How JavaScript Works & Execution Context Explained 🚀

Introduction

Ever wondered how JavaScript executes your code behind the scenes? Let's break it down in **simple terms**!

What is an Execution Context?

Everything in JavaScript happens **inside an Execution Context (EC)**. Think of it as a **big box** where your code is **stored and executed**.

Execution Context Has Two Components:

□ Memory Component (Variable Environment)

- Stores **variables & functions** as **key-value pairs** before execution.
- Example:
- `let a = 10;`
- `function greet() {`
- `    console.log("Hello");`
  `}`

  Memory Component (before execution):

  ```
  a: undefined
  greet: function() {...}
  ```

□ Code Component (Thread of Execution)

- **Executes JavaScript code line by line, in order**.
- It retrieves stored variables and functions from memory and runs them step by step.
- JavaScript follows a **single-threaded** and **synchronous** execution model, meaning it **only moves to the next line after the current one is fully executed**.

JavaScript is Single-Threaded & Synchronous

JavaScript follows two important rules:

✅ **Single-threaded:** Executes **one command at a time** (like a queue).
✅ **Synchronous: Each line must complete** before moving to the next.

Example:

```
console.log("Start");
let x = 5;
console.log(x);
console.log("End");
```

Execution Flow:

1. "Start" is printed
2. `x = 5` is stored in memory
3. `5` is printed
4. "End" is printed

Execution Context Visualized:

```
Execution Context (EC)

□ Memory Component
- x: undefined → 5
- Functions stored

□ Code Component
- console.log("Start")
- Assign x = 5
- console.log(x)
- console.log("End")
```

Execution Context Phases

JavaScript code runs in **two phases**:

1. **Creation Phase (Memory Allocation)**

- All variables are set to **undefined**.
- Functions are **stored in memory**.

2. **Execution Phase**

- Code runs **line by line**.
- Variables are **assigned values**.

Example with `var`:

```
console.log(y);  // Output: undefined
var y = 10;
console.log(y);  // Output: 10
```

🚀 Why `undefined`?

Because JavaScript **reserves memory** for `y`, but doesn't assign a value until the execution phase.

Call Stack in JavaScript

Execution Contexts are **stacked like plates** 🍽 in a **Call Stack**:

1. Global Execution Context (GEC) is created.

2. Each function call **creates a new Execution Context**.

3. Once a function finishes, its context **is removed** from the stack.

Example:

```
function first() {
  console.log("First");
}
function second() {
  first();
  console.log("Second");
}
second();
console.log("End");
```

Call Stack Flow:

```
┌──────────┐
│   GEC    │   // Global Execution Context
│ ──────── │
│ second() │   // Call second()
│ first()  │   // Call first()
└──────────┘
```

🚀 **Execution Order:**

✅ `first()` runs → prints "First"
✅ `first()` finishes → removed from stack
✅ `second()` prints "Second" → finishes
✅ "End" prints after `second()` completes

Summary

✅ JavaScript runs inside an **Execution Context**.
✅ It has **Memory (Variable Env) & Code (Execution Thread)**.
✅ JavaScript is **Single-Threaded & Synchronous**.
✅ The **Call Stack** manages execution.

🎯 **Now you know how JavaScript actually runs your code!** 🚀

💡 **Did you find this helpful?** Comment below! 👇