Execution context and call stack

Let's break down two of the most fundamental concepts in JavaScript's runtime behavior:

🔄 Execution Context

✅ What is it?

An **execution context** is an environment where JavaScript code is evaluated and executed. It defines what data is available and how the code is run.

🔧 Types of Execution Contexts

1. **Global Execution Context (GEC)**
   - Created by default when a JavaScript program starts.
   - Contains global variables and functions.
   - In browsers, it's the `window` object.
   - Only one global context exists.
2. **Function Execution Context (FEC)**
   - Created every time a function is invoked.
   - Each function has its own execution context.
3. **Eval Execution Context**
   - Created inside the `eval()` function (rarely used and not recommended).

🧠 Execution Context Lifecycle

1. **Creation Phase**
   - Scope is set up.
   - Variable and function declarations are hoisted.

- `this`, `arguments`, and lexical environment are established.

2. **Execution Phase**
   - Code is executed line-by-line.
   - Variable assignments and function calls are made.

🛠️ Example:

```
let x = 10;
function show() {
  let y = 20;
  console.log(x + y);
}
show();
```

Execution steps:

1. Global Context is created:
   - `x` is declared and assigned `10`
   - `show` function is hoisted
2. When `show()` is called:
   - A new **Function Execution Context** is created
   - `y = 20` is stored in local memory
   - `console.log(x + y)` → `10 + 20 = 30`

📚 Call Stack

✅ What is it?

The **call stack** is a data structure that keeps track of function calls in your code. It follows **LIFO** (Last In, First Out).

🧠 How it works:

1. Global execution context is pushed onto the stack.
2. Each time a function is called, a new context is pushed.
3. When a function finishes, its context is popped from the stack.

🛠️ Example:

```
function one() {
  console.log("One");
  two();
}
function two() {
  console.log("Two");
  three();
}
function three() {
  console.log("Three");
}
one();
```

📦 Call Stack Behavior:

1. `one()` → pushed
2. `console.log("One")` runs
3. `two()` → pushed
4. `console.log("Two")` runs
5. `three()` → pushed
6. `console.log("Three")` runs
7. `three()` completes → popped
8. `two()` completes → popped
9. `one()` completes → popped

❌ Stack Overflow

If too many contexts are pushed without being popped (like in infinite recursion), you get a **stack overflow**:

```
function recurse() {
  recurse();
}
recurse(); // ❌ Maximum call stack size exceeded
```

✅ Summary

Concept	Description
Execution Context	The "environment" where JS runs code.
Call Stack	The structure that tracks the order of function execution.