***JAVASCRIPT EXECUTION CONTEXT***

The JavaScript execution context is a fundamental concept that represents the environment in which JavaScript code is executed. Every piece of JavaScript code runs in an execution context that determines the scope of variables, the value of `this`, and how code is evaluated and executed. Understanding execution contexts is crucial for grasping how JavaScript handle’s function calls, variable scope, and the `this` keyword.

There are three main types of execution contexts in JavaScript:

1. **Global Execution Context (GEC):**

* This is the default or base execution context in which JavaScript code starts its execution when the script loads for the first time.
* There's only one global execution context in a JavaScript program.
* Variables and functions declared in the global scope are attached to the global object (`window` in web browsers, `global` in Node.js).

1. **Functional Execution Context (FEC):**

* This context is created for each function call in the code. Each function has its own execution context, separate from the global execution context.
* Variables and functions defined inside a particular function are not accessible from outside the function, providing a scope chain and closure.
* The functional execution context also manages the scope of variables, the arguments passed to the function, and the value of `this`.

1. **Eval Execution Context:**

* Created by the execution of the `eval` function, although its use is discouraged in modern JavaScript due to security and performance reasons.

Each execution context has two phases:

1. **Creation Phase:**

* The JavaScript engine scans the code for variable declarations (var, let, const) and function declarations. In this phase, it creates the Variable Object (VO) or Lexical Environment and the Scope Chain.
* Functions are fully hoisted (moved to the top of their scope), and variables are partially hoisted (the declaration is hoisted, but the assignment happens in place).
* Determines the value of `this`.

1. **Execution Phase:**

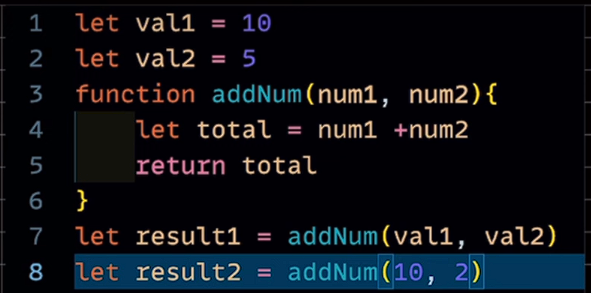
* The code of the function that generated the current execution context is run line by line by the JavaScript engine.
* The variables are assigned their values as the code executes.
* Function calls are made, and as a result, new execution contexts may be created on top of the execution stack.

The JavaScript engine maintains a stack known as the execution stack (or call stack), where the global execution context sits at the bottom. Whenever a function is called, its execution context is pushed onto the stack. When the function execution completes, its execution context is popped off the stack, returning control to the context below it.

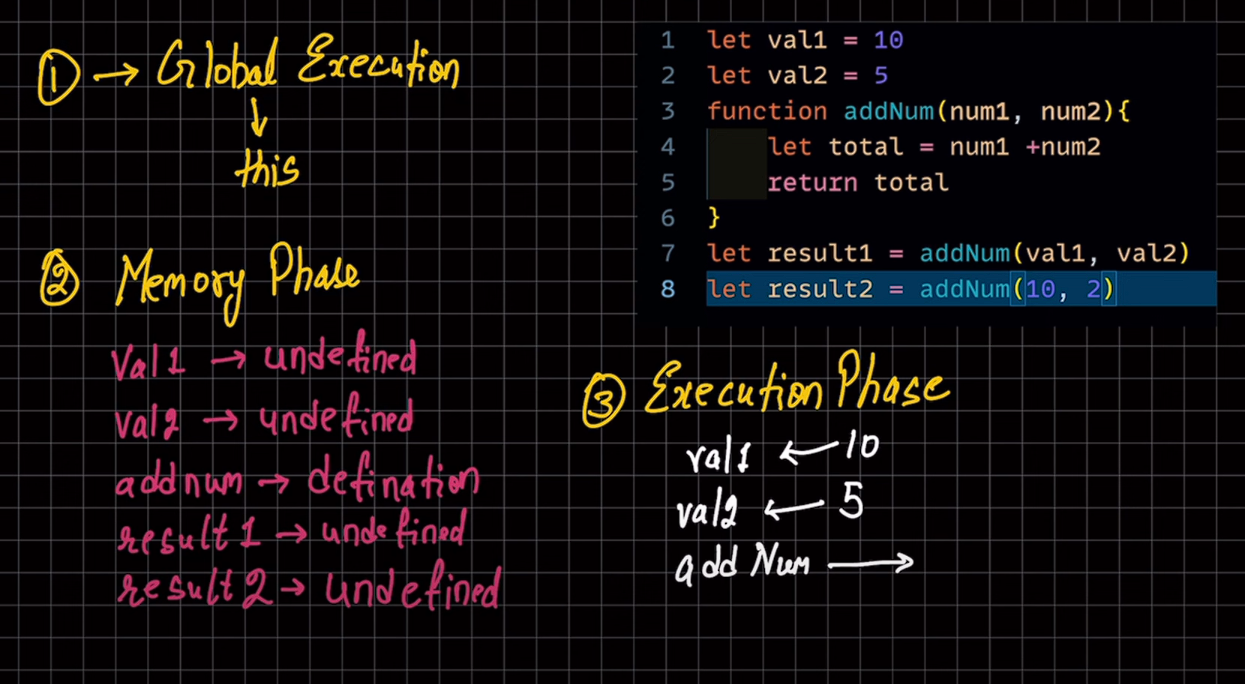
This mechanism ensures that variables and functions have the correct scope and access, and it manages the execution flow of a JavaScript program, especially during function calls, recursion, and callbacks.

***COMPLETE PROCESS BY EXAMPLE:***

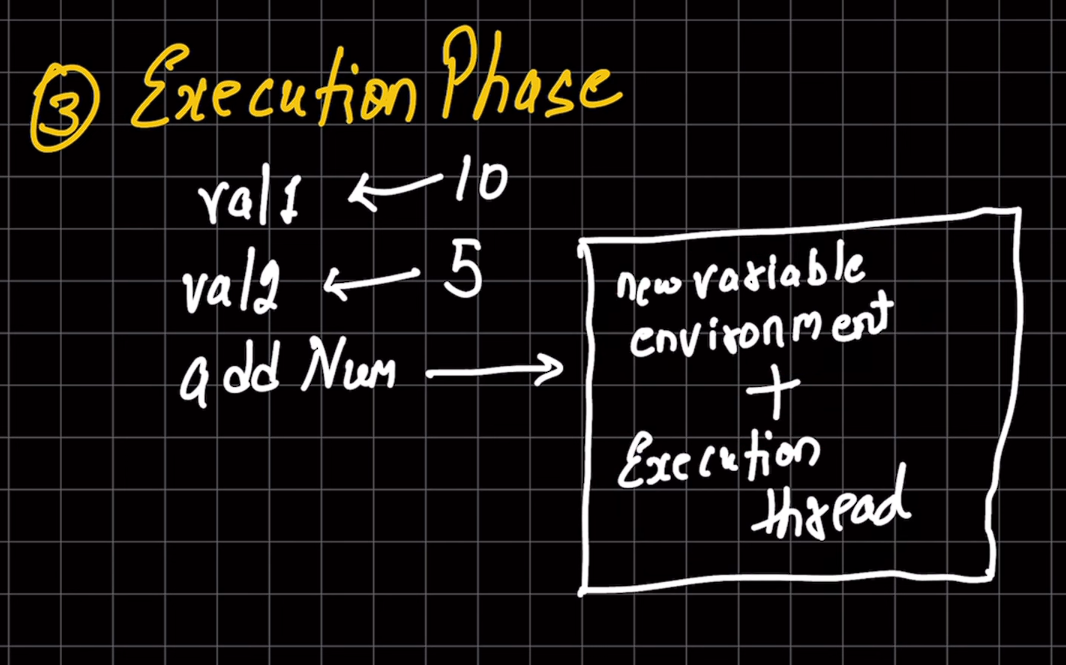
The following code will be used for the explanation



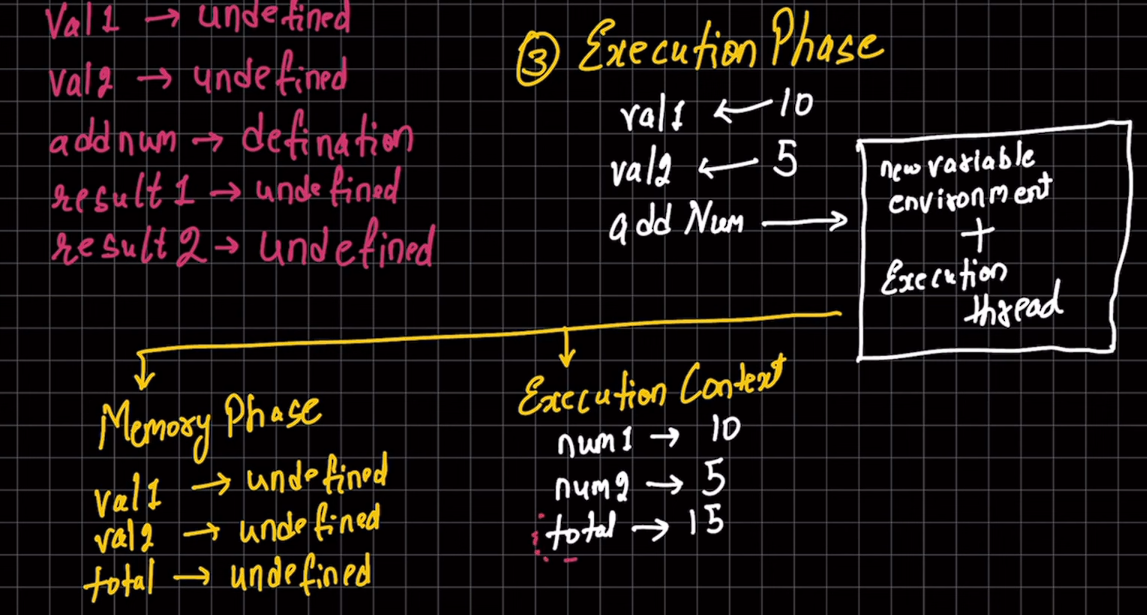
Step1: CODE RUNS TILL LINE 6



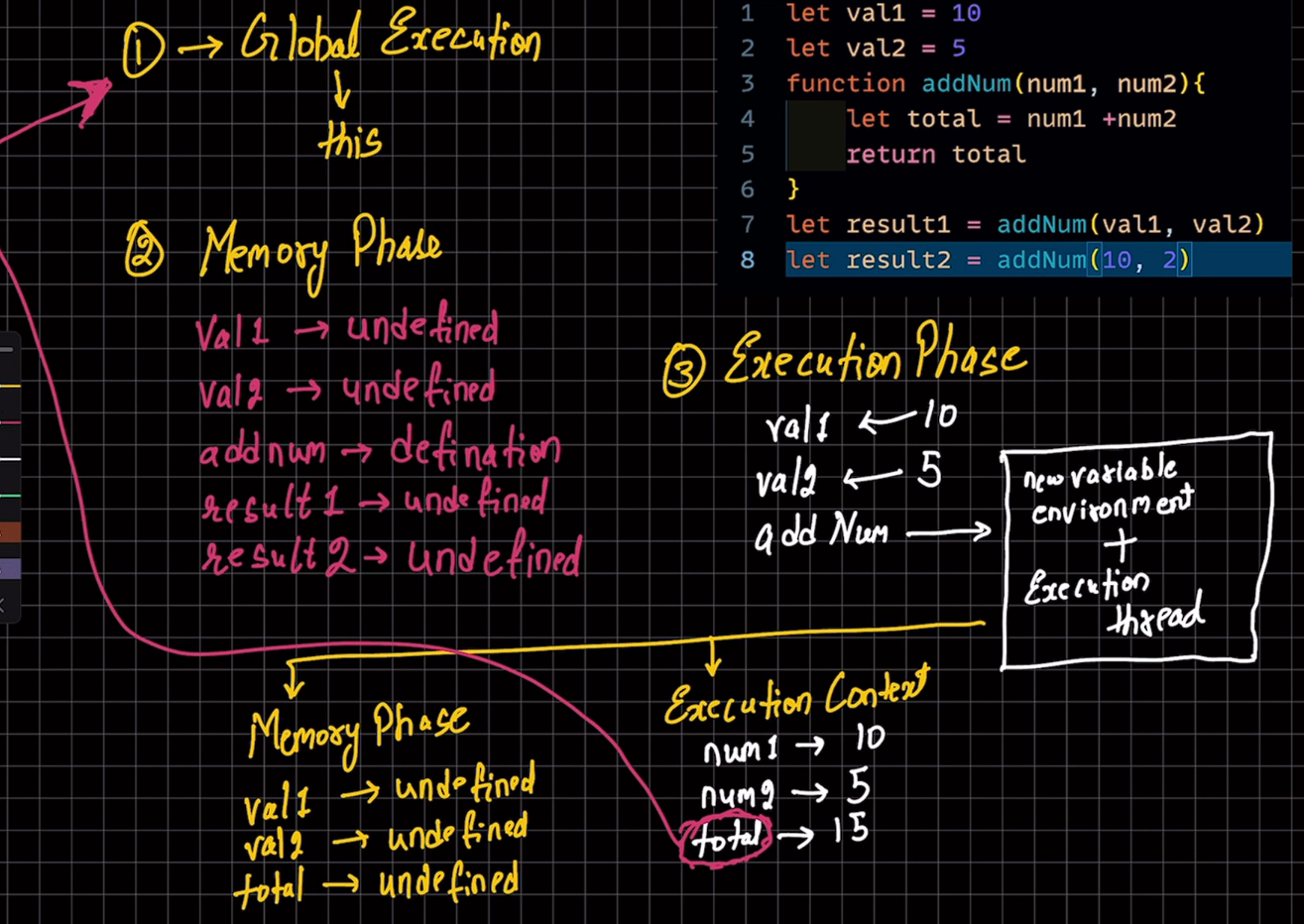
Step 2: FOR LINE 7: Functional Executional Context



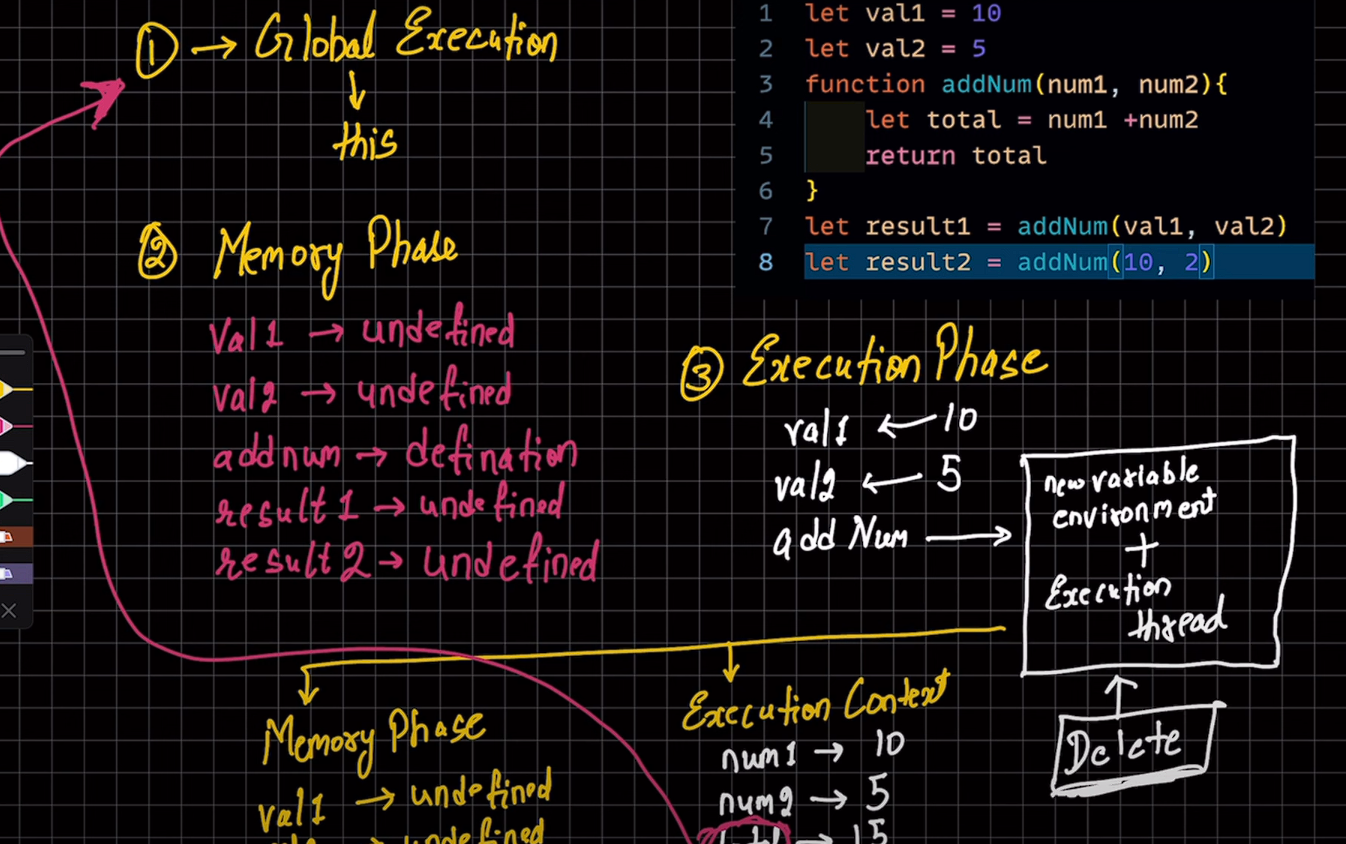
Step 3: The same process now happens for this functional executional context



Step 4: Now the total value is returned to the Global Executional Context



Step 5: after this the functional executional context is deleted



Step 6: the result will get the value as 15 and at line 8 a new functional executional context will be made

