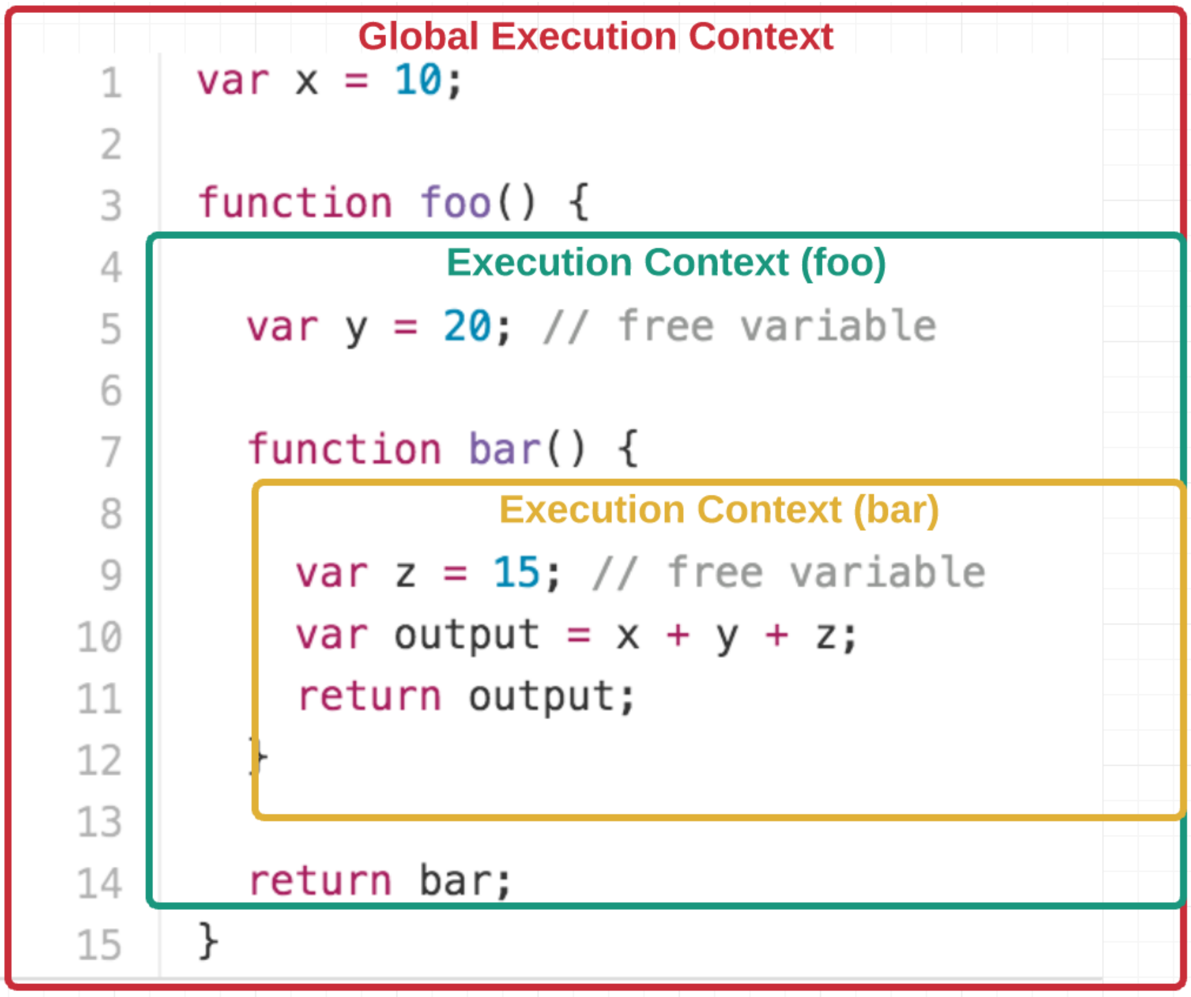

```javascript
var x = 10;

function foo() {

  var y = 20; // free variable

  function bar() {

    var z = 15; // free variable
    var output = x + y + z;
    return output;
  }

  return bar;
}
```

When JavaScript code is run, it is executed within a global execution context, which represents the top-level scope. Each time a function is called, a new execution context is created, forming a stack of execution contexts known as the "call stack". The call stack follows a Last-In-First-Out (LIFO) order, meaning that the most recently called function is at the top of the stack.
An execution context consists of three key components:

> Variable Environment: It stores the variables and function declarations within the current scope. This includes the function arguments, local variables, and any variables declared using the var keyword. The variable environment is initialized with the values of the variables and functions defined within the current scope.
>
> Scope Chain: It is a list of all the variable environments within which the current function is lexically located. It allows the function to access variables and functions from its outer (enclosing) scopes. The scope chain is created based on the nested structure of the code, and it is used during variable lookup.
>
> This Value: It refers to the object that the current function is bound to when invoked. The value of this depends on how the function is called, and it provides a reference to the object on which the function is operating.

```
var x = 10;

function outer() {
  var y = 20;

  function inner() {
    var z = 30;
    console.log(x + y + z);
  }

  inner();
}

outer();
```

Execution context is an important concept in JavaScript that helps in understanding how code is executed. It refers to the environment in which JavaScript code is executed, including variables, functions, and the scope chain.
When JavaScript code is run, it is executed within a global execution context, which represents the top-level scope. Each time a function is called, a new execution context is created, forming a stack of execution contexts known as the "call stack". The call stack follows a Last-In-First-Out (LIFO) order, meaning that the most recently called function is at the top of the stack.
An execution context consists of three key components:

Variable Environment: It stores the variables and function declarations within the current scope. This includes the function arguments, local variables, and any variables declared using the var keyword. The variable environment is initialized with the values of the variables and functions defined within the current scope.

Scope Chain: It is a list of all the variable environments within which the current function is lexically located. It allows the function to access variables and functions from its outer (enclosing) scopes. The scope chain is created based on the nested structure of the code, and it is used during variable lookup.

This Value: It refers to the object that the current function is bound to when invoked. The value of this depends on how the function is called, and it provides a reference to the object on which the function is operating.
To illustrate the concept of execution context, let's consider the following code **example:**

javascript
Copy code

```
var x = 10;

function outer() {
  var y = 20;

  function inner() {
    var z = 30;
    console.log(x + y + z);
  }

  inner();
}

outer();
```

In this example, when the code is executed, the following execution contexts are created:

Global Execution Context:

- Variable Environment: x (initialized with 10)
- Scope Chain: null (global scope)
- This Value: window (assuming this code is running in a browser)

outer Execution Context:

- Variable Environment: y (initialized with 20)
- Scope Chain: Global Environment
- This Value: window

inner Execution Context:

- Variable Environment: z (initialized with 30)
- Scope Chain: outer Environment -> Global Environment
- This Value: window

The execution starts with the global context, executing the outer function within it. This creates a new execution context for outer and pushes it onto the call stack. Within outer, the inner function is called, creating another execution context and pushing it onto the call stack. Finally, the inner function executes and logs the sum of x, y, and z (which is 60).
As each function completes its execution, its execution context is popped off the call stack, and control returns to the calling context.