2. How JavaScript Code is executed? ❤️& Call Stack

📘 Index (Table of Contents)

1. **Introduction to Execution Context**
2. **Phases of Execution Context**
    - Memory Creation Phase
    - Code Execution Phase
3. **Call Stack and Its Role**
4. **Understanding Hoisting**
5. **Function Execution Context**
6. **Summary and Key Takeaways**

❓ Questions and Answers

1. What is an Execution Context in JavaScript?

Q: What does "Execution Context" mean in JavaScript?

A: An Execution Context is the environment in which JavaScript code is evaluated and executed. It encompasses the variable environment (memory component) and the thread of execution (code component).

Analogy: Think of an Execution Context as a workspace where all the tools (variables and functions) are laid out, and tasks (code) are carried out step by step.

---

2. What are the Phases of Execution Context?

Q: What happens during the two phases of an Execution Context?

A: The Execution Context operates in two phases:

- **Memory Creation Phase:** JavaScript scans the code, allocates memory for variables and functions, and initializes variables with `undefined`. Functions are stored with their code.
- **Code Execution Phase:** JavaScript executes the code line by line, assigning values to variables and invoking functions.

Example:

```
var x = 5;
function greet() {
  console.log("Hello");
}
```

- **Memory Creation Phase:** `x` is allocated with `undefined`, and `greet` is stored with its function code.
- **Code Execution Phase:** `x` is assigned the value `5`, and `greet` is ready to be invoked.

---

3. How does the Call Stack work?

Q: What is the Call Stack, and how does it function in JavaScript?

A: The Call Stack is a data structure that keeps track of function calls. It operates on a Last In, First Out (LIFO) principle: the last function called is the first one to finish. When a function is called, its Execution Context is pushed onto the stack; when it finishes, it's popped off.

Analogy: Imagine a stack of plates in a cafeteria. The last plate added is the first one to be removed. Similarly, the last function called is the first one to complete execution.

---

4. What is Hoisting in JavaScript?

Q: How does hoisting affect variable and function declarations?

A: Hoisting is JavaScript's behavior of moving declarations to the top of their scope during the compile phase. This means variables and functions can be used before they are declared.

Example:

```
console.log(a); // undefined
var a = 10;
```

In this example, `a` is hoisted and initialized with `undefined`, so the `console.log` doesn't throw an error.

---

5. How does Function Execution Context work?

Q: What happens when a function is invoked?

A: When a function is called, a new Execution Context is created for that function. This context has its own memory and code components. It goes through the same two phases: memory creation and code execution.

Example:

```
function square(num) {
  return num * num;
}
```

When `square(4)` is called, a new Execution Context is created for `square`, and the code inside it is executed.

🔑 Summary and Key Takeaways

- **Execution Context:** The environment where JavaScript code is executed, consisting of the memory and code components.
- **Two Phases:** Memory Creation (allocating memory and initializing variables) and Code Execution (executing code line by line).
- **Call Stack:** A stack data structure that manages the execution order of function calls.
- **Hoisting:** JavaScript's behavior of moving declarations to the top, allowing variables and functions to be used before they are declared.
- **Function Execution Context:** Each function call creates a new Execution Context with its own memory and code components.