Global Execution Context

Global execution context:

Global Execution Context is the first context that gets created when the JavaScript engine starts executing code.

Key Components

1. Memory Allocation (Creation Phase):

- During this phase, the engine sets up the memory for variables and functions.
- **Variables** declared with **var** are hoisted and initialized with **undefined**.
- **Function declarations** are hoisted and their definitions are stored in memory.
- Variables declared with **let** and **const** are also hoisted but are not initialized. They remain in a temporal dead zone until they are assigned a value

2. Code Execution (Execution Phase):

· The JavaScript engine executes the code line by line.

· Variables declared with **var** are assigned their values.

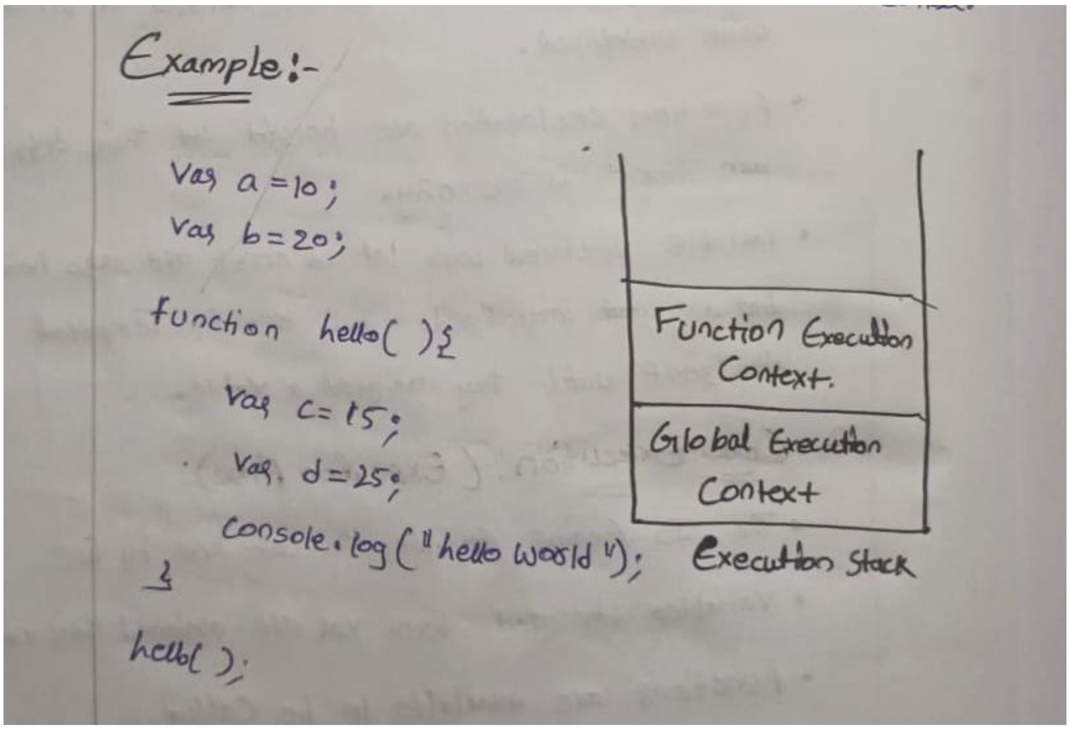
· Functions are available to be called.

· Variables declared with **let** and **const** are assigned values when their declaration is encountered in the code.

How it works on the functions

When a function is invoked, a new **Execution Context(function execution context)** is created specifically for that function. This context is separate from the Global Execution Context but follows similar principles.

- Each function invocation creates a new execution context.
- Variables declared inside a function are local to that function and are not accessible outside it.
- The scope chain allows inner functions to access variables from their parent functions and the global context.

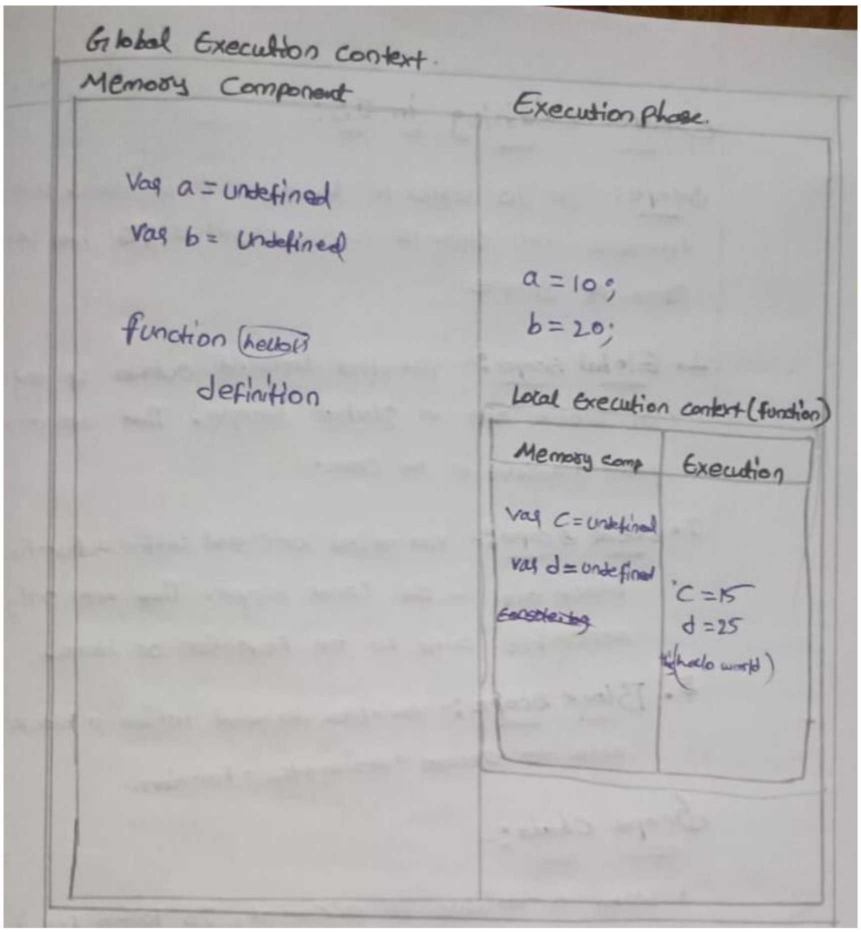


Example:-

```
Var a=10;
Var b=20;

function hello( ){
    Var c=15;
    Var d=25;
    console.log("hello world");
}

hello( );
```

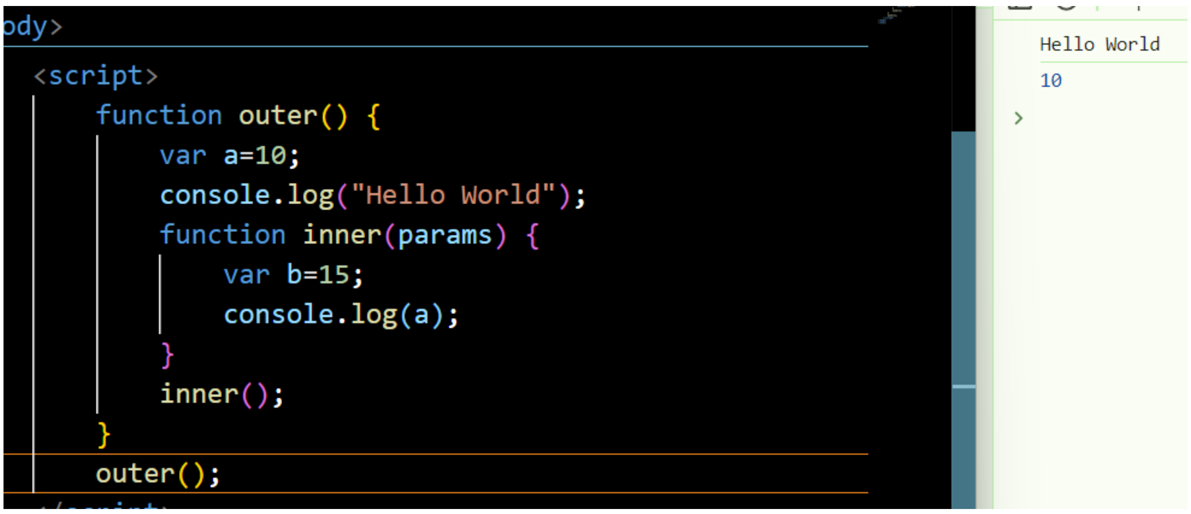


Closures

A closure is a function that has access to its own scope, the scope of the outer function, and the global scope. This means a closure can remember and access variables from its outer function even after that function has finished executing.

(Or)

When a inner function have a access to the variable of outer function even after the outer function has been executed.



```
ody>
  <script>
      function outer() {
          var a=10;
          console.log("Hello World");
          function inner(params) {
              var b=15;
              console.log(a);
          }
          inner();
      }
      outer();
```

```
Hello World
10
>
```

Scope Chaining in JavaScript

Scope in JavaScript refers to the context in which variables, functions, and objects are accessible. JavaScript has three types of scope:

1. **Global Scope:** Variables declared outside of any function or block are in the global scope. They are accessible from anywhere in the code.

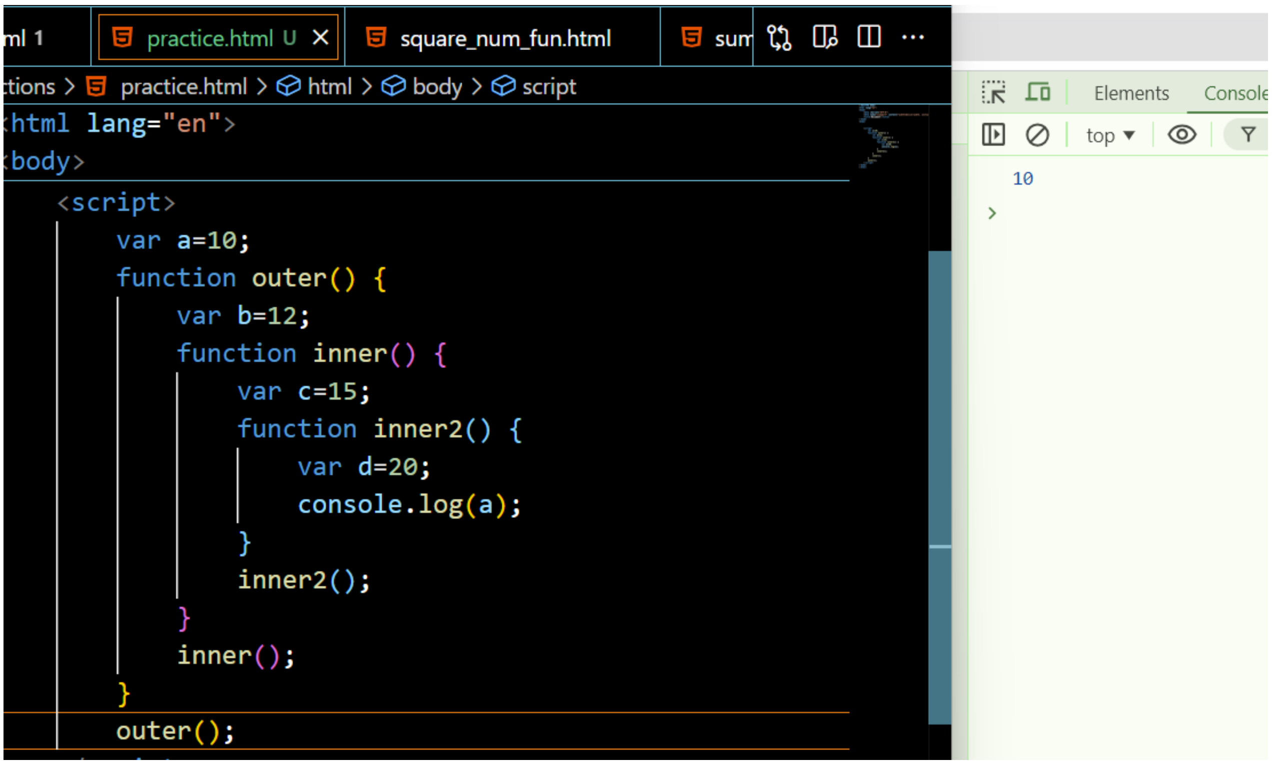
2. **Local Scope:** Variables declared within a function or block are in the local scope. They are only accessible within that function or block.

3. **Block Scope:** Variables declared inside a block can't able to access outside of the function

Scope Chain:

- When a variable is accessed, JavaScript looks for it in the current scope.
- If the variable is not found, it looks in the outer scope.
- This process continues until it reaches the global scope.
- If the variable is not found in any scope, it results in a ReferenceError

Example:



```
html lang="en">
body>
    <script>
        var a=10;
        function outer() {
            var b=12;
            function inner() {
                var c=15;
                function inner2() {
                    var d=20;
                    console.log(a);
                }
                inner2();
            }
            inner();
        }
        outer();
```

```
10
>
```

Lexical Scoping:

- JavaScript uses lexical scoping, meaning that the scope of a variable is determined by its position in the source code.
- Inner functions have access to variables declared in their outer functions (but not vice versa).