How JS Works Internally?

Friday, February 28, 2025 5:21 PM

1. Global Execution Context (GEC) and How JavaScript Works Under the Hood:

- Whenever JavaScript code runs, a **Global Execution Context (GEC)** is created, which consists of two phases: **Memory Creation Phase** and **Code Execution Phase**.
- In the **Memory Creation Phase**, all variables (var) are set to undefined, and function declarations are stored entirely in memory.
- In the **Execution Phase**, JavaScript updates variable values and executes functions as it reads the code line by line.
- Function expressions (anonymous functions assigned to variables) behave like normal variables and are initially assigned undefined, leading to errors if called before declaration.

2. Hoisting:

- **Hoisting** is the process where variables and functions are moved to the top of their scope during the memory creation phase.
- Variables declared with var are hoisted and assigned undefined, while functions declared with the function keyword are hoisted with their full definition.
- let and const are also hoisted but are placed in the **Temporal Dead Zone (TDZ)** until their declaration is encountered.
- Accessing let or const before their declaration results in a **ReferenceError**.

Temporal Dead Zone (TDZ):

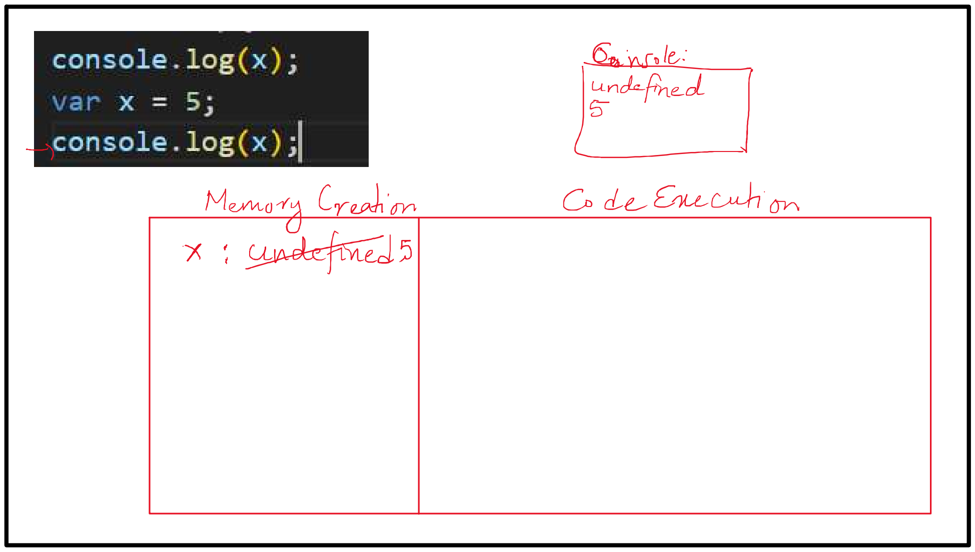
- The **TDZ** is the time between the start of the execution context and the point where a let or const variable is declared.
- Unlike var, which gets undefined in memory, let and const are stored in the **TDZ** and cannot be accessed before declaration.
- Attempting to use these variables before declaration results in a **ReferenceError**.
- This ensures better error handling and prevents unintentional use of undeclared variables.

3. Call Stack

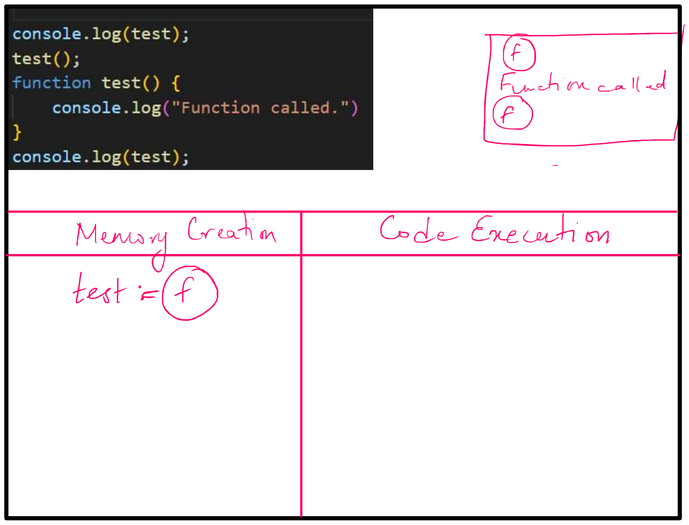
- The **Global Execution Context (GEC)** is the **first item** that gets pushed onto the **Call Stack** when JavaScript starts execution. It remains there until the entire script finishes running.
- Whenever a function is called, JavaScript **pushes it onto the Call Stack**, and when the function completes execution, it is **popped off** the stack.
- If a function calls another function, the new function gets added **on top** of the stack, and JavaScript always executes the function **at the top first** following the **LIFO (Last In, First Out)** principle.

4. Lexical Environment and Scope Chaining:

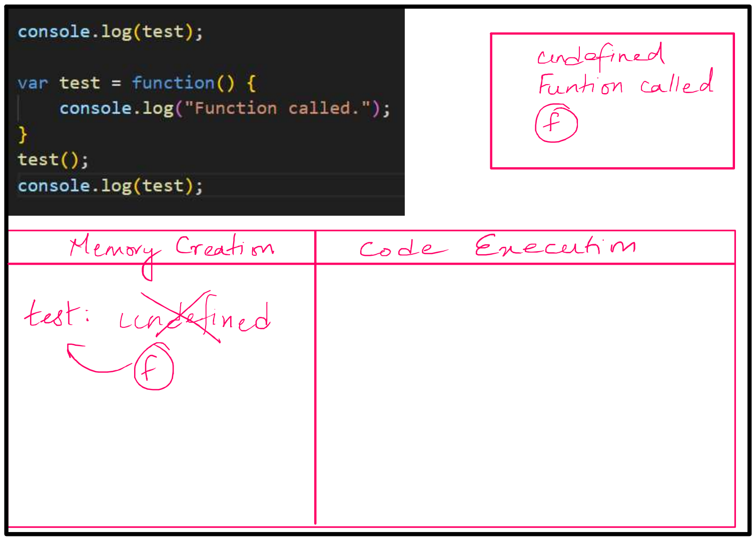
- **Lexical Environment** consists of the **current function's scope** and a **reference to its parent's scope** (outer environment).
- When accessing a variable, JavaScript first looks in the **local scope**, then moves outward through **scope chaining** until it finds the variable.
- Scope chaining connects **nested functions** with their parent's lexical environment, enabling access to outer variables.
- If a variable is not found in any scope, JavaScript throws a **ReferenceError**.



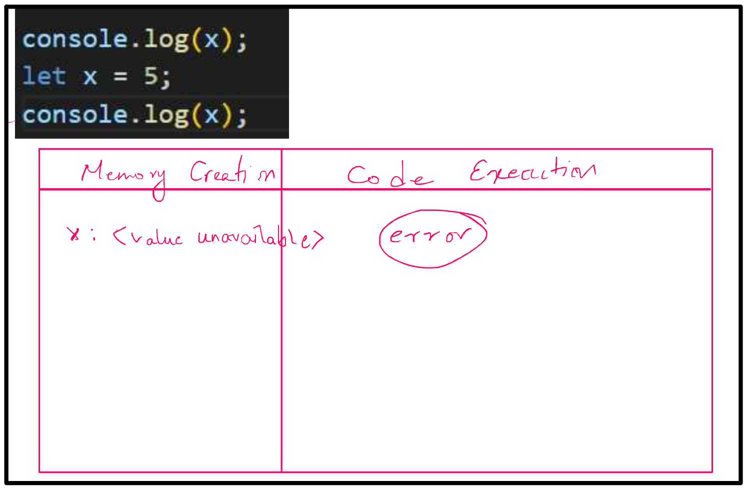
```
console.log(x);
var x = 5;
console.log(x);
```



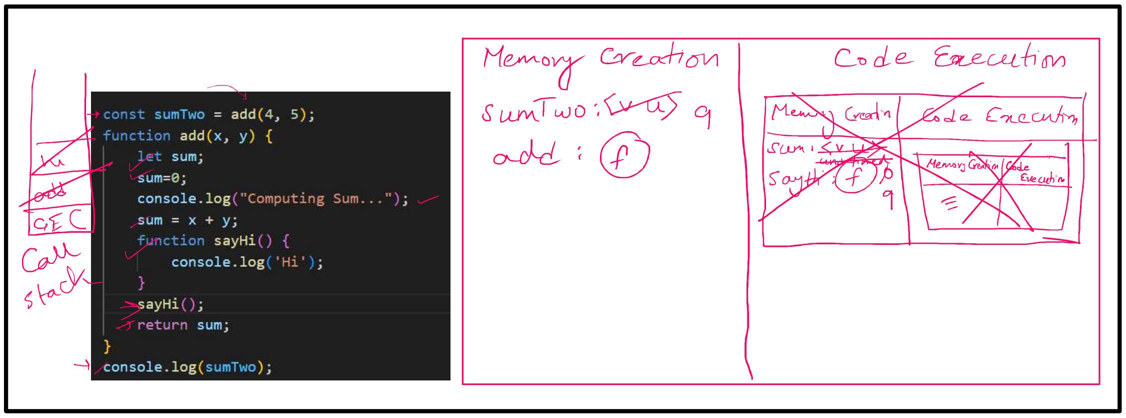
```
console.log(test);
test();
function test() {
    console.log("Function called.")
}
console.log(test);
```



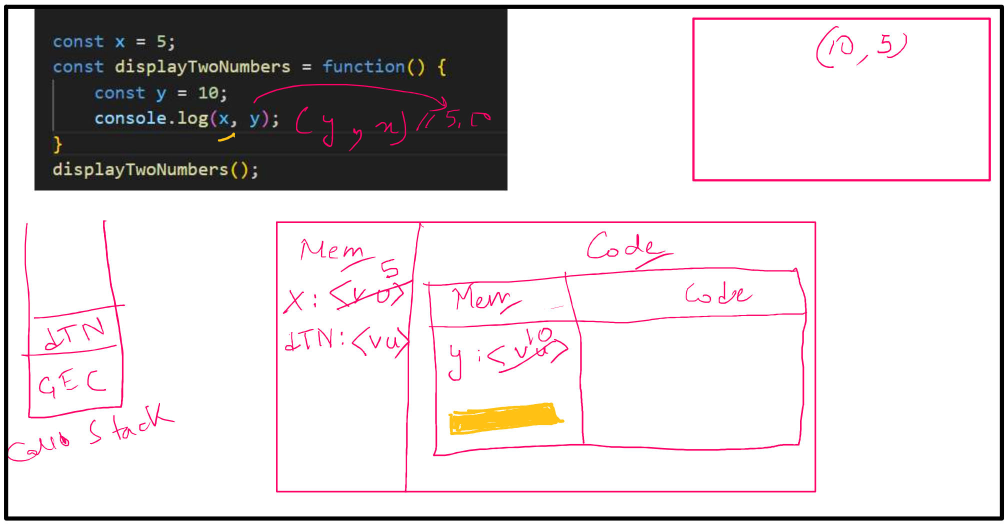
```
console.log(test);

var test = function() {
    console.log("Function called.");
}
test();
console.log(test);
```



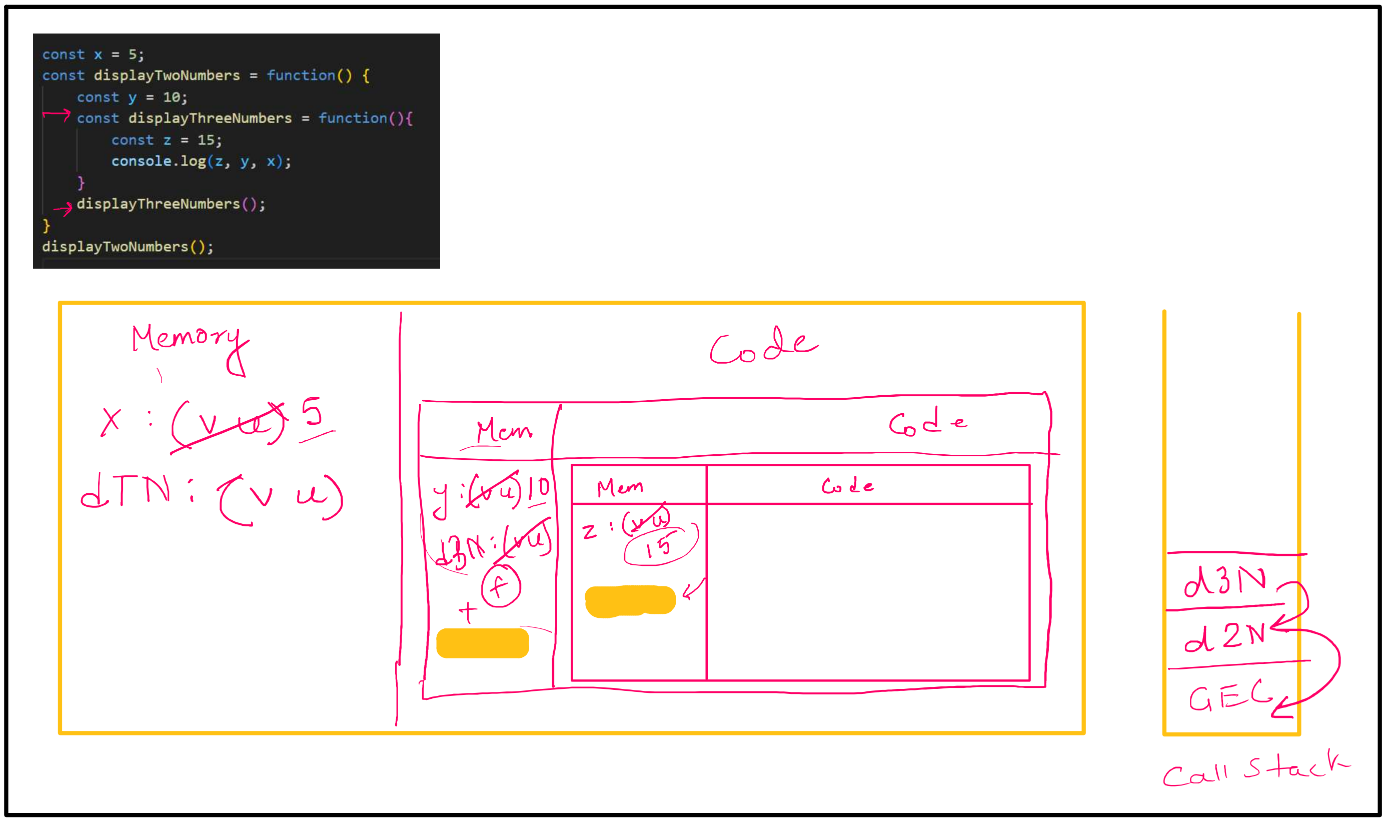
```
console.log(x);
let x = 5;
console.log(x);
```



```
const sumTwo = add(4, 5);
function add(x, y) {
    let sum;
    sum=0;
    console.log("Computing Sum...");
    sum = x + y;
    function sayHi() {
        console.log('Hi');
    }
    sayHi();
    return sum;
}
console.log(sumTwo);
```



```
const x = 5;
const displayTwoNumbers = function() {
    const y = 10;
    console.log(x, y);
}
displayTwoNumbers();
```



```
const x = 5;
const displayTwoNumbers = function() {
    const y = 10;
    const displayThreeNumbers = function(){
        const z = 15;
        console.log(z, y, x);
    }
    displayThreeNumbers();
}
displayTwoNumbers();
```