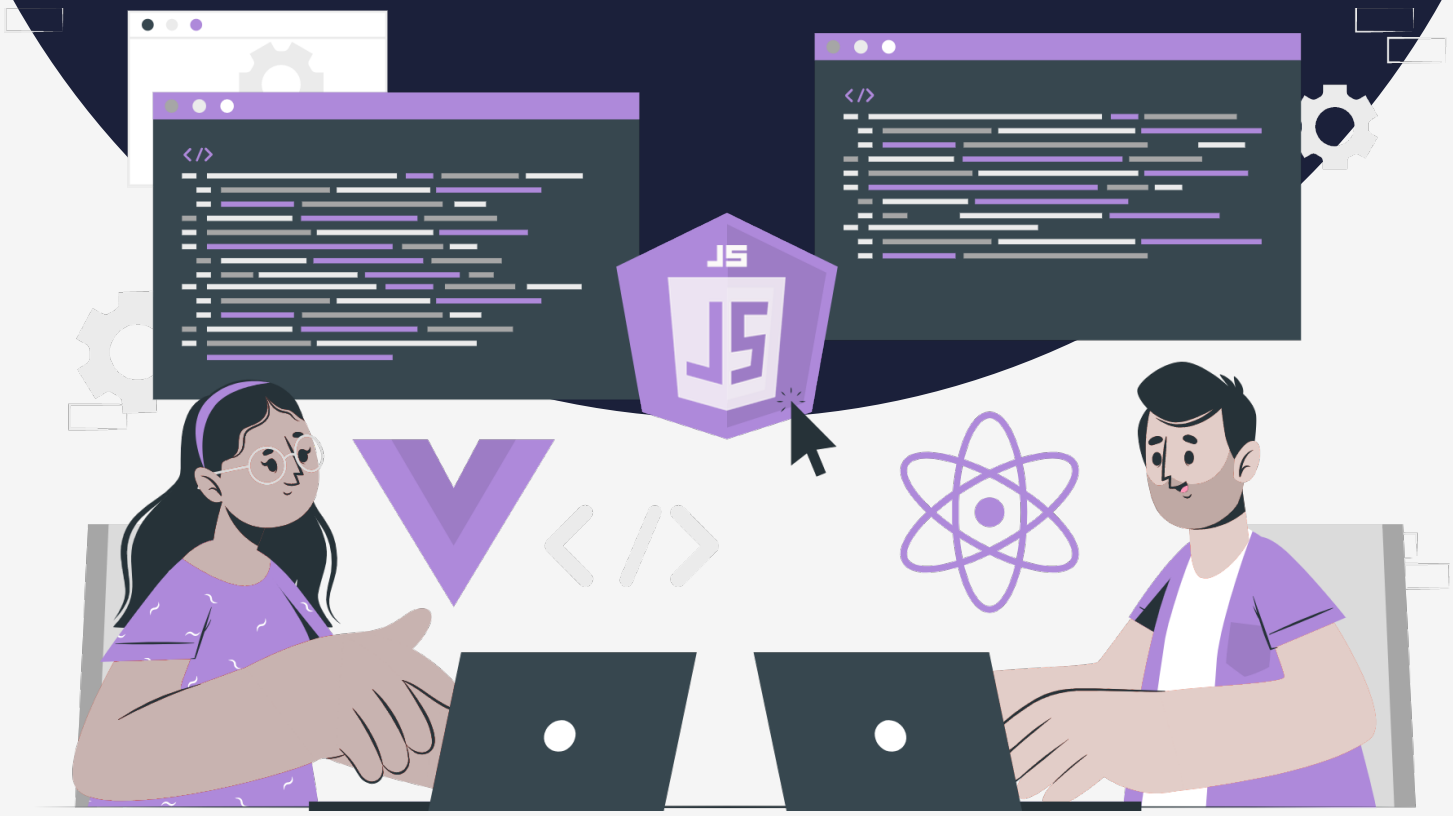
Lesson:

Call Stack



Topics Covered:

1. Introduction.
2. What is call stack?
3. Call stack location.
4. Callstack: How it works?
5. Stack overflow and underflow.
6. Visualising Call Stack.

The call stack is a fundamental concept in JavaScript and is crucial to understand how functions and code execution work. In JavaScript, every time a function is called, it is added to the call stack. When the function is finished executing, it is removed from the stack, and the execution continues from where it left off. A call stack is an essential tool for tracking the order in which functions are called and helps prevent stack overflow errors. Understanding how the call stack works is crucial for any developer to write efficient, bug-free code. We will be looking at call stack in depth in this lecture.

What is call stack?

The call stack in JavaScript is a data structure that helps manage the execution of functions. Whenever a function is called, it is added to the call stack. When the function completes its execution, it is removed from the stack. This process follows the Last-In-First-Out (LIFO) principle, meaning that the last function added to the stack is the first one to be executed and removed from the stack.

The call stack is essential to JavaScript because it ensures the correct sequence of function calls. If a function relies on the result of another function, the call stack ensures that the dependent function is called only after the required function has completed its execution. This process helps prevent errors and ensures that the code is executed efficiently. However, if the call stack becomes too large, it can cause a stack overflow error, which occurs when the stack's memory limit is exceeded. Therefore, it's crucial to keep the call stack size in check to avoid such errors.

Call stack location

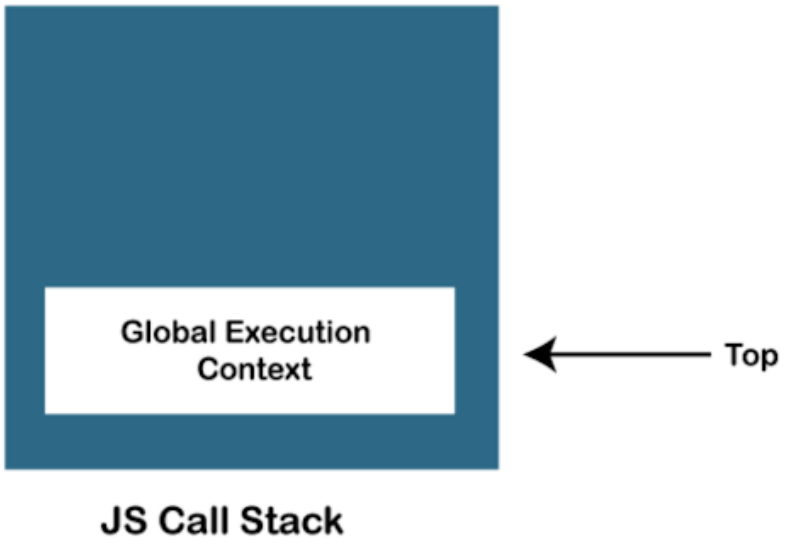
The JavaScript call stack is an internal data structure that is maintained by the JavaScript engine. It is present in the browser's runtime environment or on js runtime like Node.js. The call stack is a part of the JavaScript runtime environment and is not accessible directly.

The call stack is created by the JavaScript engine to keep track of the execution context of each function. Whenever a function is invoked, the engine creates an execution context and adds it to the call stack. The execution context contains information such as the function's arguments, local variables, and the function's scope chain which we have seen in the previous lecture. Once the function completes its execution, the execution context is removed from the call stack.

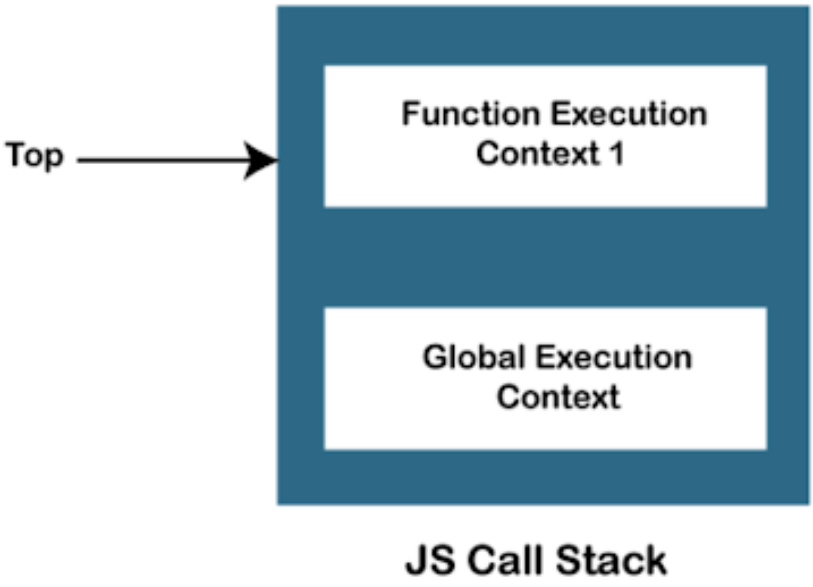
Considering the previous concepts learned, The call stack is an essential part of JavaScript's event-driven, single-threaded model of execution. It ensures that functions are executed in the correct sequence and that the execution of one function does not interfere with the execution of another.

Callstack: How it works?

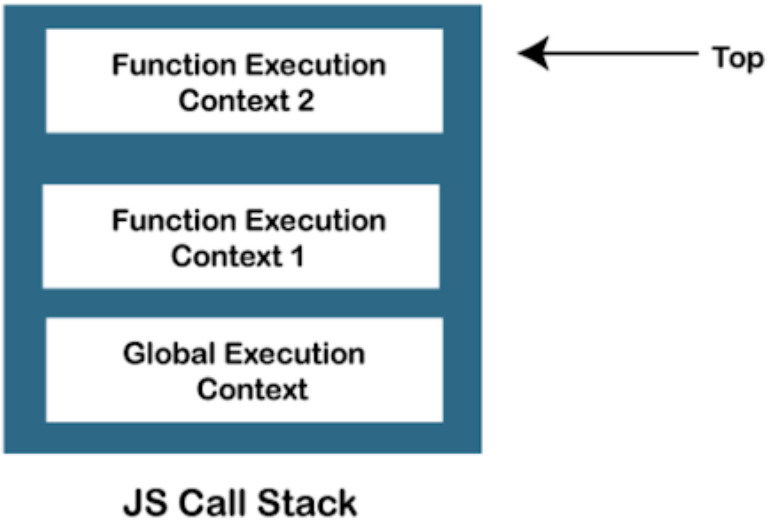
1. Whenever a user executes a script, the JavaScript engine generates a Global execution context and places it at the top of the call stack for execution.



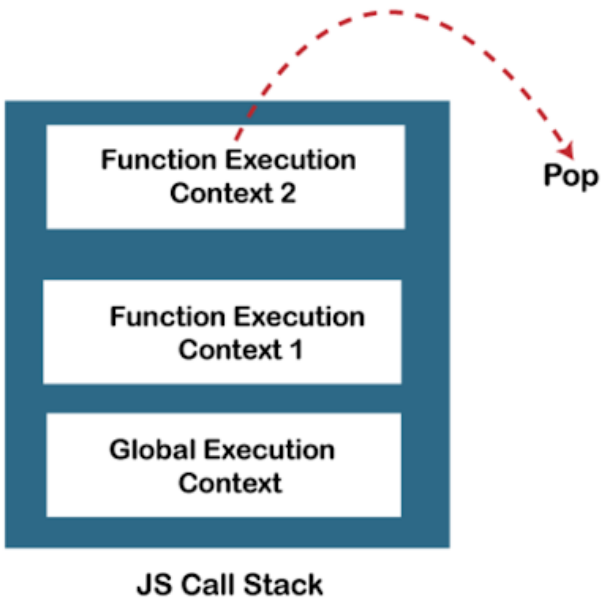
2. Whenever a function is called, the JavaScript engine generates a Function execution context and places it at the top of the call stack.



3. If a function calls another function, the JavaScript engine generates a Function execution context for the called function, adds it to the top of the call stack, and starts executing it.



4. After a function finishes executing, the JavaScript engine removes it from the call stack, allowing the execution of other functions stored in the stack to continue.



Stack overflow and underflow

The call stack in JavaScript has a limited amount of space available, and if this space is exceeded, it can lead to a "stack overflow" error. This error occurs when there are too many nested function calls, and the call stack becomes full, causing it to run out of memory. When this happens, the JavaScript engine cannot add any more function execution contexts to the stack, and the program terminates with an error message.

On the other hand, if there are no more function execution contexts left in the call stack, and we try to pop an execution context, it leads to a "stack underflow" error. This error occurs when the program tries to remove an execution context from an empty call stack. This can happen when we try to pop more functions than what is present in the stack.

Both of these errors can cause the program to crash, and they are difficult to debug since they don't provide much information about the underlying cause. Therefore, it's essential to write code that is efficient and avoids nested function calls to prevent stack overflow errors. It's also important to handle errors gracefully to prevent the program from crashing due to a stack underflow error.

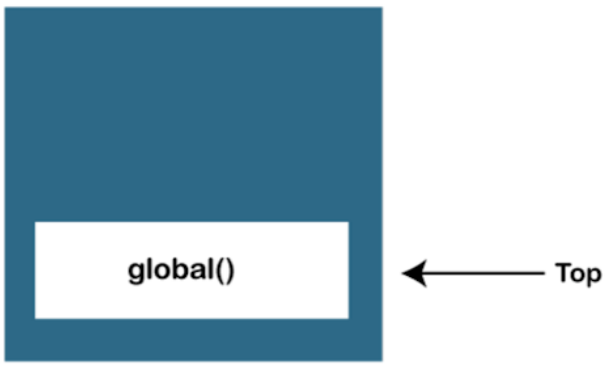
Visualising Call Stack

Let's understand the working of the call stack in depth by considering an example of the below code.

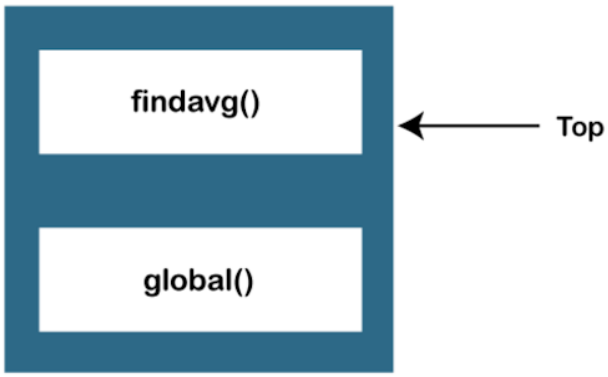
```
function getSum(x, y) {
    return x+ y;
}
function findavg(x,y) {
    return getSum(x,y) / 2;
}

let result = findavg(10, 2);
```

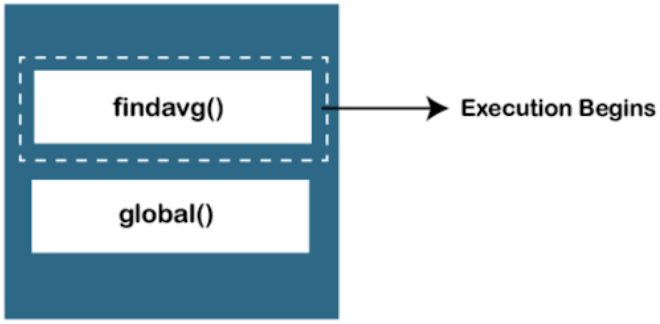
1. At the start of script execution, the JavaScript engine generates a global execution context (i.e., the global() function) and places it at the top of the call stack.



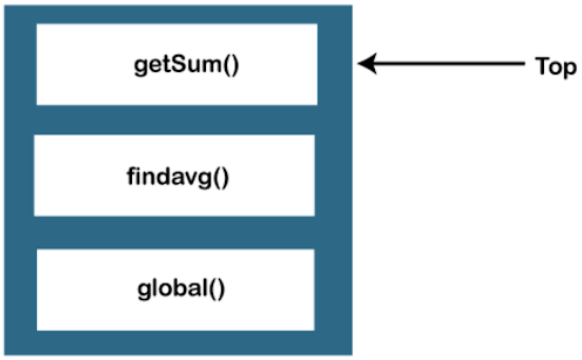
2. When the findavg(10,20) function is called, the JavaScript engine generates a function execution context for it and adds it to the top of the call stack.



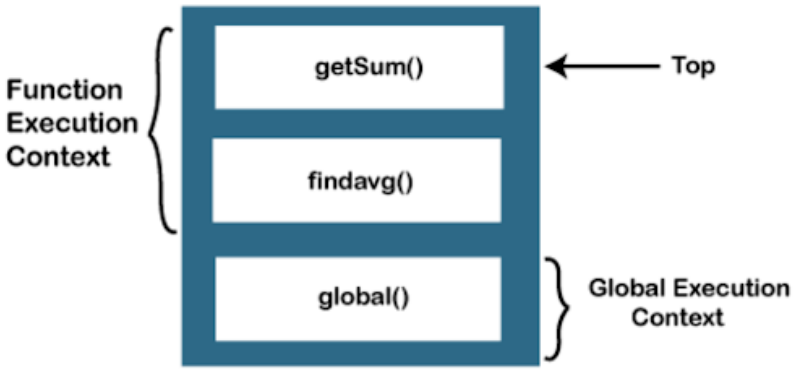
3. Since the findavg() function is at the top of the stack, the JavaScript engine starts executing it.



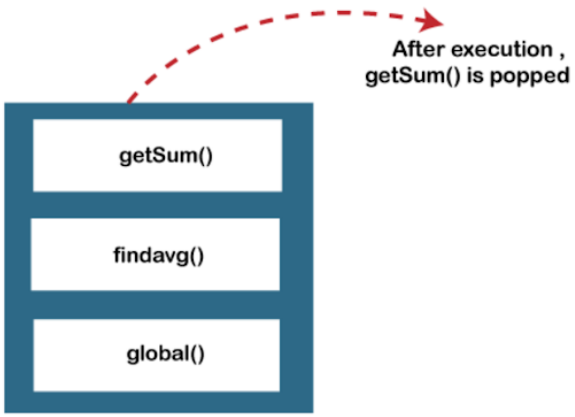
4. As per the code, since the getSum() function is called inside the definition of the findavg() function, the JavaScript engine generates a function execution context for the getSum() function and adds it to the top of the stack.



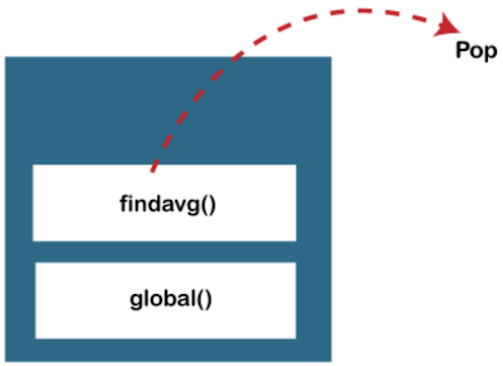
5. Now, there are two functional execution contexts and a global execution context.



6. Thus, the JavaScript engine executes the getSum() function first and then removes it from the call stack.



7. Likewise, the findavg() function is executed and removed from the call stack.



8. Since both function executions have ended, and no additional functions remain in the call stack, the JavaScript engine stops executing the call stack and shifts its focus to other execution tasks.