JS code Execution

🚀 Introduction: How JavaScript Works Behind the Scenes

Ever wondered how JavaScript executes your code? 🤔

- Is it **synchronous** or **asynchronous**?
- Is it **single-threaded** or **multi-threaded**?
- Let's break it down step by step!

🔄 JavaScript is Synchronous & Single-Threaded

- **Single-threaded** = Only **one task at a time** (no multitasking).
- **Synchronous** = Tasks run **in order** (next line waits for current line to finish).

Example:

```
console.log("First");
console.log("Second"); // Runs ONLY after "First" finishes.
```

Wait, what about `AJAX` (Async JS)? 😉

- Remember: **JS is synchronous by default**.

🚀 What Happens When You Run JavaScript Code?

When a JS program runs, the **JavaScript Engine** creates an **Execution Context** to manage everything.

> "Everything in JavaScript happens inside the Execution Context."

🔢 Two Phases of Execution Context

Phase 1: Memory Creation Phase

- JS allocates memory to all variables/functions.
- **Variables:** Initialized with `undefined`.
- **Functions:** Store the **entire function code**.

Example:

```
var n = 2;
function square (num){
   var ans = num * num;
   return ans;
}
var square2 = square(n);
var square4 = square(4);
```

```
// Before execution (Phase 1)
var n = 2; // n = undefined
function square(num) { ... } // square = function code
var square2 = square(n); // square2 = undefined
var square4 = square(4); // square2 = undefined
```

- *(Think of it like a brain storing info!)*

Phase 2: Code Execution Phase

- JS runs the code line-by-line and **updates values**.
- **Variables:** Replace `undefined` with actual values.
- **Functions:** Invoke when called.

Example:

```
// After execution (Phase 2)
n = 2; // Now n = 2
```

```
square2 = square(2); // Invokes square(), returns 4
square4 = square(4); // Invokes square(), returns 16
```

- *(Like a chef following a recipe step-by-step!)*

🚀 Function Invocation & Variable Environment

1. Execution Context Creation

- Each function call creates a **new execution context** with:
  - **Memory Phase:** Variables are initialized as `undefined`, and functions are stored fully.
  - **Execution Phase:** Code runs line-by-line.

```
function greet() {
  console.log(x); // undefined (hoisted)
  var x = 10;
}
greet(); // New execution context created
```

2. Variable Environment

- **Local Scope:** Variables inside a function are isolated.
- **Global Scope:** Accessible everywhere.

```
let globalVar = "🌍";

function checkScope() {
  let localVar = "🏠";
  console.log(globalVar); // "🌍" (accessible)
}
console.log(localVar); // ReferenceError (not defined)
```

- If any variable is needed inside a function, first consider inside the local scope, then the global scope.

```
var x = 1;
a();
console.log(x); // 1

function a(){
   var x = 10;
   console.log(x); // 10
}
```

Key Points:

- **Parameters vs. Arguments:**
  - `num` (parameter) gets its value from `n` (argument).
- **Return Statement:**
  - Exits the function and returns the value to the caller.
  - The function's execution context is **deleted** after `return`.

📚 Call Stack (Execution Context Stack)

- A **stack** data structure that manages execution contexts.
- **How it works:**
  1. **Global Execution Context (GEC)** is pushed first.
  2. When a function is called, its **EC is pushed** on top.
  3. When a function finishes, its **EC is popped** off.

Example:

```
Call Stack:
1. square(4) (function/new) EC  ➔ [Top]
```

```
2. Global EC    → [Bottom]
```

- After the return statement, the function EC popped out from stack.
- **Other Names:** Execution Context Stack, Program Stack, Control Stack, Runtime Stack, Machine Stack.

🔁 Recap: How JS Code Runs

1. **Global Execution Context** is created.
2. **Memory Allocation Phase:**
   - Variables = `undefined`, Functions = stored code.
3. **Code Execution Phase:**
   - Variables get values.
   - Functions create new ECs when invoked.
4. **Call Stack** manages the order of execution.

Visualization:

```
Phase 1 (Memory):
  n: undefined → Phase 2 (Code): n = 2
  square: function code → Invoked → New EC created
```

💡 Key Takeaways

- No execution context → No JavaScript!
- **Execution Context = Memory + Code Components.**
- **Functions** create mini-programs (new ECs).
- **Call Stack** ensures orderly execution.
- **JS is Synchronous & Single-Threaded** (one command at a time).
