Execution Context

Outline

Some Terminologies

- **Parser:** A Parser or Syntax Parser is a program that reads your code line-by-line. It understands how the code fits the syntax defined by the Programming Language and what it (the code) is expected to do.

- **JavaScript Engine:** A JavaScript engine is simply a computer program that receives JavaScript source code and compiles it to the binary instructions (machine code) that a CPU can understand.

  - JavaScript engines are typically developed by web browser vendors, and each major browser has one. Examples include the **V8 engine** for Google chrome, **SpiderMonkey** for Firefox, and **Chakra** for Internet Explorer.

Execution Context (EC)

- Whenever JavaScript engine encounters JS code, it creates a special environment to handle the **transformation** and **execution** of this JavaScript code. This environment is known as the Execution Context.
- The Execution Context contains the code that's currently running, and everything that aids in its execution.

Execution Context (EC)

- During the Execution Context run-time,
  - the specific code gets parsed by a parser,
  - the variables and functions are stored in memory,
  - executable byte-code gets generated,
  - and the code gets executed.

- There are two kinds of Eexcution Context in JS
  - Global EC
  - Function EC

Global Execution Context (GEC)

- Whenever the JavaScript engine receives a script file, it first creates a default Execution Context known as the Global Execution Context (GEC).
- The GEC is the base/default Execution Context where all JavaScript code that is not inside of a function gets executed.
- For every JavaScript file, there can only be one GEC.

Function Execution Context (FEC)

- Whenever a function is called, the JavaScript engine creates a different type of Execution Context known as a Function Execution Context (FEC) within the GEC to evaluate and execute the code within that function.
- Since every function call gets its own FEC, there can be more than one FEC at the run-time of a script.

Execution Context Creation

- The creation of an Execution Context (GEC or FEC) happens in two phases:
  - Creation Phase
  - Execution Phase

Creation Phase

- In the creation phase, the Execution Context is first associated with an Execution Context Object (ECO).

- The Execution Context Object stores a lot of important data which the code in the Execution Context uses during its run-time.

- The creation phase occurs in 3 stages, during which the properties of the Execution Context Object are defined and set. These stages are:

  1. Creation of the Variable Object (VO)
  2. Creation of the Scope Chain
  3. Setting the value of the this keyword

Stage 1 : Creation of the VO

- The Variable Object (VO) is an object-like container created within an Execution Context.

- It stores the variables and function declarations defined within that Execution Context.

- In the GEC, for each variable declared with the **var** keyword, a property is added to VO that points to that variable and is set to '**undefined**'.

- Also, for every function declaration, a property is added to the VO, pointing to that function, and that property is stored in memory.

Stage 1 : Creation of the VO

- This means that all the function declarations will be stored and made accessible inside the VO, even before the code starts running.
- The FEC, on the other hand, does not construct a global VO. Rather, it generates an **array-like object** called the '**argument**' object, which includes all of the arguments supplied to the function.
- This process of storing variables and function declaration in memory prior to the execution of the code is known as **Hoisting**.

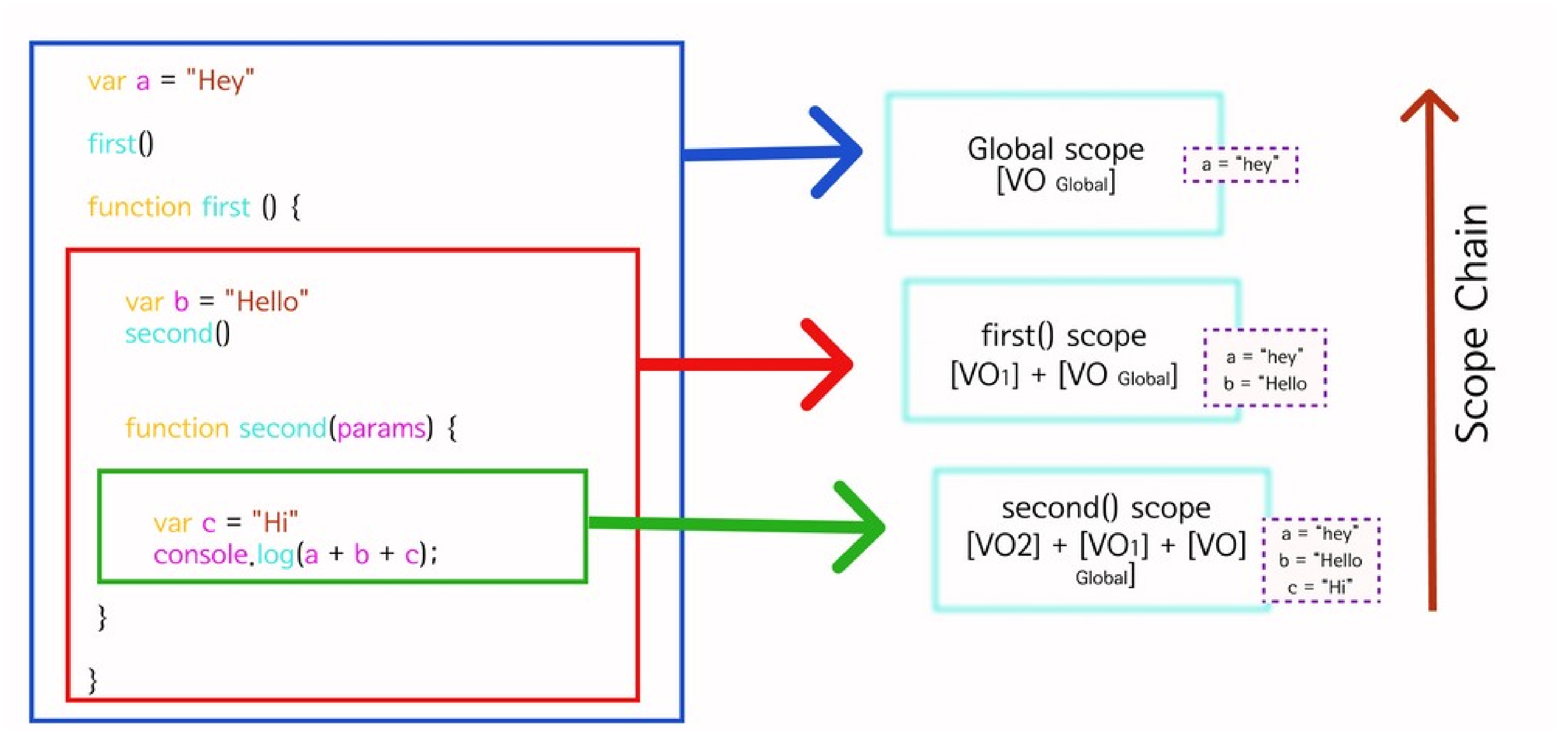
Ground Rules for Hoisting

- In JavaScript, variables declared with **var** keyword are hoisted and variables declared with **let** or **const** keywords are not hoisted.
- Hoisting only works for function **declarations** (statements), not for function **expressions**.

Stage 2 : Creation of the Scope Chain

- Scope in JavaScript is a mechanism that determines how accessible a piece of code is to other parts of the codebase.
- Each Function Execution Context creates its scope: the space/environment where the variables and functions it defined can be accessed via a process called Scoping.
- When a function is defined in another function, the inner function has access to the code defined in that of the outer function, and that of its parents. This behavior is called lexical scoping.
- However, the outer function does not have access to the code within the inner function.

Stage 2 : Creation of the Scope Chain



```
var a = "Hey"

first()

function first () {

    var b = "Hello"
    second()


    function second(params) {

        var c = "Hi"
        console.log(a + b + c);
    }

}
```

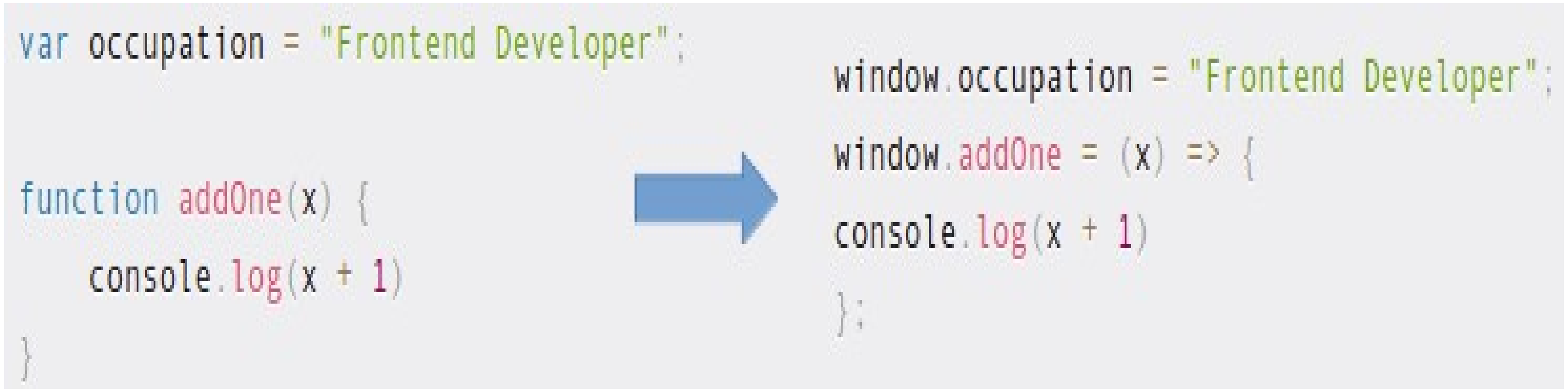
- This idea of the JavaScript engine traversing up the scopes of the execution contexts that a function is defined in, in order to resolve variables and functions invoked in them is called the **scope chain**.

Stage 3 : Setting The Value of The "this" Keyword

- The next and final stage after scoping in the creation phase of an Execution Context is setting the value of the this keyword.
- The JavaScript this keyword refers to the scope where an Execution Context belongs.
- Once the scope chain is created, the value of 'this' is initialized by the JS engine.

Stage 3 : "this" in the GEC

- In the GEC (outside of any function and object), this refers to the global object — which is the window object.
- Thus, function declarations and variables initialized with the var keyword get assigned as methods and properties to the global object – window object.



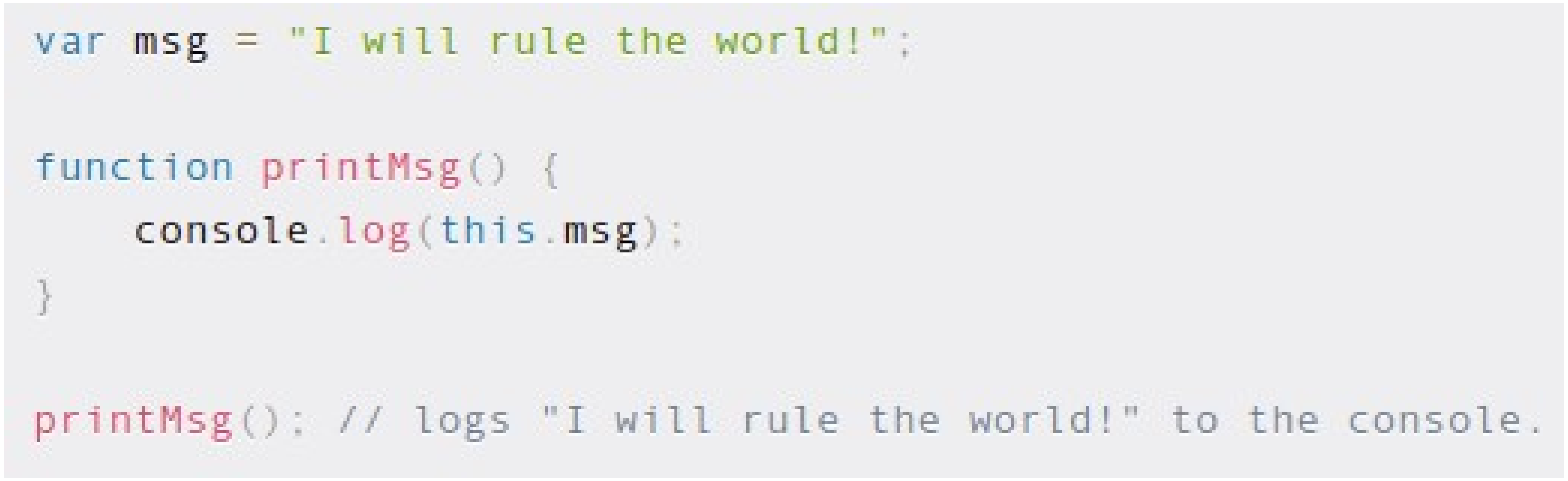
```
var occupation = "Frontend Developer";

function addOne(x) {
    console.log(x + 1)
}
```

```
window.occupation = "Frontend Developer";
window.addOne = (x) => {
console.log(x + 1)
};
```

Stage 3 : "this" in the FEC

- In the case of the FEC, it doesn't create the **“this”** object. Rather, it get's access to that of the environment it is defined in.

- Here that'll be the window object, as the function is defined in the GEC:

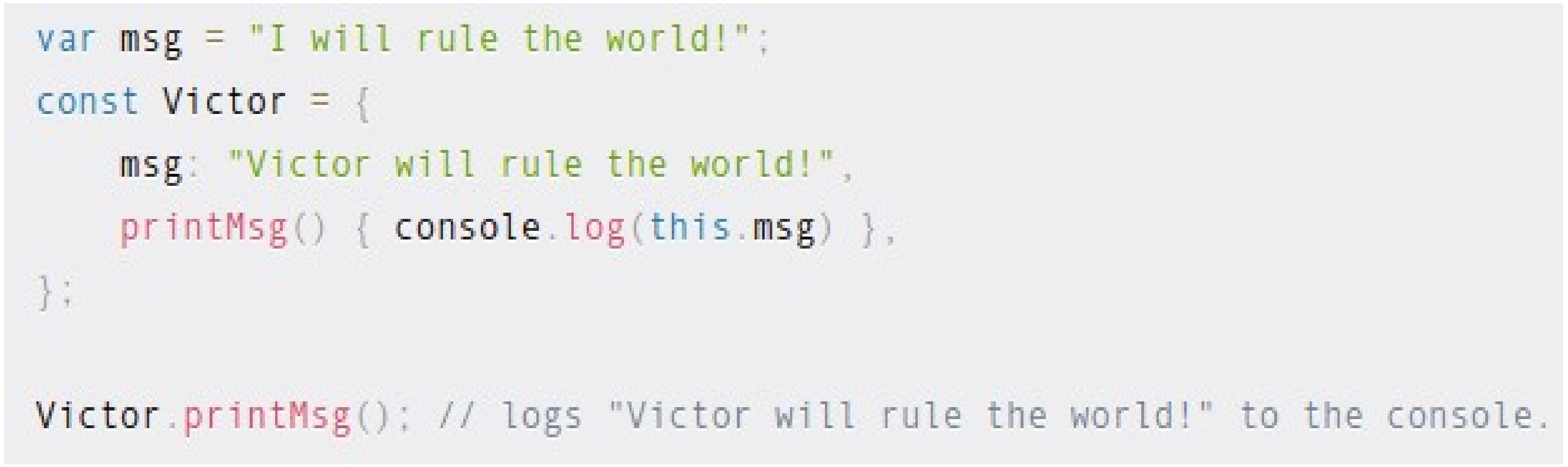


```
var msg = "I will rule the world!";

function printMsg() {
    console.log(this.msg);
}

printMsg(); // logs "I will rule the world!" to the console.
```

"this" in Objects

- In objects, the “**this**” keyword doesn't point to the GEC, but to the object itself. Referencing this within an object will be the same as:

theObject.thePropertyOrMethodDefinedInIt;



```
var msg = "I will rule the world!";
const Victor = {
    msg: "Victor will rule the world!",
    printMsg() { console.log(this.msg) },
};

Victor.printMsg(); // logs "Victor will rule the world!" to the console.
```

Execution Phase

- This is the stage where the actual code execution begins.
- JS again runs through the code line by line.
- Now code is being executed, here all the calculations and evaluation will take place.

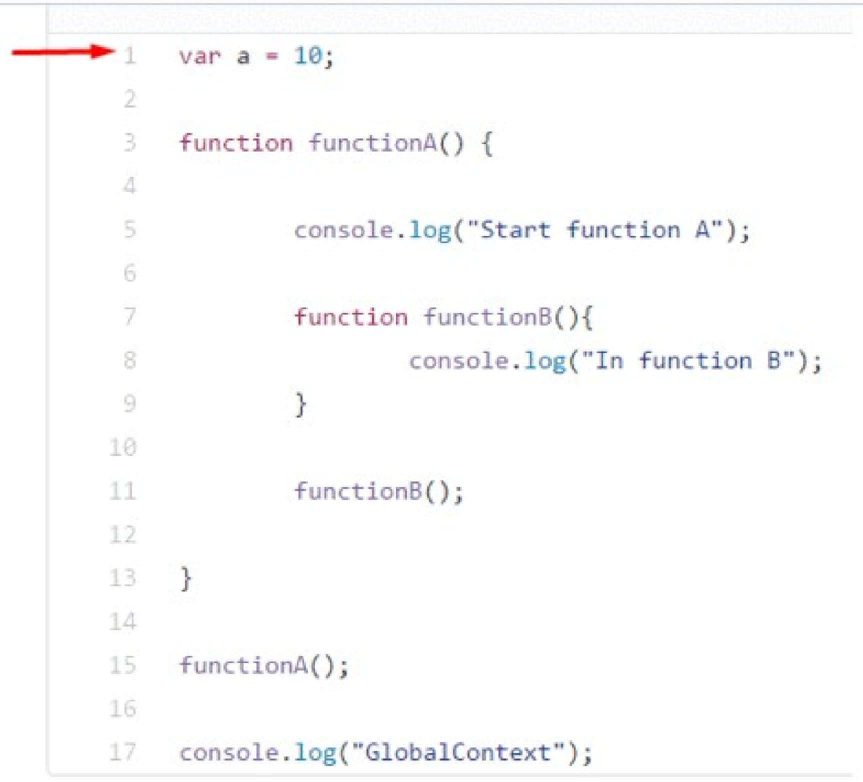
Execution Stack

- The Execution Stack, also known as the Call Stack, keeps track of all the Execution Contexts created during the life cycle of a script.
- Due to the single-threaded nature of JavaScript, a stack of piled-up execution contexts to be executed is created, known as the Execution Stack.
- When scripts load in the browser, the Global context is created as the default context where the JS engine starts executing code and is placed at the bottom of the execution stack.

Execution Stack

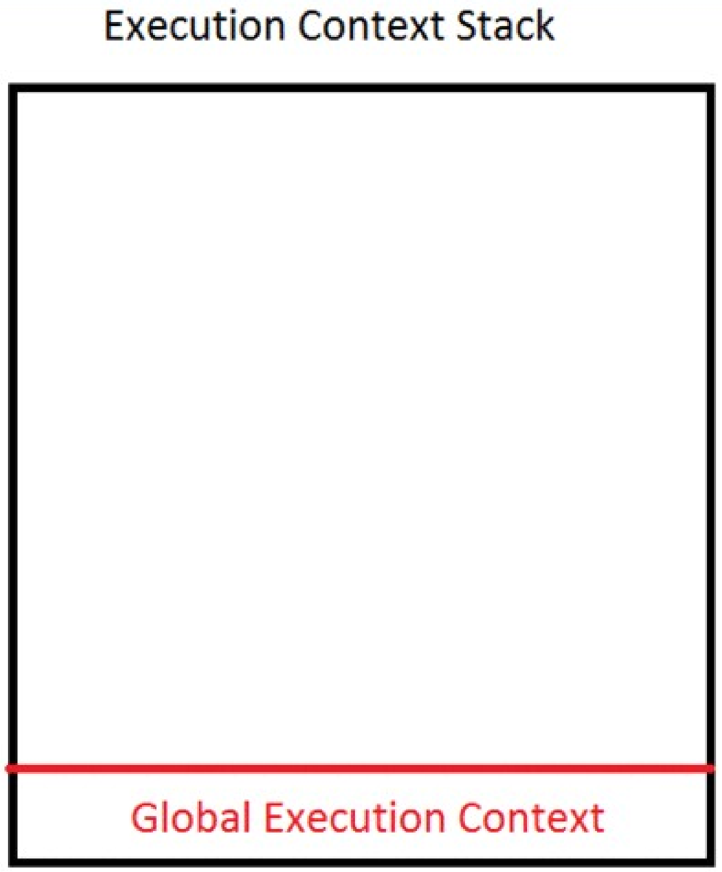
- For each function call, a new FEC is created for that function and is placed on top of the currently executing Execution Context.
- The Execution Context at the top of the Execution stack becomes the active Execution Context, and will always get executed first by the JS engine.
- As soon as the execution of all the code within the active Execution Context is done, the JS engine pops out that particular function's Execution Context of the execution stack, moves towards the next below it, and so on.

Execution Context Stack

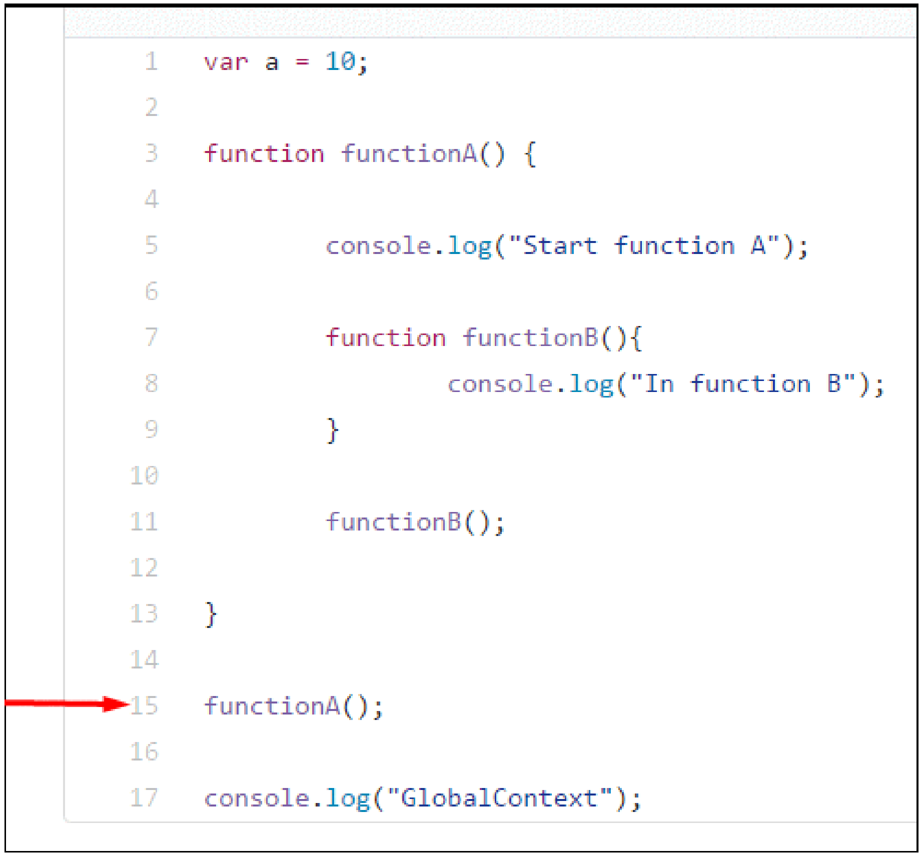


```
var a = 10;

function functionA() {

        console.log("Start function A");

        function functionB(){
                console.log("In function B");
        }

        functionB();

}

functionA();

console.log("GlobalContext");
```

Just before the code execution begins, JS engine pushed global execution context in execution context stack.

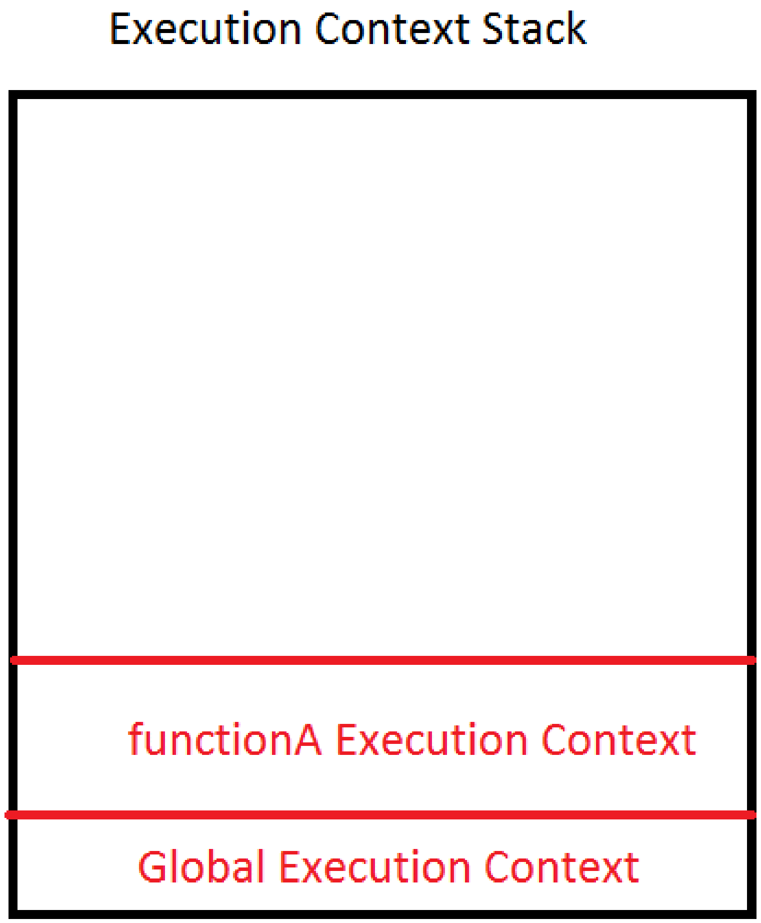


Execution Context Stack

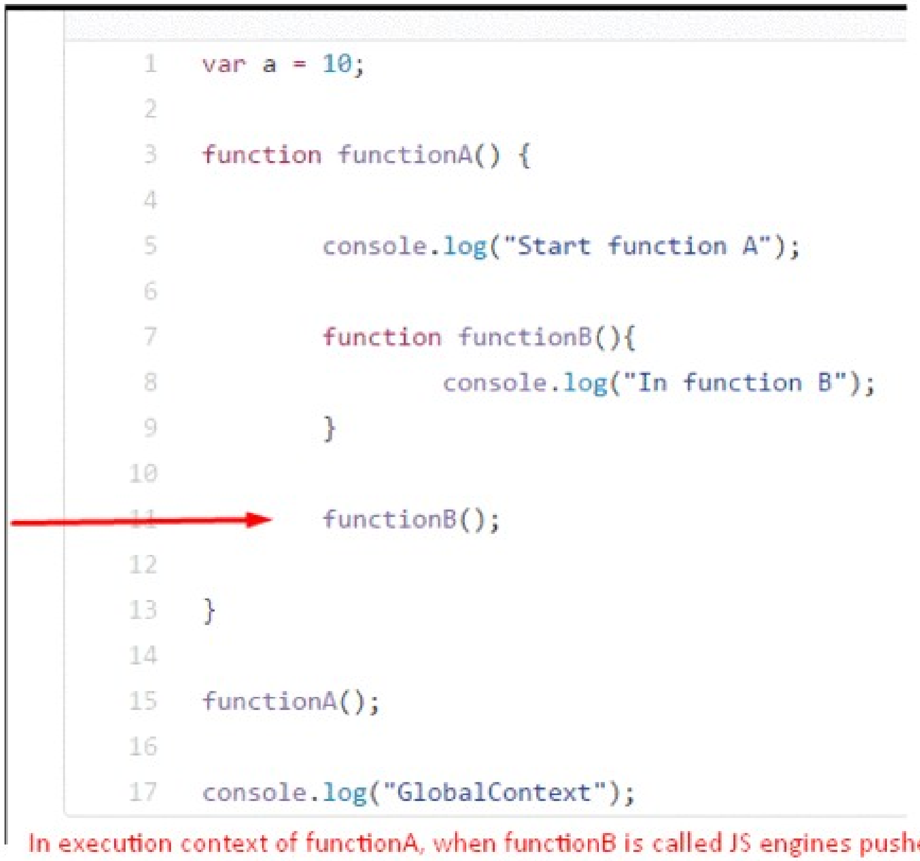


```
var a = 10;

function functionA() {

        console.log("Start function A");

        function functionB(){
                console.log("In function B");
        }

        functionB();

}

functionA();

console.log("GlobalContext");
```

When functionA() is called in GEC, JS engine enters functionA execution context in ECS and start executing functionA

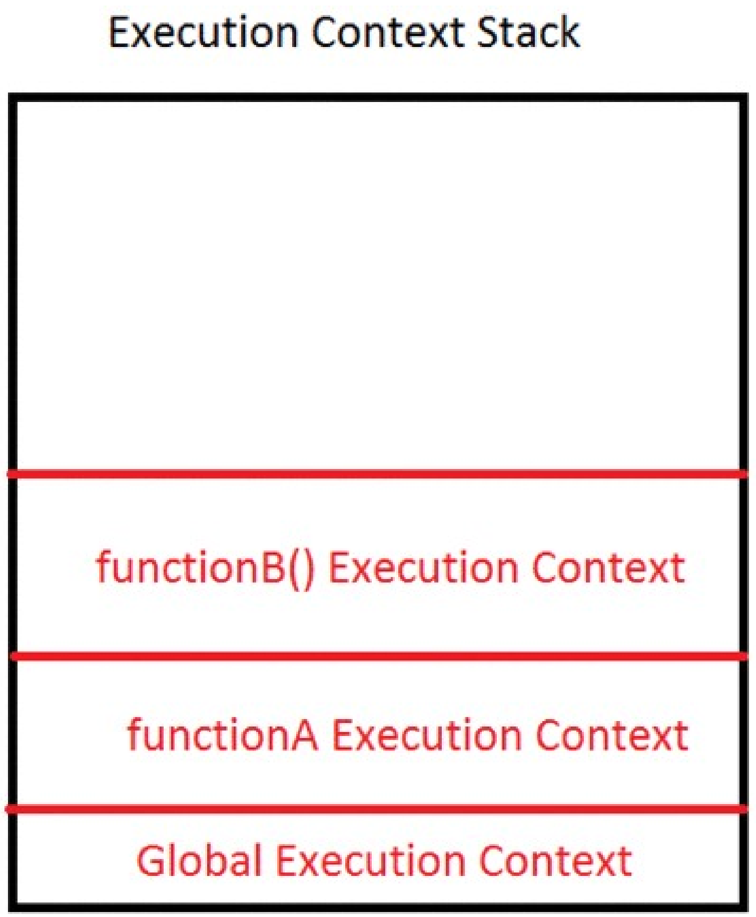


Execution Context Stack

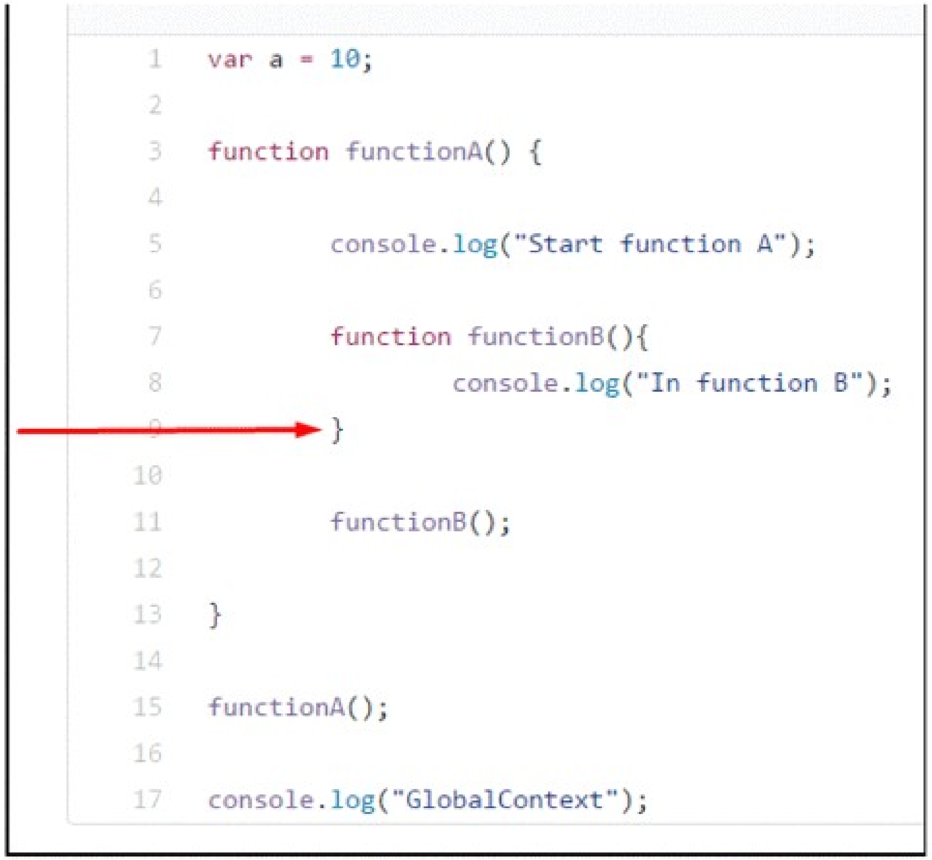


```
var a = 10;

function functionA() {

        console.log("Start function A");

        function functionB(){
                console.log("In function B");
        }

        functionB();

}

functionA();

console.log("GlobalContext");
```

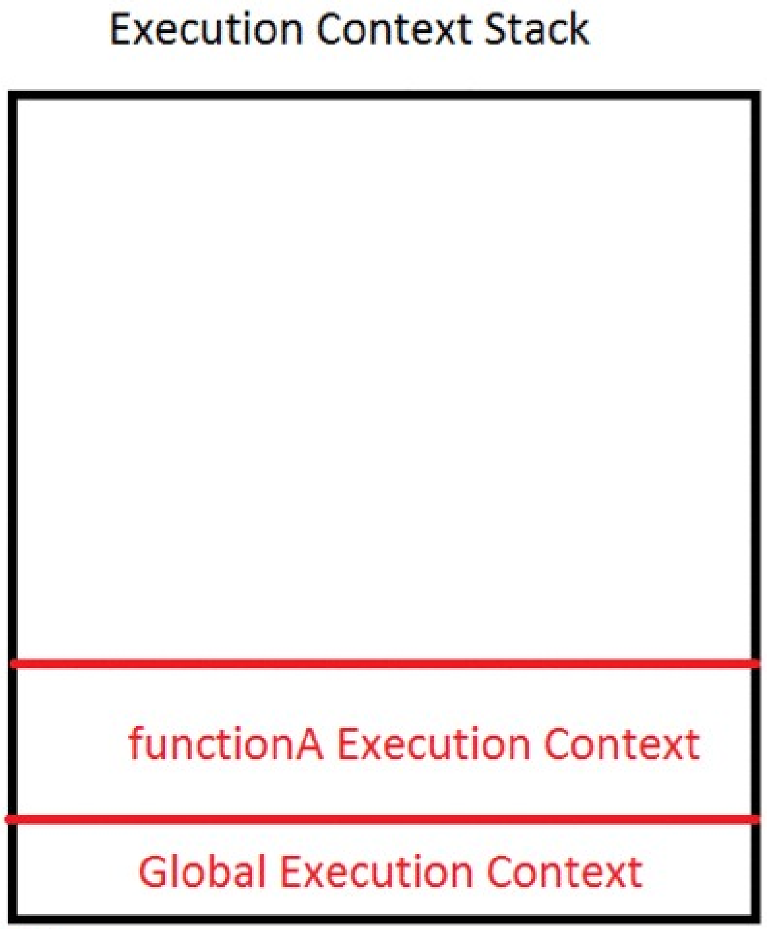
In execution context of functionA, when functionB is called JS engines pushes execution context of fucntionB on top of ECS and start executing functionB



Execution Context Stack

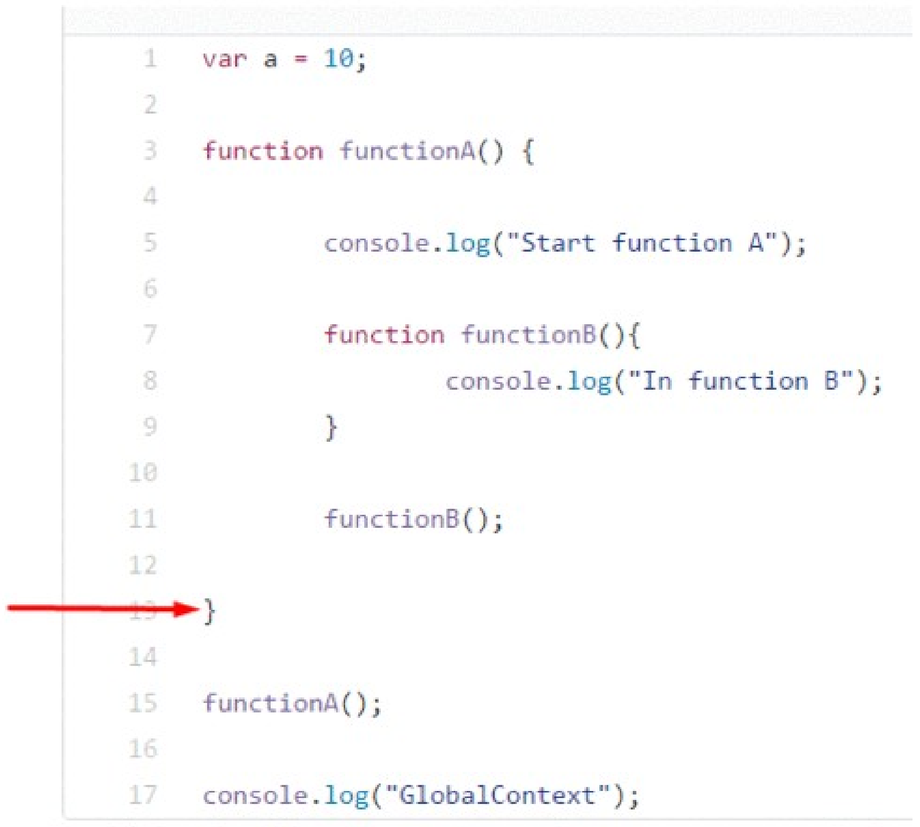


```
var a = 10;

function functionA() {

        console.log("Start function A");

        function functionB(){
                console.log("In function B");
        }

        functionB();

}

functionA();

console.log("GlobalContext");
```



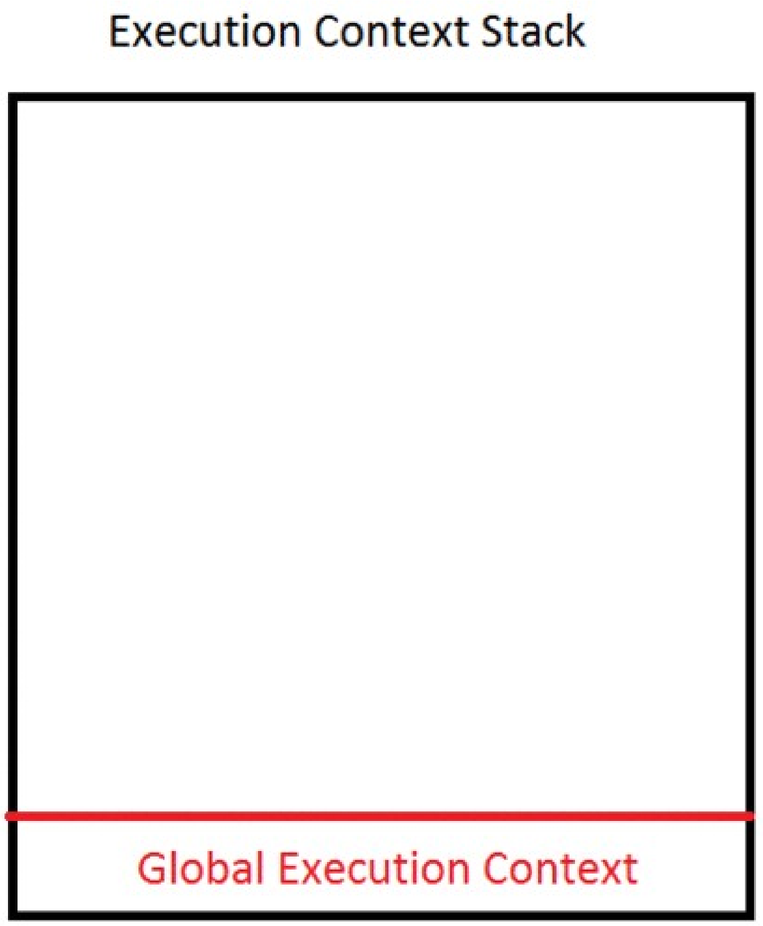
When all the code of functionB gets executed, JS engines pops execution context of functionB and again starts executing functionA as its execution context is on top of stack

Execution Context Stack

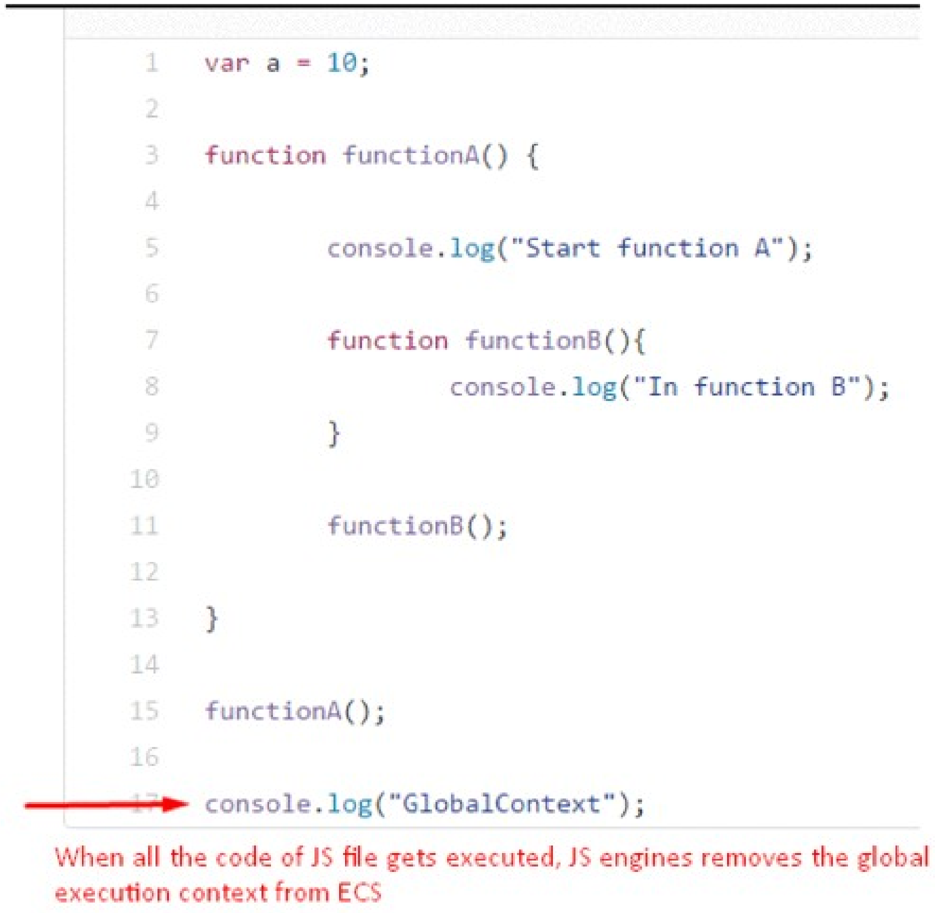


```
var a = 10;

function functionA() {

        console.log("Start function A");

        function functionB(){
                console.log("In function B");
        }

        functionB();

}

functionA();

console.log("GlobalContext");
```

When all the code of functionA gets executed, JS engines pops out the functionA execution context. As after this global execution context is on top of the stack, JS engine starts executing the global code i.e. console.log("GlobalContext")

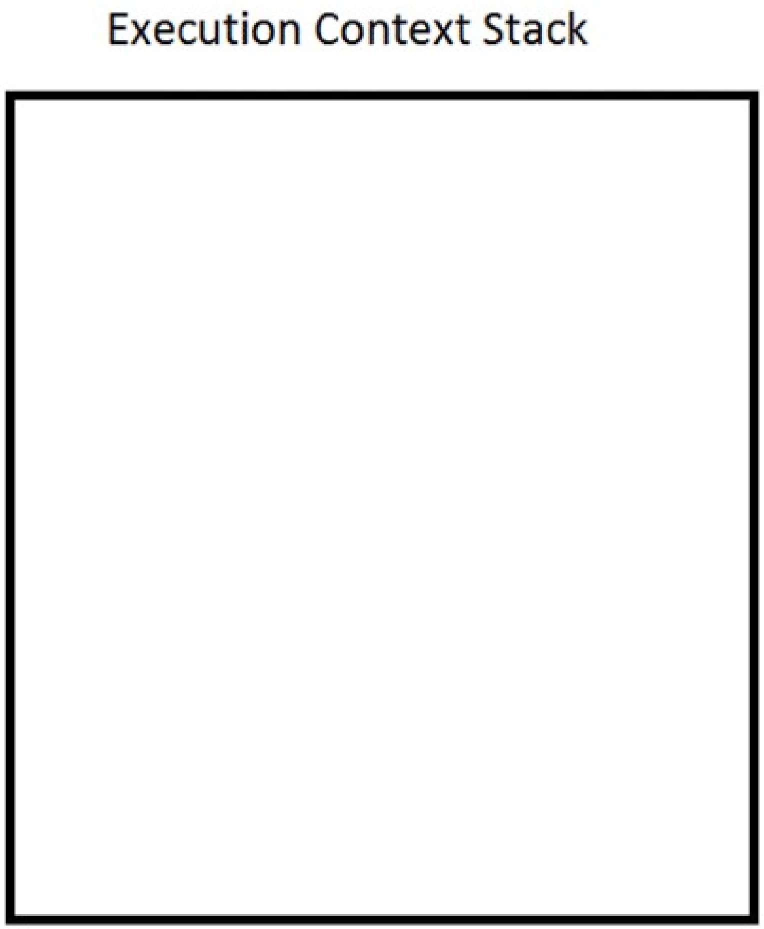


Execution Context Stack



```
var a = 10;

function functionA() {

        console.log("Start function A");

        function functionB(){
                console.log("In function B");
        }

        functionB();

}

functionA();

console.log("GlobalContext");
```

When all the code of JS file gets executed, JS engines removes the global execution context from ECS



References

- https://www.freecodecamp.org/news/execution-context-how-javascript-works-behind-the-scenes
- https://medium.com/@happymishra66/execution-context-in-javascript-319dd72e8e2c