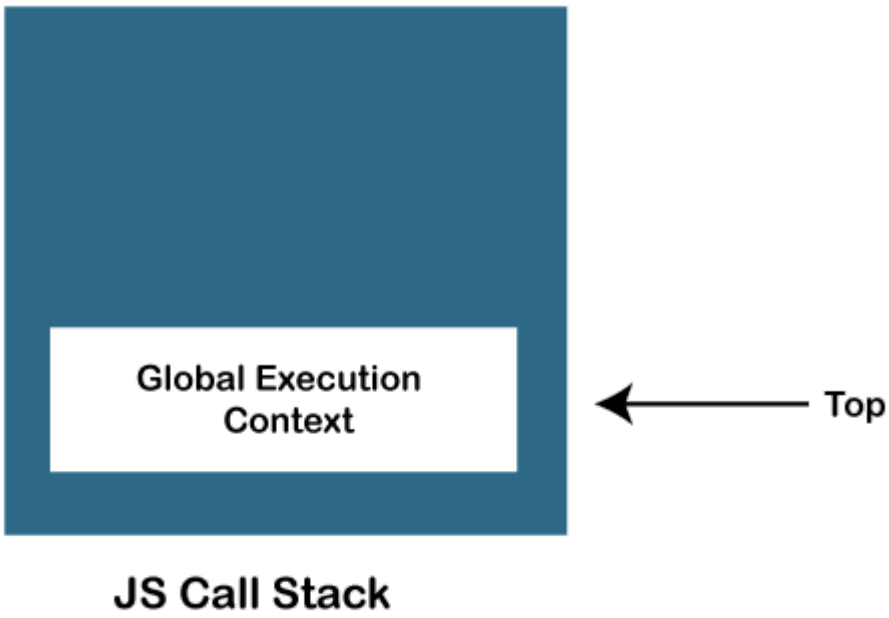
# What is JS Call Stack

The JavaScript execution contexts (Global execution context and function execution context) are executed via the JavaScript engine. In order to manage these execution contexts, the JS engine uses the call stack. So, the JS call stack is a data structure that keeps track information of the functions being called and executed. Thus, if the user invokes a function for execution, the specified function gets pushed/added in the call stack, and when the user returns from a function, it means the function is popped out from the call stack. Thus, call stack is a normal stack data structure that follows the stack order principal, i.e., LIFO (Last In First Out).

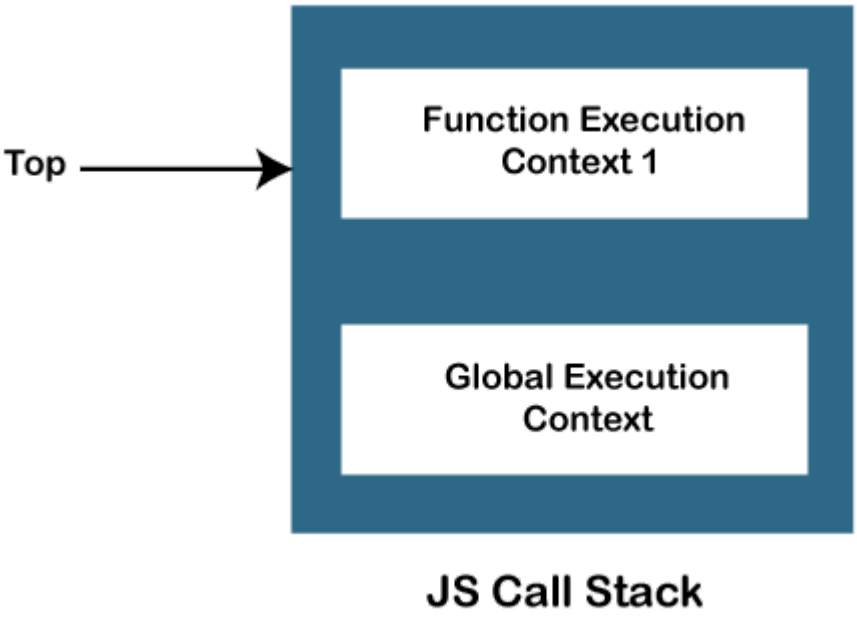
## Role of JavaScript Call Stack

There are the following points where the call stack is being used by the JS engine:

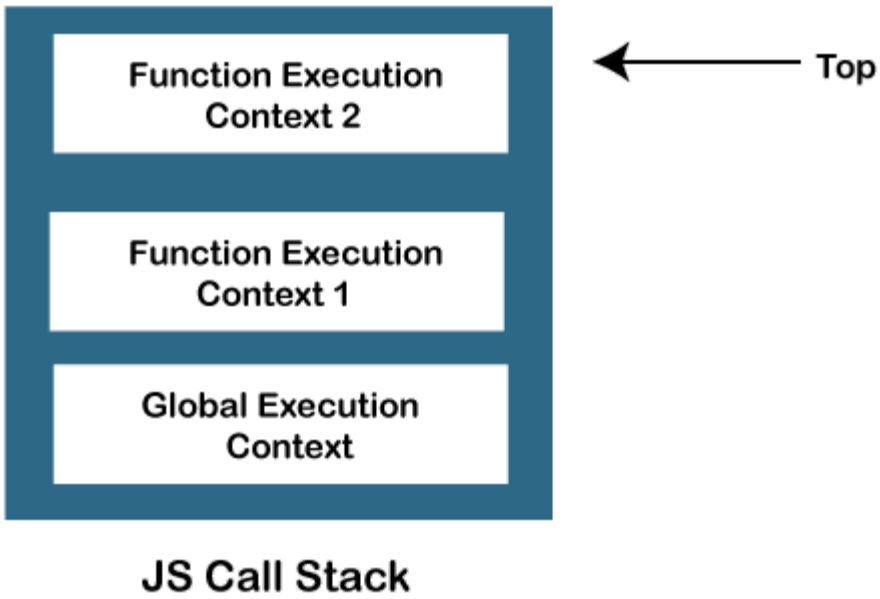
- When any script is executed by the user, the JS engine creates a Global execution context and then adds it on the call stack and at the top of the stack so that it may get executed.



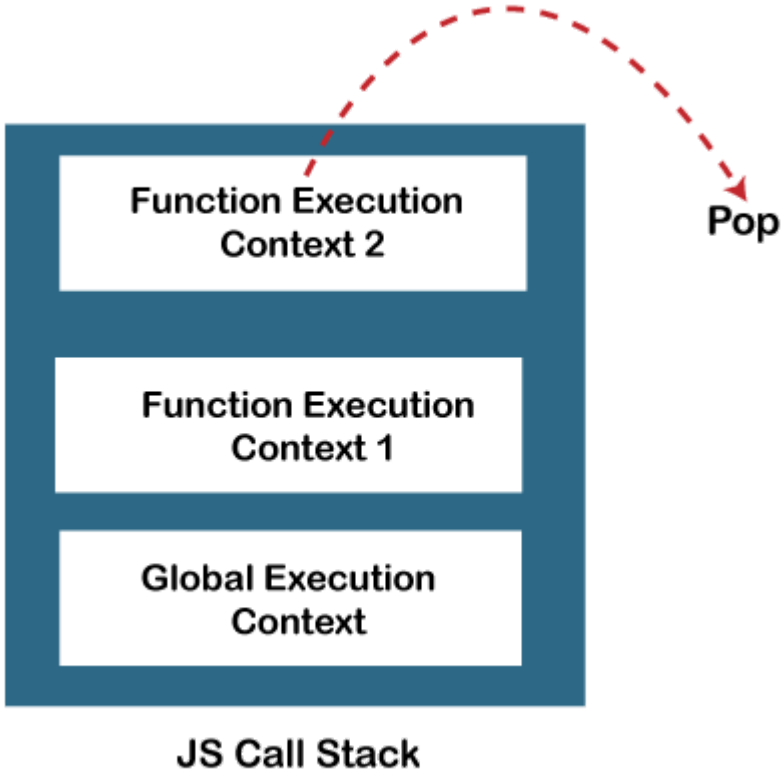

- When any function is invoked, the JS engine creates a Function execution context and adds it on the stack and at the top of the stack so that the invoked function may get executed.




- In case a function invokes another function, the JS engine creates a Function execution context for the invoked function, adds it to the top of the stack, and begins the execution.




- When any function execution gets completed, the JS engine pops it out of the stack and continues the execution of the other functions stored in the stack.




- If no space is left in the stack and we try to push more functions, it throws a "stack overflow" error, and if no further execution context is present in the call stack, it throws a "Stack Underflow" error.

## JavaScript Call Stack Example

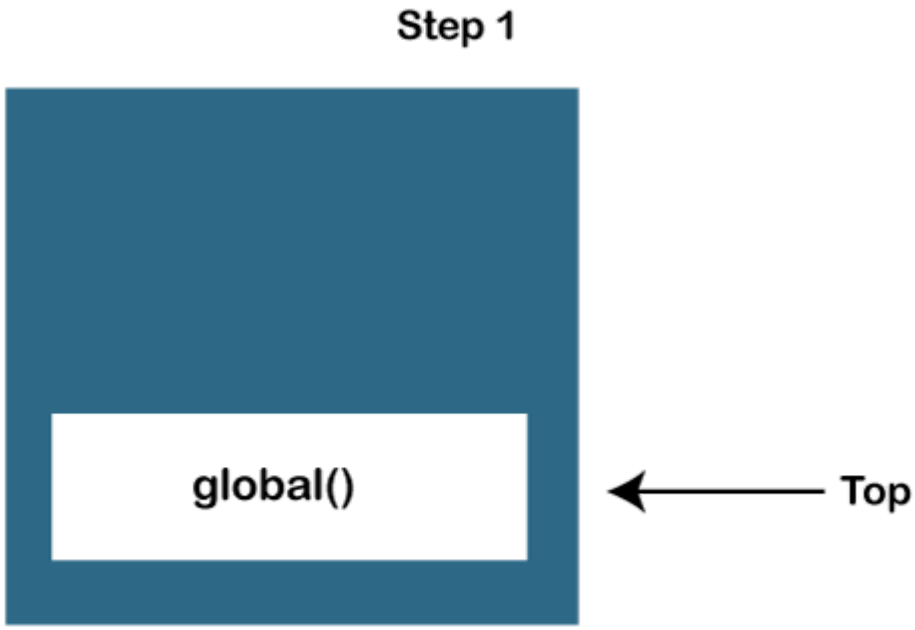
Let's see an example to understand the use of the JavaScript Call Stack function:

```
function getSum(x, y) {
   return x+ y;
}
function findavg(x,y) {
    return getSum(x,y) / 2;
}
let z = findavg(10, 20);
```

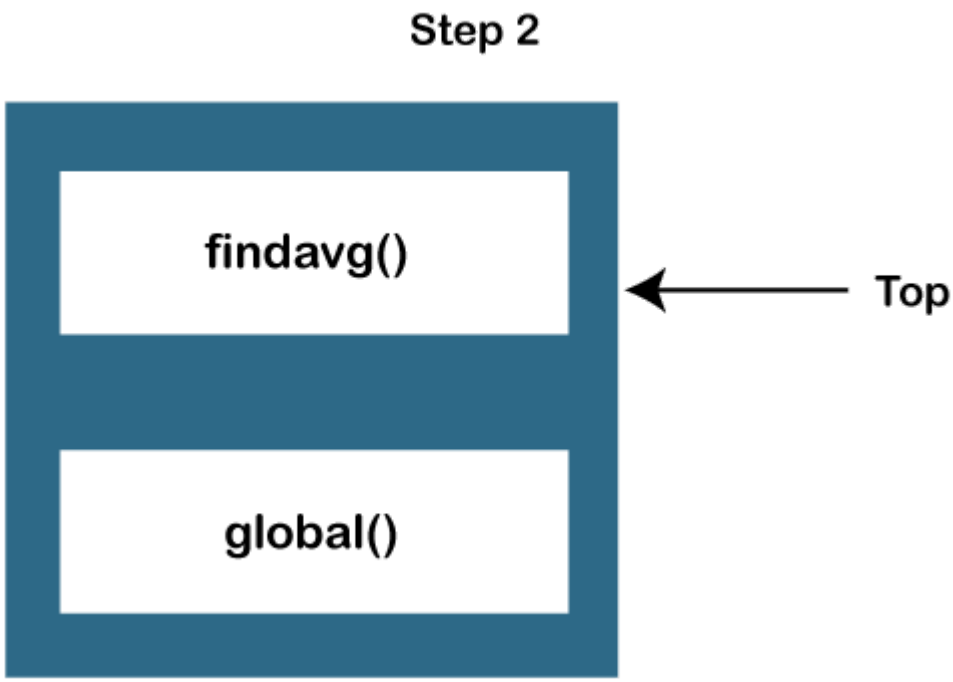
## How the code works

In the above code, we have created two functions, getSum () and findavg (), and the execution of the script begins in the following described steps:

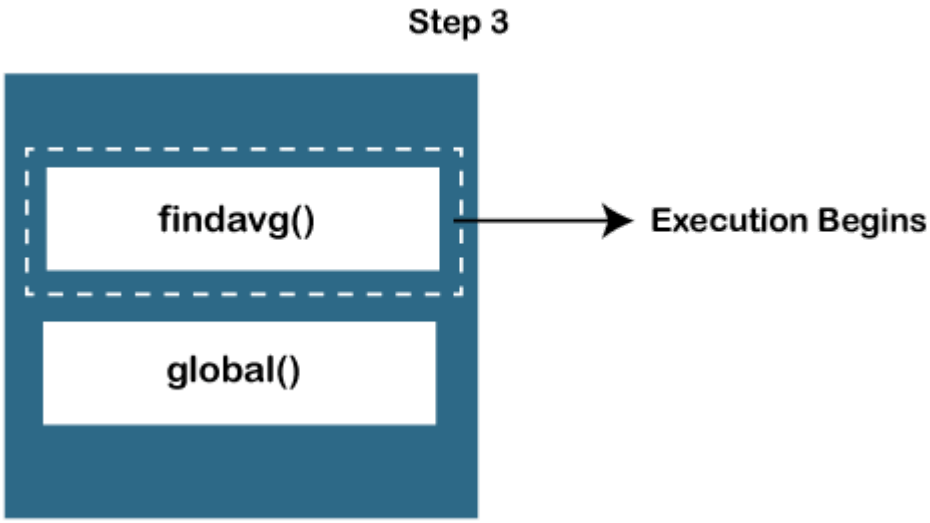
1. When the execution of the script begins, the JS engine initially creates a global execution context (i.e., global () function) and adds it to the top of the call stack.
2. The global execution moves to the execution phase of the life cycle after entering the creation phase, as you can see in the below image:



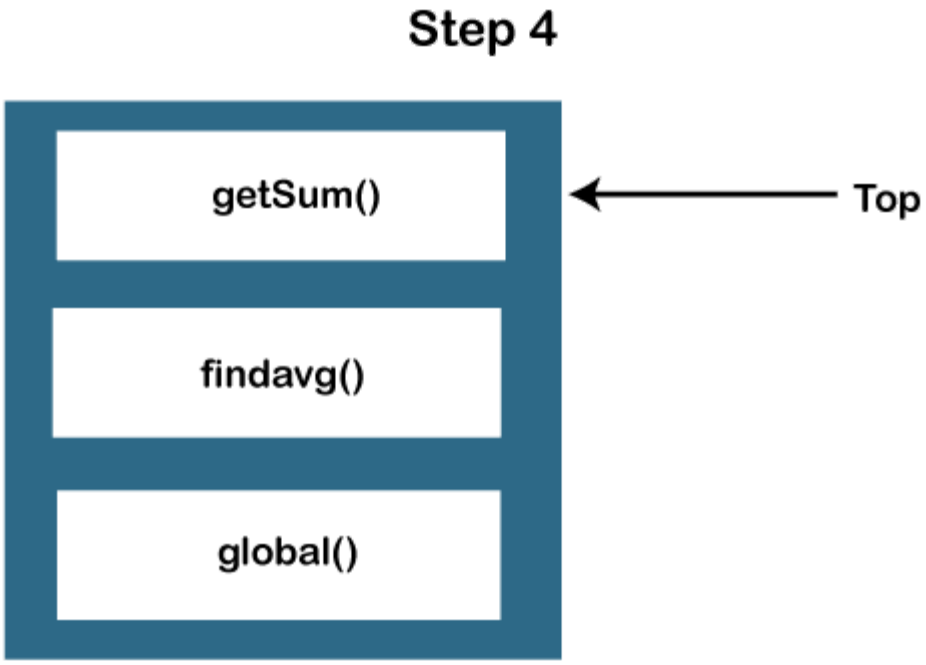

3. The findavg (10, 20) function gets invoked, and so the JS engine creates the function execution context for it. Then push it on the top of the call stack.
4. So, now in the call stack, two functions are pushed, i.e., global () and findavg(), and on the top of the stack, the findavg() function is present, as you can see in the below image



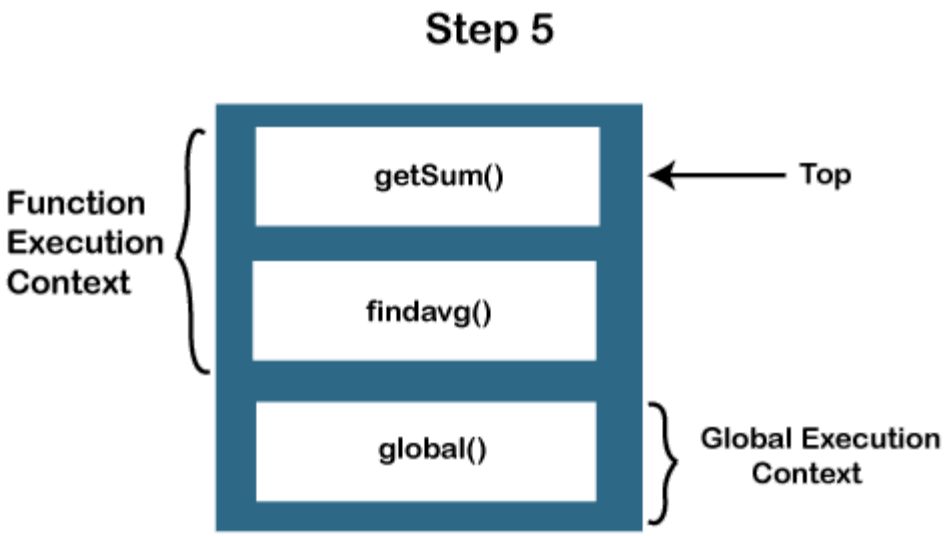
5. The JS engine begins the execution of the findavg () function because it exists on the top of the stack, as you can see in the image:



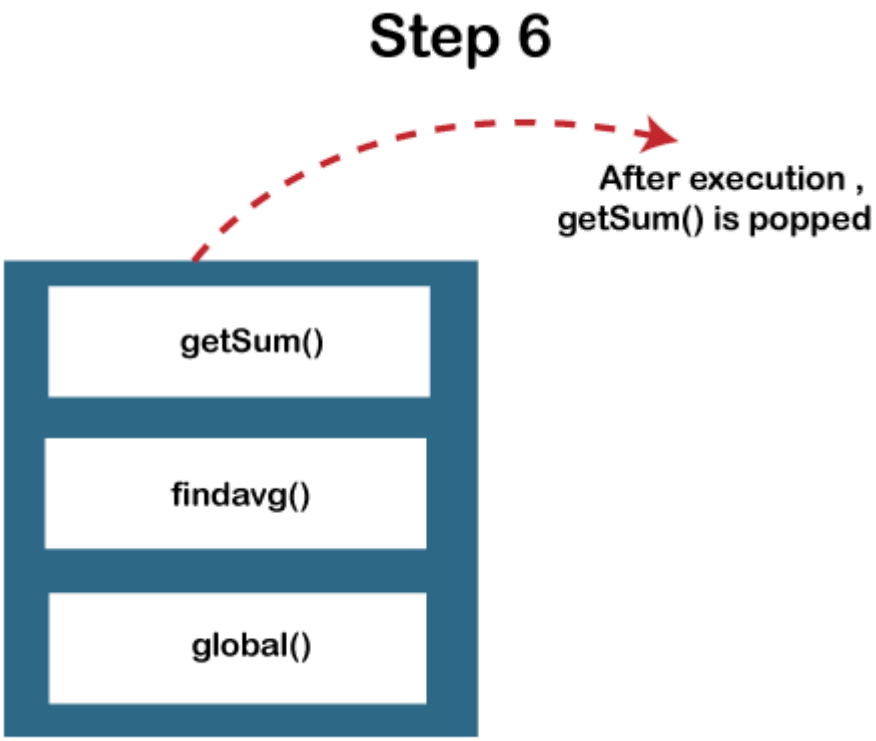

6. As in the code, the getSum () function is invoked inside the findavg () function definition, so the JS engine creates a function execution context for the getSum () function and pushes it on the top of the stack.
7. Now, in the stack, there are three functions present, which are global (), findavg (), and getSum () functions, as you can see in the below image:



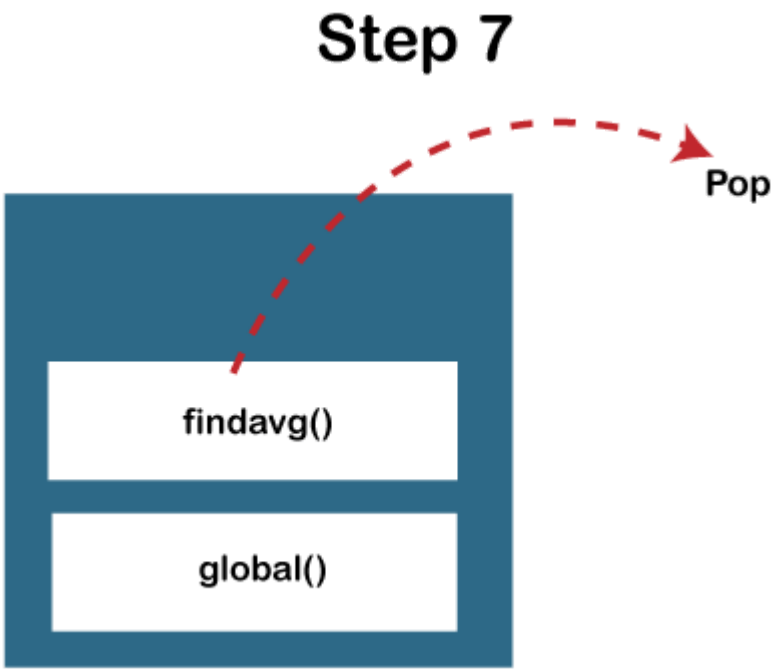

There are two functional execution contexts and a global execution context as you can see below:




8. So, the JS engine executes the getSum () function first and pops it out of the call stack.

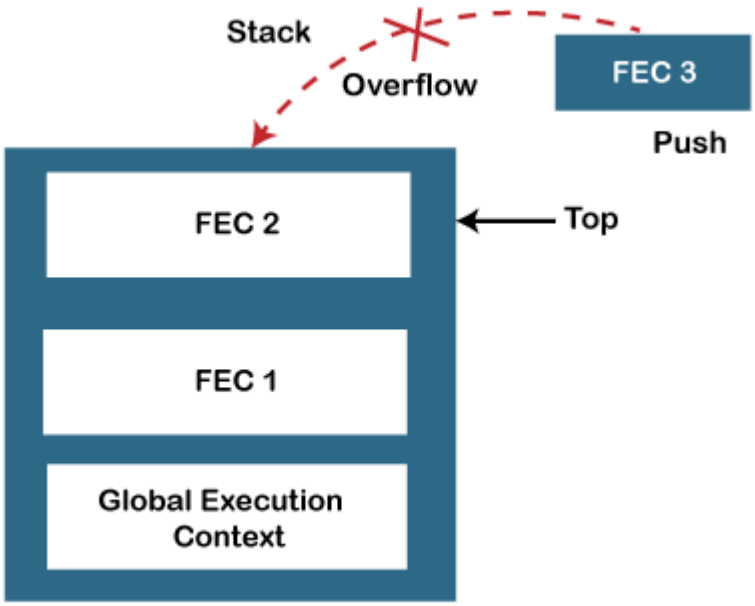



9. Similarly, the findavg () function gets executed and gets out of the call stack.




10. As both executions of the functions are completed, and no other function for execution is left in the call stack. The JS engine stops the execution of the call stack and moves for the other execution tasks.

## When does call stack overflow




The overflow condition occurs when there is no more space left in the call stack, or the condition may occur when there is a recursive function that has no exit point. The JavaScript call stack is defined with a fixed size that depends on the implementation of the host environment (which is either the Node.js or web browser). So, when the

limit of the defined size of the stack is exceeded, then stack overflow occurs. Thus, it throws a stack overflow error.

**Example:**

The below example describes the stack overflow condition:

```
function test(){
    test();
}

test();
```

So, in the above code, we can see that we have invoked the **test ()** function recursively, which means this function will execute until the host environment maximum call size exceeded, and thus the stack throws the stack overflow error.

**Point to be noted:**

JavaScript is a synchronous and single-threaded programming language. It means that when any script gets executed, then the JS engine executes the code line by line, starting from top to bottom. So, the JavaScript engine has only one call stack, and it can do only one thing at a time.