Explain what is execution context in detail with diagram

JavaScript is a single-threaded interpreted language. Every browser has its own JavaScript engine. Google Chrome has the V8 engine, Mozilla Firefox has SpiderMonkey, and so on. They all are used for the same goal, because the browsers cannot directly understand JavaScript code.

Behind the scenes, JavaScript is doing so many things.

Execution Context

When the JavaScript engine scans a script file, it makes an environment called the Execution Context that handles the entire transformation and execution of the code. During the context runtime, the parser parses the source code and allocates memory for the variables and functions. The source code is generated and gets executed.

There are two types of execution contexts: global and function. The global execution context is created when a JavaScript script first starts to run, and it represents the global scope in JavaScript. A function execution context is created whenever a function is called, representing the function's local scope.

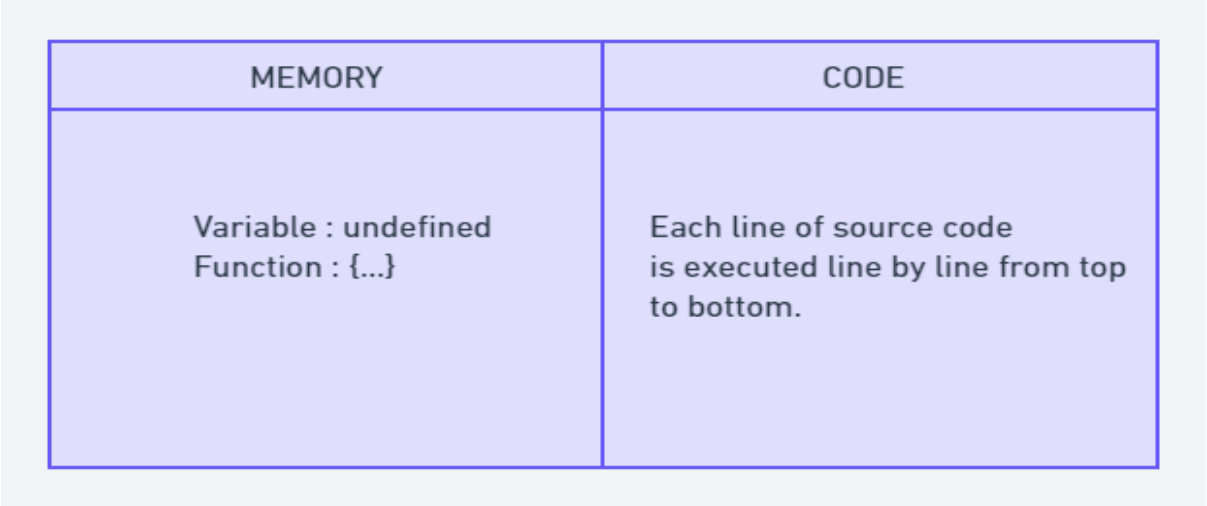
Phases of the JavaScript Execution Context

There are two phases of JavaScript execution context:

1. Creation phase: In this phase, the JavaScript engine creates the execution context and sets up the script's environment. It determines the values of variables and functions and sets up the scope chain for the execution context.
2. Execution phase: In this phase, the JavaScript engine executes the code in the execution context. It processes any statements or expressions in the script and evaluates any function calls.

Everything in JS happens inside this execution context. It is divided into two components. One is memory and the other is code. It is important to remember that these phases and components are applicable to both global and functional execution contexts.

1.Creation Phase

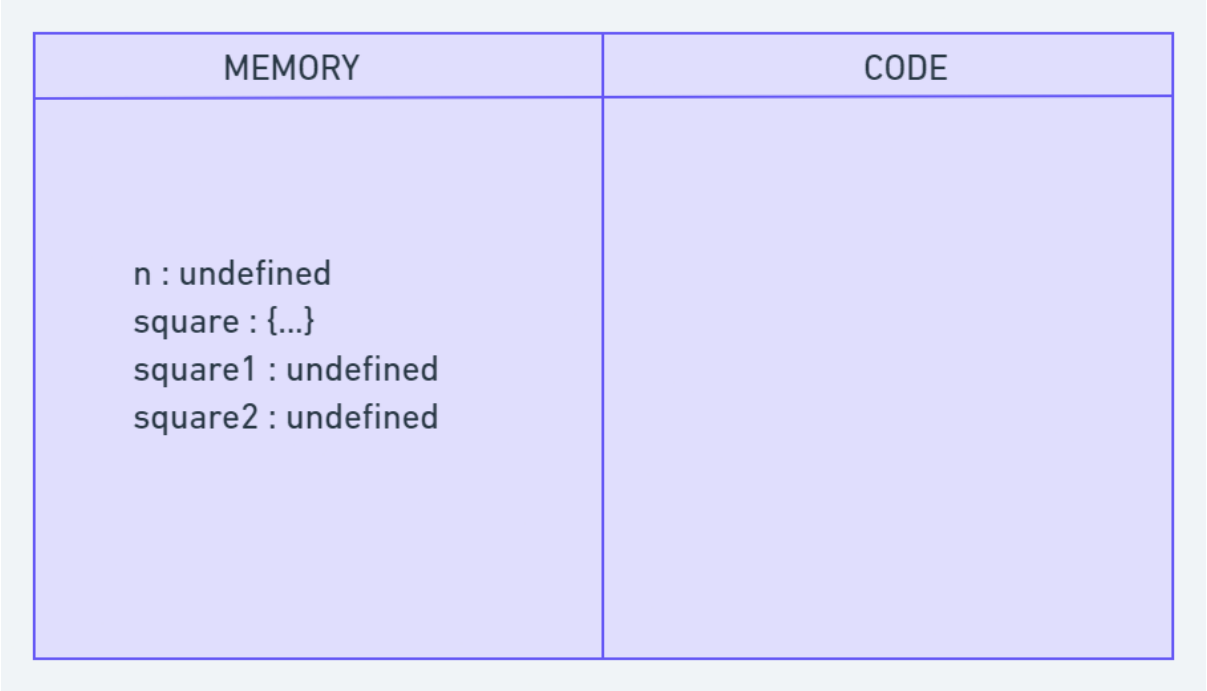


Execution Phase

At the very beginning, the JavaScript engine executes the entire source code, creates a global execution context, and then does the following things:

1. Creates a global object that is window in the browser and global in NodeJs.
2. Sets up a memory for storing variables and functions.
3. Stores the variables with values as undefined and function references.

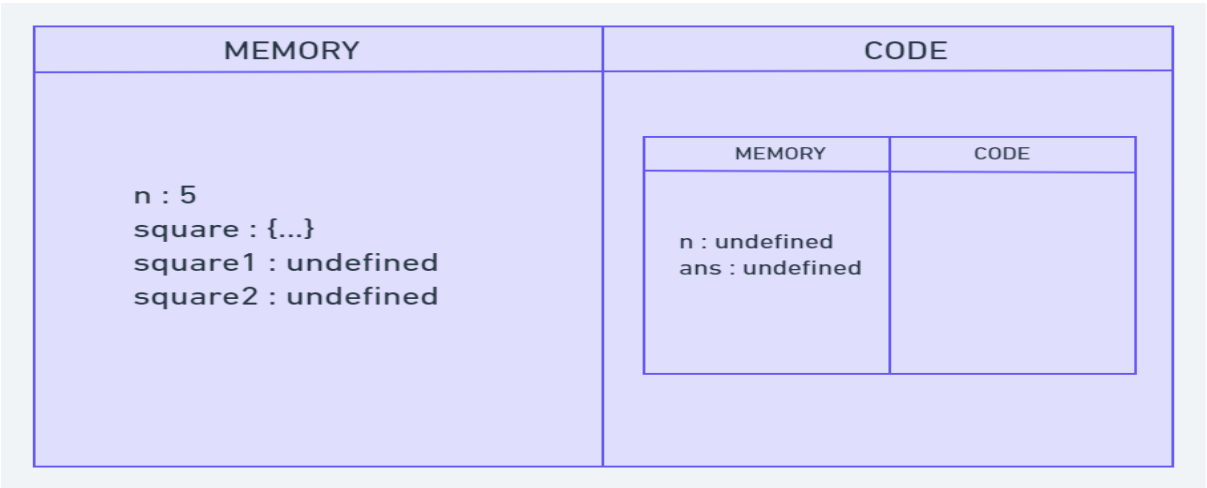
This is called the creation phase. Here's a diagram to help explain it:



Execution Phase

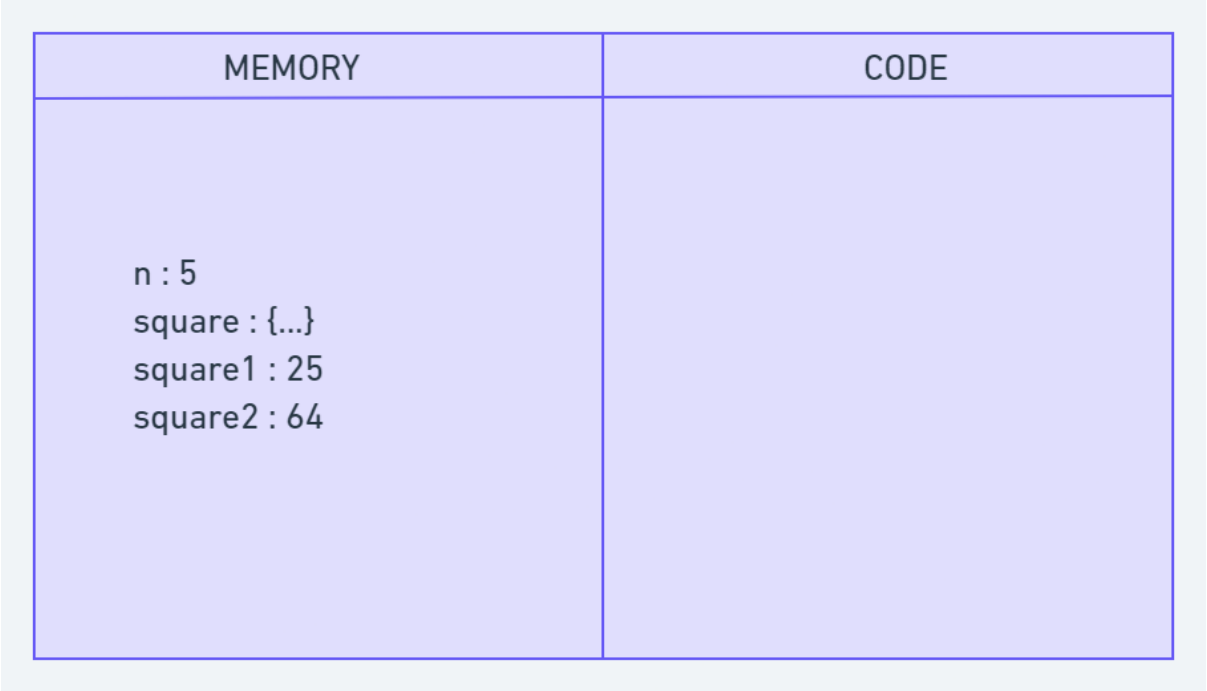
Now, in this phase, it starts going through the entire code line by line from top to bottom. As soon as it encounters n = 5, it assigns the value 5 to 'n' in memory. Until now, the value of 'n' was undefined by default.

Then we get to the 'square' function. As the function has been allocated in memory, it directly jumps into the line var square1 = square(n);. square() will be invoked and JavaScript once again will create a new function execution context.



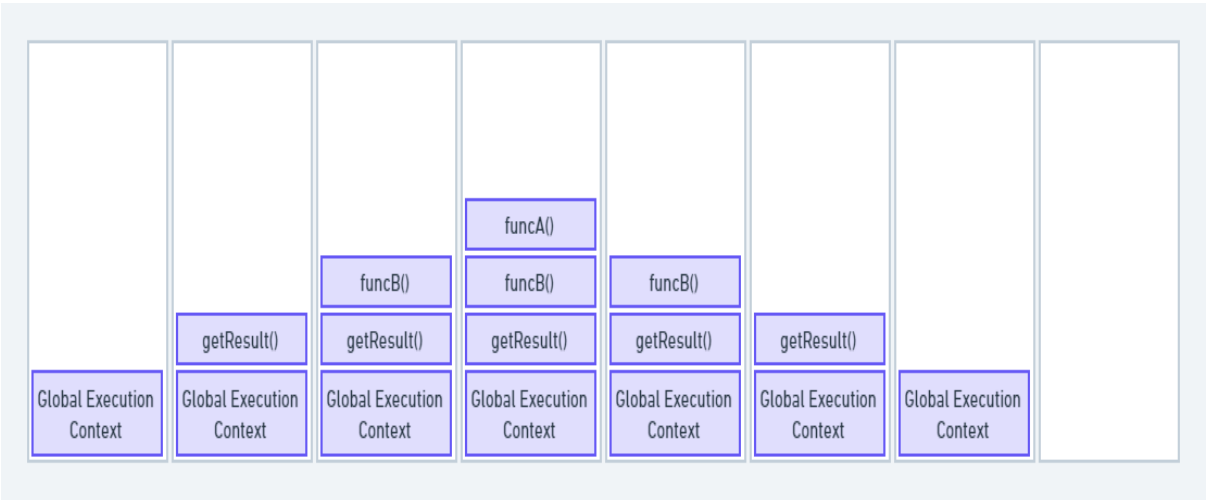
Code Execution Phase

Once the calculation is done, it assigns the value of square in the 'ans' variable that was undefined before. The function will return the value, and the function execution context will be destroyed.
The returned value from square() will be assigned on square1. This happens for square2 also. Once the entire code execution is done completely, the global context will look like this and it will be destroyed also.



Call Stack

A call stack is also known as an 'Execution Context Stack', 'Runtime Stack', or 'Machine Stack'.
It uses the LIFO principle (Last-In-First-Out). When the engine first starts executing the script, it creates a global context and pushes it on the stack. Whenever a function is invoked, similarly, the JS engine creates a function stack context for the function and pushes it to the top of the call stack and starts executing it.
When execution of the current function is complete, then the JavaScript engine will automatically remove the context from the call stack and it goes back to its parent.
In this example, the JS engine creates a global execution context that enters the creation phase.
First it allocates memory for funcA, funcB, the getResult function, and the res variable. Then it invokes getResult(), which will be pushed on the call stack.
Then getResult() will call funcB(). At this point, funcB's context will be stored on the top of the stack. Then it will start executing and call another function funcA(). Similarly, funcA's context will be pushed.
Once execution of each function is done, it will be removed from the call stack. The following picture depicts the entire process of the execution:


funcA()
funcB()
funcB()
funcB()
getResult()
getResult()
getResult()
getResult()
getResult()
Global Execution Context
Global Execution Context
Global Execution Context
Global Execution Context
Global Execution Context
Global Execution Context
Global Execution Context