

# JavaScript: Execution Context & Lexical Environment

---

## 1. Execution Context (EC)

---

An **Execution Context** is the environment in which JavaScript code is evaluated and executed.

### Types of Execution Contexts

#### 1. Global Execution Context (GEC)

- Created when the JS engine starts.
- Creates the global object (`window` in browsers, `global` in Node.js).
- `this` points to the global object.
- Only one global context exists.

#### 2. Function Execution Context (FEC)

- Created whenever a function is invoked.
- Each function has its own EC.
- Contains:
  - Arguments object
  - Local variables
  - References to outer environments

#### 3. Eval Execution Context

- Created when code is executed inside `eval()`.
  - Rarely used.
-

## Lifecycle of Execution Context

### 1. Creation Phase

- Variable Object (VO) / Environment Record created.
- Stores function arguments, variables (`var`), and function declarations.
- Scope chain established (lexical environment links).
- `this` binding determined.

### 2. Execution Phase

- Code is executed line by line.
  - Variables assigned, functions invoked.
- 

## Execution Stack (Call Stack)

- JavaScript is **single-threaded**.
  - Execution contexts are managed in a stack (LIFO):
    - GEC is at the bottom.
    - Each function call pushes a new EC.
    - Function returns → EC popped off.
- 

## 2. Lexical Environment (LE)

A **Lexical Environment** is a structure that holds identifier-variable mapping (where variable names map to memory addresses).

### Structure

- **Environment Record**: Stores variable/function declarations.
- **Outer Lexical Environment Reference**: A reference to its parent's LE (static scope).

### Key Properties

#### 1. Lexical = Defined at code-writing time

- Scope is determined by where functions are declared, not where they are called.

```
function outer() {
  let a = 10;
  function inner() {
    console.log(a); // inner can access a
  }
  inner();
}
outer();
```

## 2. Closures

- When a function "remembers" variables from its lexical scope, even after that scope has finished execution.

```
function makeCounter() {
  let count = 0; // variable in outer (lexical) scope
  // inner function forms a closure over "count"
  return function() {
    count++;
    return count;
  };
}

// "counter" is now that inner function (closure)
const counter = makeCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

Here:

- `makeCounter` finishes executing, but the variable `count` is kept alive because the returned function still references it.
- Each call to `counter()` accesses the same `count` — that's the closure in action.

## 3. Scope Chain

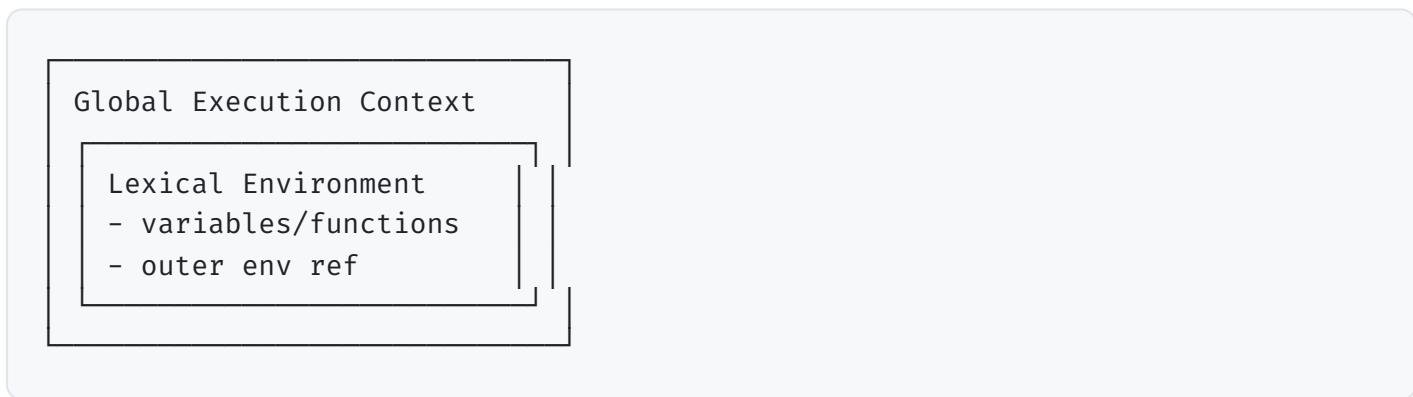
- Variable resolution starts in the current LE.
- If not found, looks up the chain.
- Ends at the Global Environment.

## 3. Execution Context vs. Lexical Environment

Aspect	Execution Context	Lexical Environment
Definition	The environment in which JS code is executed.	The structure that defines variable scope.
Created	At runtime (when code executes).	At compile time (when code is written).
Contains	VO, Scope Chain, <code>this</code> .	Environment Record + Reference to outer scope.
Scope	Dynamic during execution.	Static, based on code structure.
Relation	Uses lexical environment to resolve variables.	Provides rules for variable access.

## 4. Visualization

### Call Stack (Execution Contexts)



## Summary

- **Execution Context:** "Where" code is executed (global, function, eval). Created at runtime.
- **Lexical Environment:** "How" variable names are resolved. Created at compile time.
- Together, they explain **scope, closures, and the call stack** in JavaScript.