Execution Context Deep Dive

(How JavaScript Engine Actually Runs Code)

Subtitle: Understand how JavaScript creates, stores, and executes every line of code using execution contexts, memory allocation, and lifecycle phases.

Website Name: haas.dev
Website Link: https://dev-roast-app.vercel.app

🧠 Introduction

So far you've seen:

- hoisting
- scope
- call stack
- closures
- recursion

All of these depend on one hidden system:

Execution Context

If you don't understand this, JavaScript will always feel like it has "random behavior."

Execution context is the **core unit of JavaScript execution**.

⚙️ Step 1: What is Execution Context?

An execution context is:

The environment in which JavaScript code is executed.

Every time JavaScript runs code, it creates a context.

Types of Execution Context:

1. Global Execution Context
2. Function Execution Context
3. Eval Execution Context (rare, not important for now)

🧠 Step 2: Global Execution Context (GEC)

When JS starts running:

- one global context is created
- it stays until program ends

Example:

```
let a = 10;


function test() {

  console.log("hello");

}


test();
```

Inside GEC:

- variables stored
- functions stored
- memory allocated

⚙️ Step 3: Function Execution Context (FEC)

Every function call creates a new context.

```
function add(a, b) {

  return a + b;

}


add(2, 3);
```

When add() runs:

A new execution context is created.

🔄 Step 4: Execution Context Lifecycle

Every context goes through 2 phases:

1. Memory Creation Phase

- variables declared
- functions stored
- memory allocated

2. Execution Phase

- code runs line by line
- values assigned
- functions executed

Flow:

Creation Phase → Execution Phase

🧠 Step 5: Internal Structure of Execution Context

Each context contains:

- Variable Environment
- Lexical Environment
- This Binding

Simplified Model:

Execution Context =

Memory + Code Execution + Scope Rules

🔄 Step 6: How Call Stack Uses Execution Context

Example:

```
function a() {
  b();
}

function b() {
  console.log("B");
}
```

a();

Stack Flow:

GEC → a() → b()

Each function = new execution context

⚠️ Step 7: Hoisting Inside Execution Context

console.log(x);

var x = 5;

Memory Phase:

x = undefined

Execution Phase:

x = 5

👉 This is NOT magic — it's execution context behavior.

🌍 Step 8: Real-World Mental Model

Think of execution context like:

A Workspace for Each Function

Each function gets:

- its own desk (memory)
- its own tools (variables)
- its own execution time

When done → desk is cleared (popped from stack)

🧠 Step 9: Lexical Environment Connection

Execution context also defines:

- scope rules
- variable access chain

So:

execution context = runtime environment + scope system

⚠️ Step 10: Common Mistakes

❌ 1. Thinking JS runs line by line only

It first creates memory phase

❌ 2. Ignoring function contexts

Every function creates a new world

❌ 3. Confusing stack with context

Stack = order
Context = environment

❌ 4. Not linking scope with execution context

Scope is inside context

🧪 Step 11: Mini Exercises

Exercise 1

Trace execution context in nested functions

Exercise 2

Predict memory phase output of var hoisting

Exercise 3

Identify how many execution contexts are created in code

🧠 Step 12: Mini Quiz

1. What is execution context?
2. What are its two phases?
3. How does call stack relate to it?
4. What is inside an execution context?

🚀 Step 13: Thinking Upgrade

If you understand execution context:

- JavaScript is no longer unpredictable
- debugging becomes logical tracing
- you think like JS engine itself

👉 This is the “engine-level thinking” point.

📌 Step 14: Summary

- Execution context = environment where JS runs
- Every function call creates a new context
- Two phases: memory + execution
- Stored and managed via call stack
- Core system behind hoisting, scope, closures