
function greet() {
return 'Hello buddies';
}
module.exports = greet;
Wrapping
(function(exports, require, module, _filename, _dirname) {
// Module code
function greet() {
return 'Hello buddies';
}
module.exports = greet;
return module.exports;
});
main.js
const file1_stuff = require('./file1.js');
const file2_stuff = require('./file2.js');
console.log(file1_stuff());
console.log(file2_stuff(5, 11));
Wrapping
(function(exports, require, module, _filename, _dirname) {
// Module code
function greet() {
return 'Hello buddies';
}
module.exports = greet;
return module.exports;
});
(function(exports, require, module, _filename, _dirname) {
// Module code
function sum(a, b) {
return a + b;
}
module.exports = sum;
return module.exports;
});
(function(exports, require, module, _filename, _dirname) {
// Module code
console.log(file1_stuff());
console.log(file2_stuff(5, 11));
});

```
C:\Users\meetp\OneDrive\Desktop\Node-Cpp-Binding>node main.js
Hello buddies!
16
```

- We did exact same for file2.js as well and visualized. You can see whatever we exported from the file can be renamed as your wish as well.
- The last before I close the chapter let me tell you one thing **“exports”** (first argument of IFEE) and **“module.exports”** point to the same place 😊.
- **exports = module.exports.**
- So, you can you **exports** keyword directly instead of writing **module. exports** every time(just add key-value pair to exports object, don’t re-assign to entirely to new stuff😊).
- **Do this:** exports.add = add, exports.greet = greet;
- **Don’t do this:** exports = null or exports = add or exports = greet(If you do this, then it’ll break the functionality of the default module system’s behavior);

Lastly, the ES6+ module system is a real module system that doesn’t follow the above mechanism and natively parse the content of the file and make it available to us without wrapping.

```
That is all from my end friends, I hope you enjoyed learning as much as I enjoyed
writing this book. I will try to add content to this book in future editions as
well. All future updates will free who has access to the first edition☺.
```

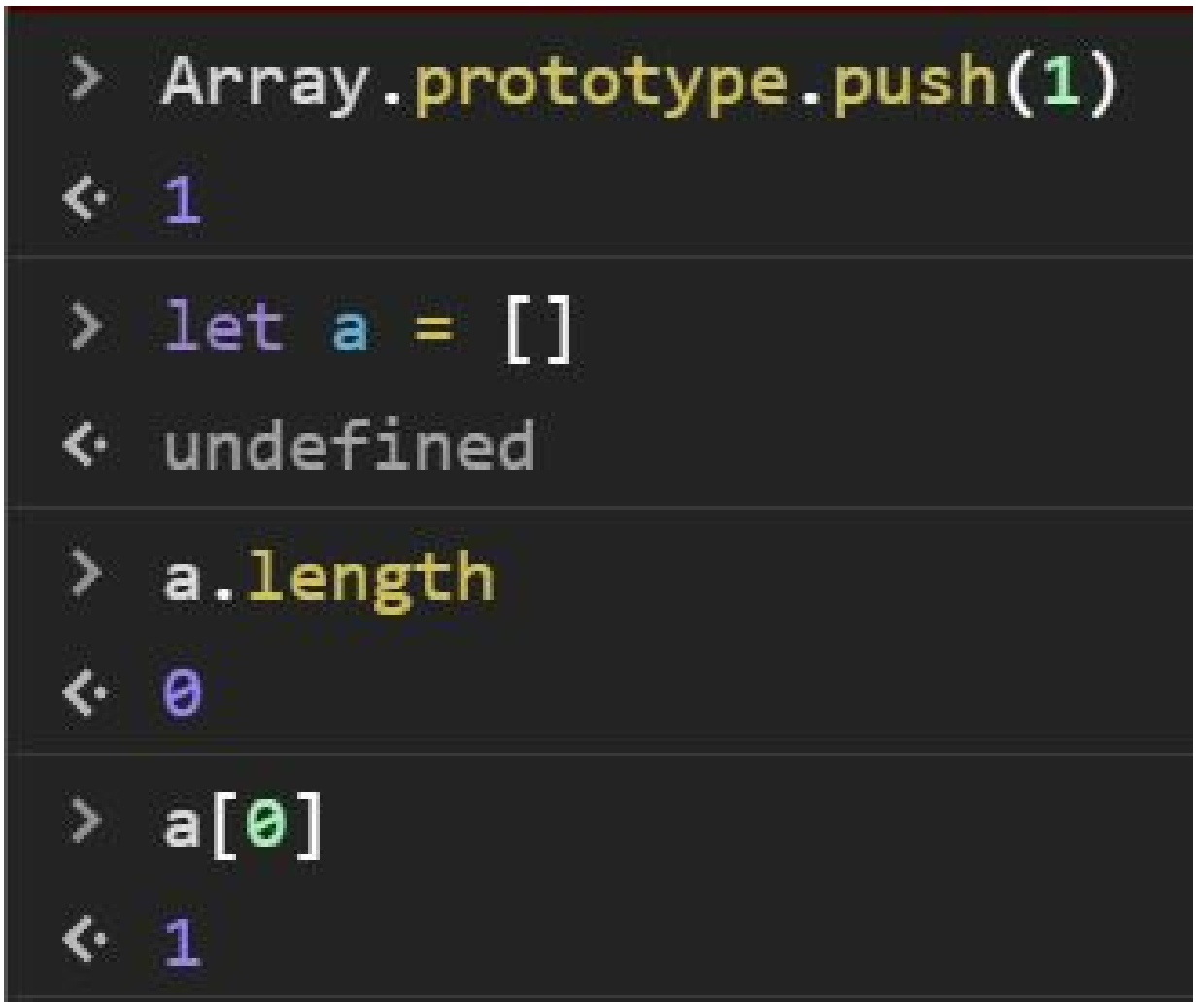
* * *

7

Bonus

```
This chapter is just about some weird javascript facts😂
```

1. typeof **null** is the object which is a bug in javascript.
2. typeof **NaN**(Not a Number) is actually **‘number’.**
3. 1 + 2 == 3 but 1.0 + 2.0 !== 3.0.
4. NaN === NaN; -> false
5. **10/0** -> Infinity; **10/-0** -> -Infinity
6. **let newFn = new Function(‘num’, ‘return num * num’);** // create function dynamically😑.
7. JavaScript is a compiled language, that parses, optimizes, and compiles code ahead of time and **it only interprets code at runtime**🙂.
8. JavaScript is not Java(Always remember this).
9. **It is a very easy language to learn. But very hard to master.**
10. Below the **‘a’** array has an element in it, but the length is still showing **0** 😂.



This is all my friends, I will keep updating this book with more content, till then enjoy learning and we will meet someday, sometime with some other book journey. Thank you for reading and happy Learning.