10th January - JavaScript Basics

- Topic 1: What is JavaScript Runtime?
  - JavaScript Engine
    - Memory Heap
    - Call Stack
  - Call Stack Execution
  - Browser Web APIs
  - Task Queues
    - Microtask Queue
    - Macrotask Queue
  - Event Loop
  - Complete Execution Example
  - Key Takeaways
  - How is Event Loop different from JS Engine (V8) ?
  - Why doesn't the JS Engine include the Event Loop?
  - Summary
- Topic 2: Execution context
  - What is an Execution Context?
  - Types of Execution Context
  - Global Execution Context (GEC)
    - What is Global Execution Context?
    - What does GEC contain?
    - 1️⃣ Memory Creation Phase (Creation Phase)
    - 2️⃣ Code Execution Phase
    - Example
  - Function Execution Context (FEC)
    - What is Function Execution Context?
    - What does FEC contain?
    - 1️⃣ Memory Creation Phase
    - 2️⃣ Code Execution Phase
    - Example
  - Execution Context & Call Stack Relationship

- How they work together
- Example
- Important Clarifications (Very Interview-Relevant)
- Common Misconceptions (Clear These Explicitly)
- Quick Summary
- What is it physically?
  - Physically, an execution context is:
- Where does it live?
  - In memory, not in JavaScript land
- How the call stack fits physically
- One-Line Wrap-Up

Scope of Session 1: JavaScript Execution, Scope & Hoisting

Topic 1: What is JavaScript Runtime?

JavaScript Runtime is the environment in which JavaScript code runs.

In a browser, the JavaScript runtime is made up of:

- **JavaScript Engine**
- **Browser Capabilities**
  - **Browser Web APIs**
  - **Queues**
  - **Event Loop**

JavaScript by itself is:

- Single-threaded
- Synchronous by default

Asynchronous behavior comes from the **browser runtime**, not from JavaScript itself.

JavaScript Engine

The JavaScript Engine is responsible for executing JavaScript code.

Ex: **V8** (**Chrome** browser), JavaScriptCore (JSC) (**Safari**), **Chakra** (Legacy browsers like internet explorer)

It contains two main components:

Memory Heap

- Stores variables, objects, and functions

Call Stack

- Executes JavaScript code
- Follows **Last In, First Out (LIFO)** order
- Only one piece of code executes at a time

Call Stack Execution

The call stack manages function execution order.

Example:

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

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

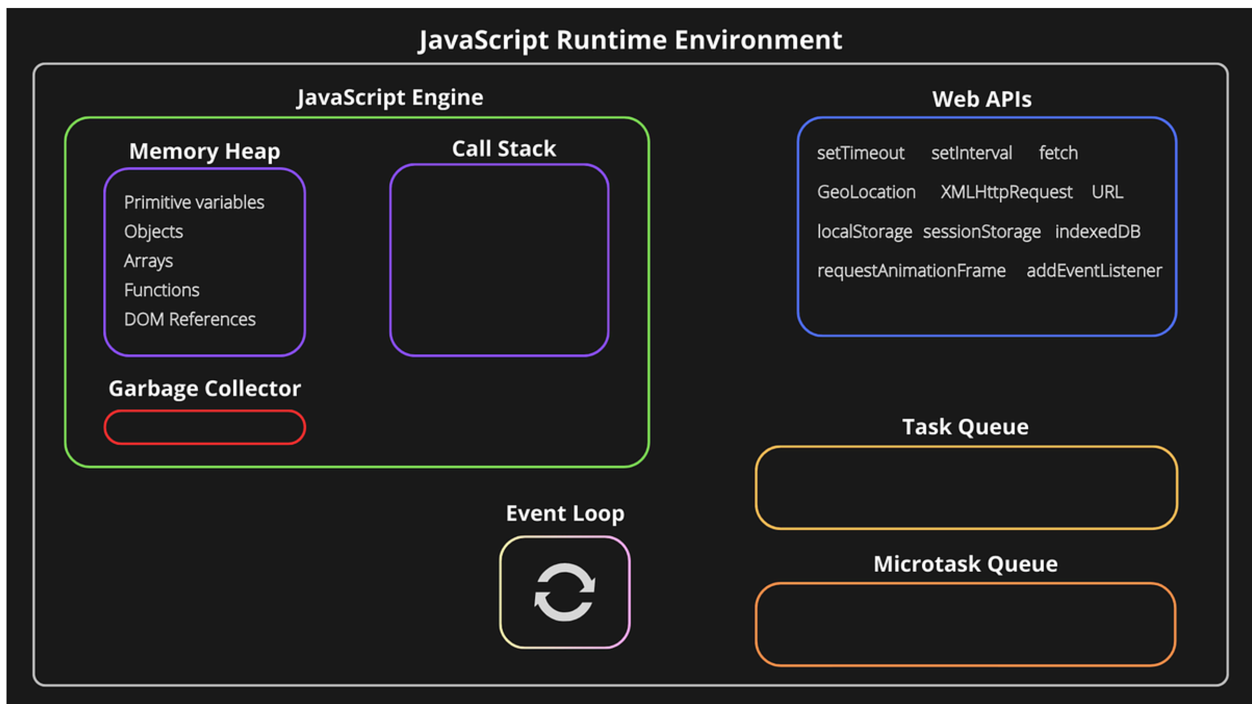
```
a();
```

Execution flow:

- `a()` is pushed onto the stack
- `b()` is pushed onto the stack
- `console.log()` executes
- Functions complete and are popped off the stack

If the call stack is blocked, no other JavaScript code can execute.

Browser Web APIs



Browsers provide additional capabilities to JavaScript through Web APIs.

Common Web APIs include:

- `setTimeout`
- `setInterval`

- `fetch`
- DOM events
- Geolocation
- Web Storage

JavaScript delegates asynchronous work to these Web APIs and continues execution.

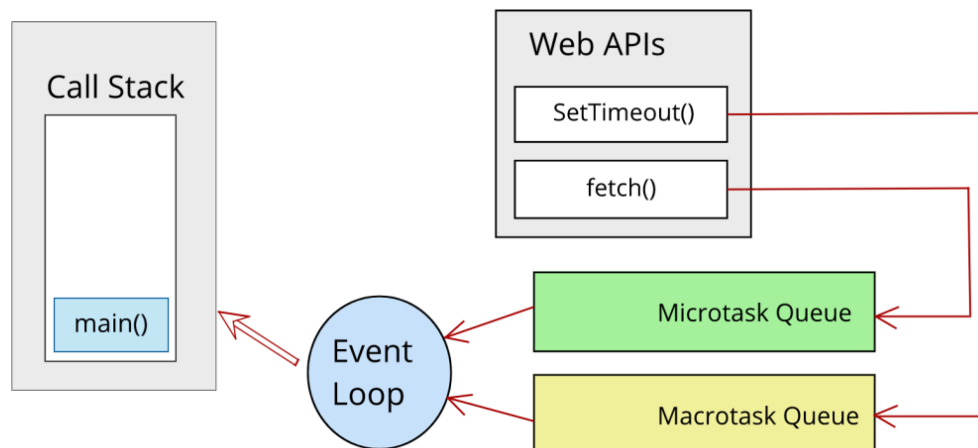
Example:

```
setTimeout(() => {
    console.log("Hello");
},2000);
```

Execution flow:

- JavaScript registers the timer
- Browser Web API handles the timer
- JavaScript continues running without waiting

Task Queues



When Web APIs complete their work, callbacks are placed into queues.

Microtask Queue

- Contains:
    - `Promise.then`
    - `async/await`
    - `queueMicrotask` (schedules a specified function to be executed as a microtask)
    - `MutationObserver` (is a built-in JavaScript Web API that provides the ability to watch for changes being made to the DOM tree and execute a specified callback function when those changes occur)

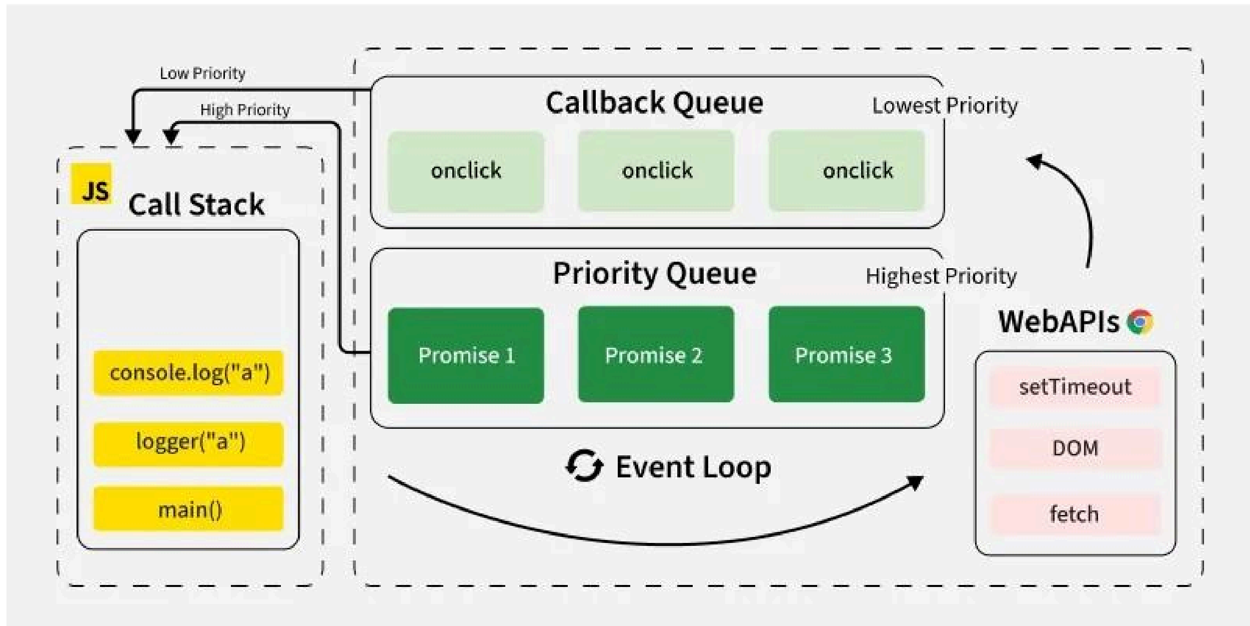
Macrotask Queue

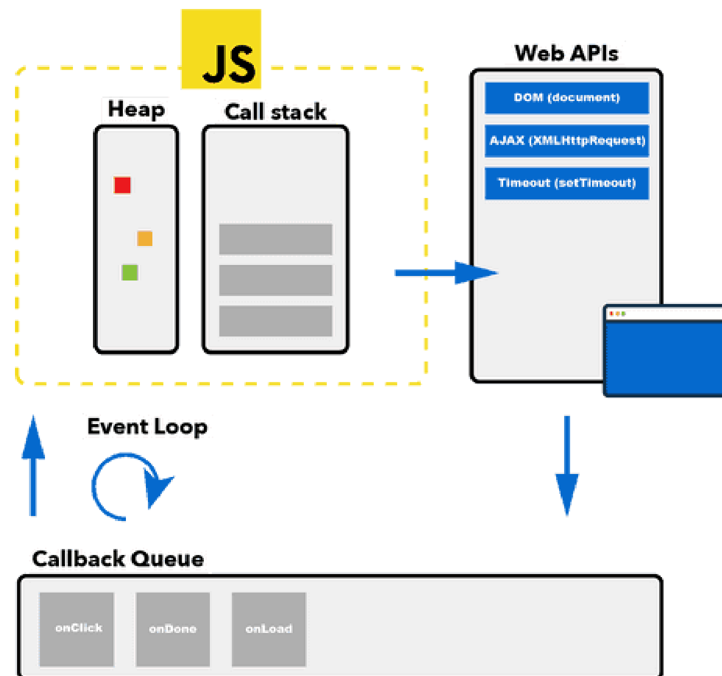
- Lower priority queue
- Contains:
    - `setTimeout`

- `setInterval`
- DOM events

Microtasks are always executed before macrotasks.

Event Loop





The event loop continuously checks the call stack and queues.

Execution order:

1. **Call stack must be empty**
2. **Execute all microtasks**
3. **Execute one macrotask**
4. **Repeat the cycle**

The event loop coordinates execution between the call stack and queues.

Complete Execution Example

```
console.log("A");

setTimeout(() => {
  console.log("B");
```

```
},0);

Promise.resolve().then(() => {
   console.log("C");
});

console.log("D");
```

Execution order:

- `A` is logged
- `setTimeout` callback is registered with Web API
- Promise callback is added to microtask queue
- `D` is logged
- Call stack becomes empty
- Microtask queue executes → `C`
- Macrotask queue executes → `B`

Final Output:

```
A
D
C
B
```

Key Takeaways

- JavaScript executes code using a single call stack
- Asynchronous operations are handled by browser Web APIs
- Microtasks have higher priority than macrotasks
- Event loop decides execution order
- `setTimeout(0)` does not mean immediate execution

How is Event Loop different from JS Engine (V8) ?

"The event loop is a browser-side scheduler that coordinates when the JavaScript engine can execute queued callbacks."

Aspect	JS Engine (V8)	Event Loop
Location	Inside JS runtime engine	Inside browser runtime
Language	C++ (executes JS semantics)	Native browser code. Ex: C / C++
Responsibility	Executes JavaScript	Schedules JavaScript
Owns Call Stack	✅ Yes	❌ No
Executes JS	✅ Yes	❌ No
Controls timing	❌ No	✅ Yes

Why doesn't the JS Engine include the Event Loop?

Because **JavaScript is designed to be host-agnostic**.

The same JS engine can run in:

- Browser
- Node.js
- Deno
- Embedded systems

Each environment provides:

- Its own APIs
- Its own event loop implementation

Example:

- Browser Event Loop ≠ Node.js Event Loop
- V8 is reused, but event loop is replaced

Summary

JavaScript is single-threaded and synchronous by default.

The browser runtime enables asynchronous behavior using Web APIs, task queues, and the event loop.

Topic 2: Execution context

What is an Execution Context?

> Execution Context is the environment in which JavaScript code is evaluated and executed.

Every time JavaScript runs code, it does so **inside an execution context**.

At any moment:

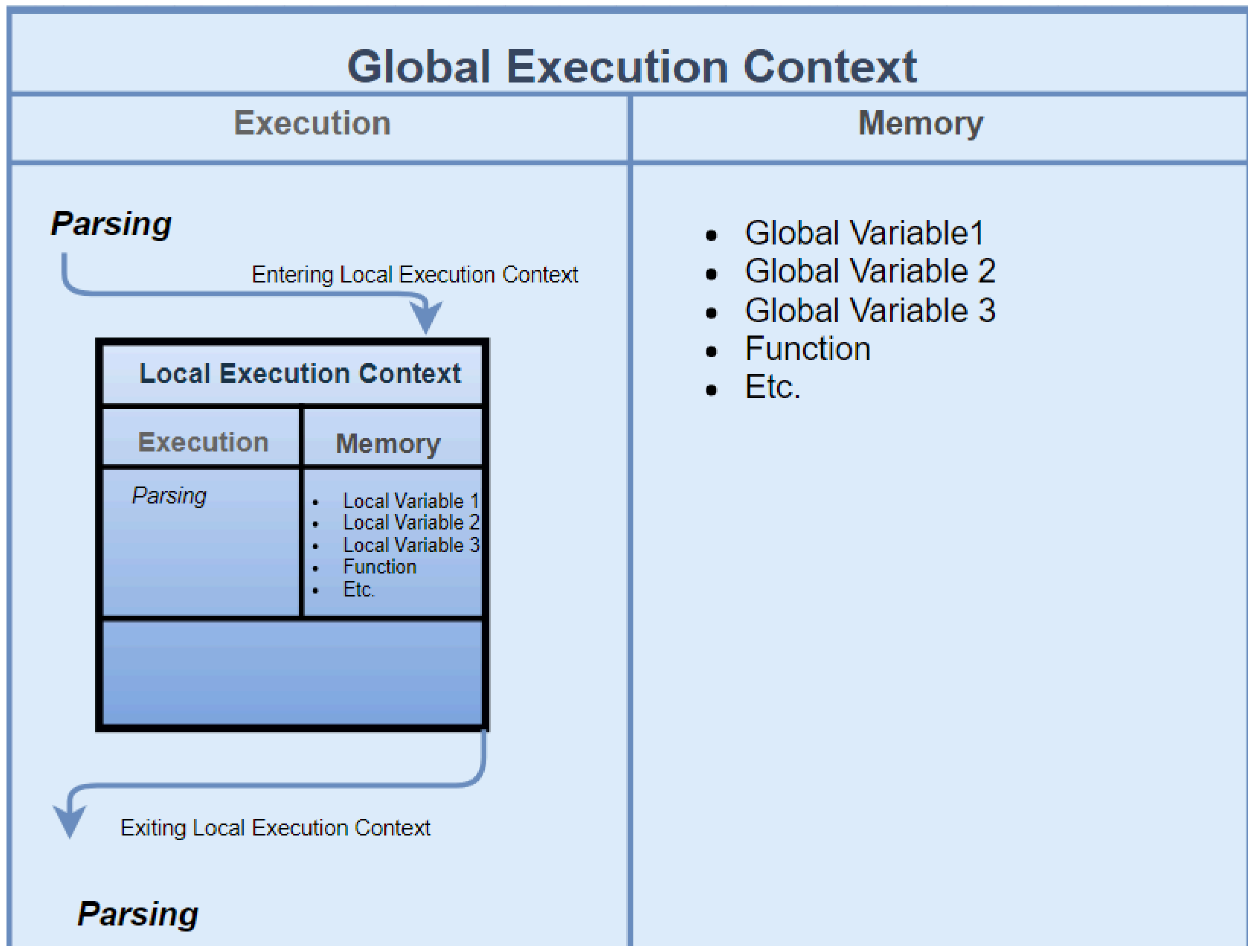
- Only **one execution context is active**
- It always sits on top of the **Call Stack**

Types of Execution Context

JavaScript has **two main execution contexts**:

1. **Global Execution Context (GEC)**
2. **Function Execution Context (FEC)**

Global Execution Context (GEC)



What is Global Execution Context?

- Created **once** when the JavaScript file starts executing. (Files are just sources of code, not runtime boundaries.)
- Represents code **not inside any function**
- Sits at the **bottom of the call stack**

What does GEC contain?

The Global Execution Context has **two phases**:

1️⃣ Memory Creation Phase (Creation Phase)

Before executing a single line of code, JavaScript:

- Allocates memory
- Sets up scope

During this phase:

- `var` variables → initialized as `undefined`
- Function declarations → fully hoisted
- `let` & `const` → allocated but **not initialized**
- `this` → points to the global object (browser: `window`)

2️⃣ Code Execution Phase

- JavaScript executes code **line by line**
- Variables get actual values
- Functions are invoked

Example

```
var a = 10;

function foo() {
    console.log("Hello");
}

foo();

const test = () => {
  console.log("test")
}

test()
```

Memory Creation Phase:

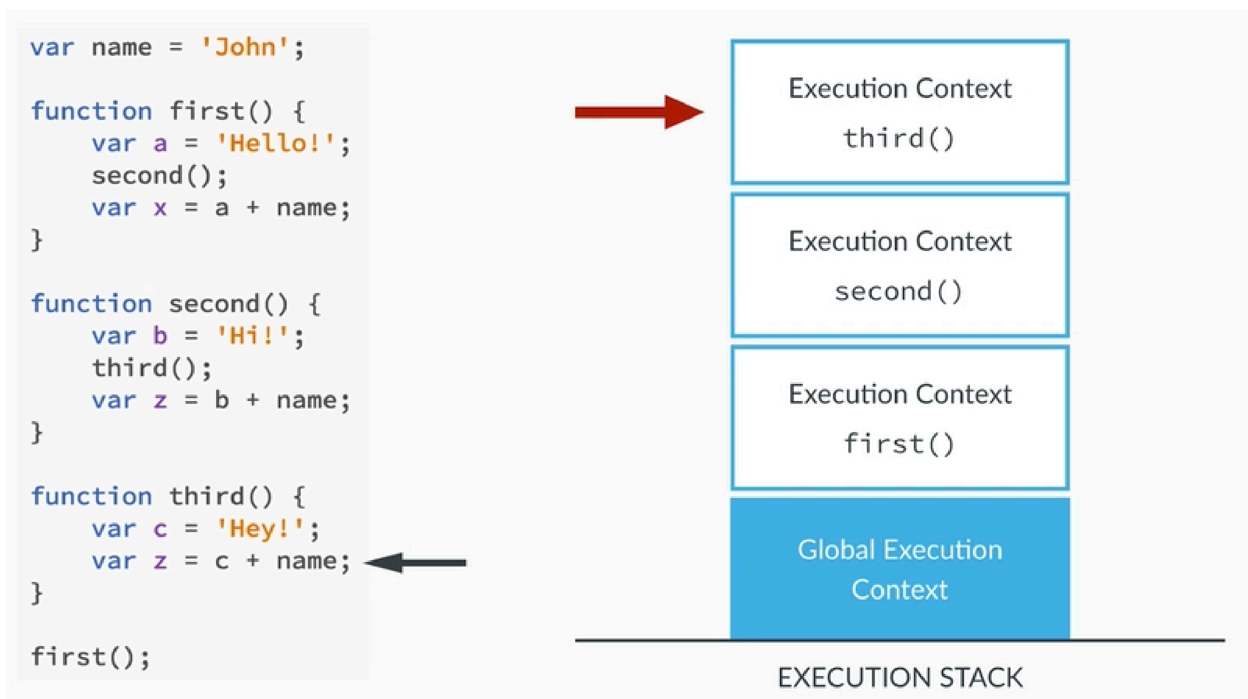
- `a` → `undefined`

- `foo` → function reference

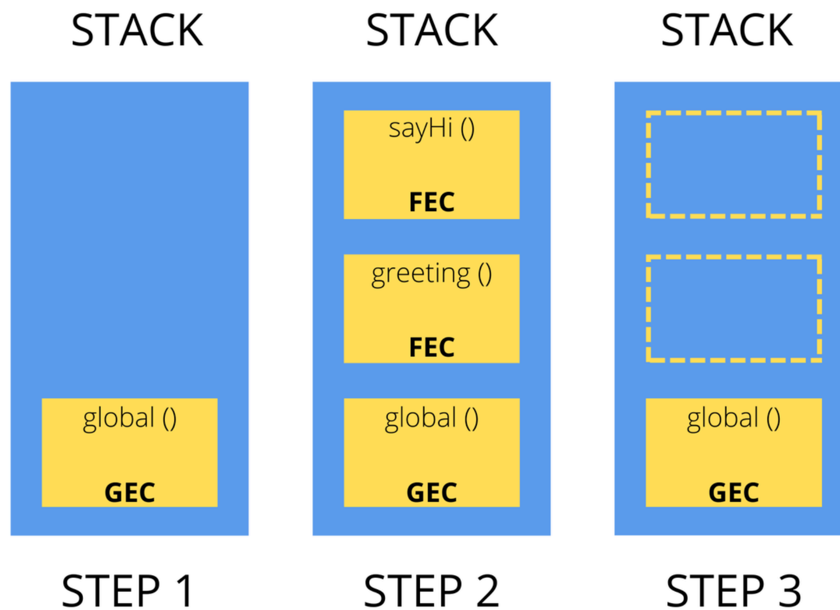
Execution Phase:

- `a` → `10`
- `foo()` is invoked

Function Execution Context (FEC)




```
function greetings() {
    console.log("Greetings");
    sayHi();
}

function sayHi() {
    console.log("Hi");
}

greetings();
```



What is Function Execution Context?

- Created **every time a function is called**
- Each function call gets a **new execution context**
- Pushed onto the **call stack**
- Destroyed after function finishes execution

What does FEC contain?

Each Function Execution Context also has **two phases**:

1️⃣ Memory Creation Phase

- Function arguments are assigned
- Local variables are allocated
- Inner function declarations are hoisted
- `this` is determined (based on how function is called)

2️⃣ Code Execution Phase

- Function body executes line by line
- Values are assigned
- Nested function calls create more execution contexts

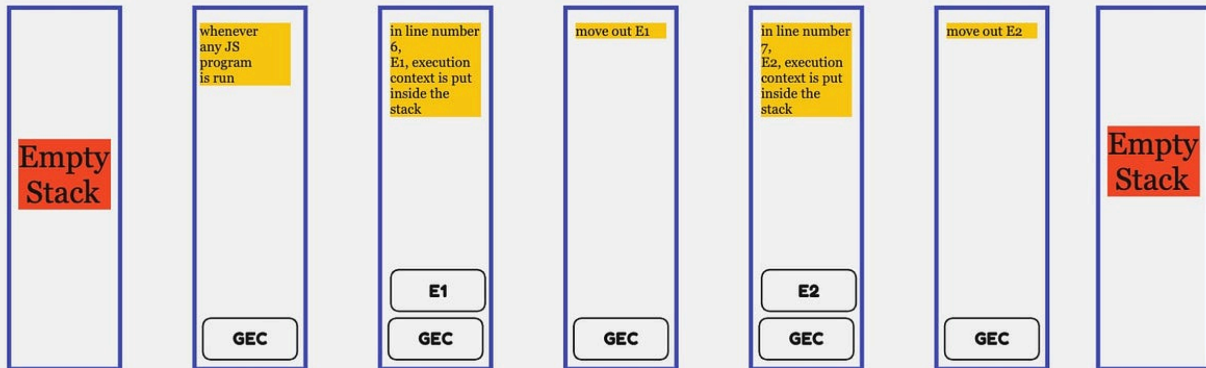
Example

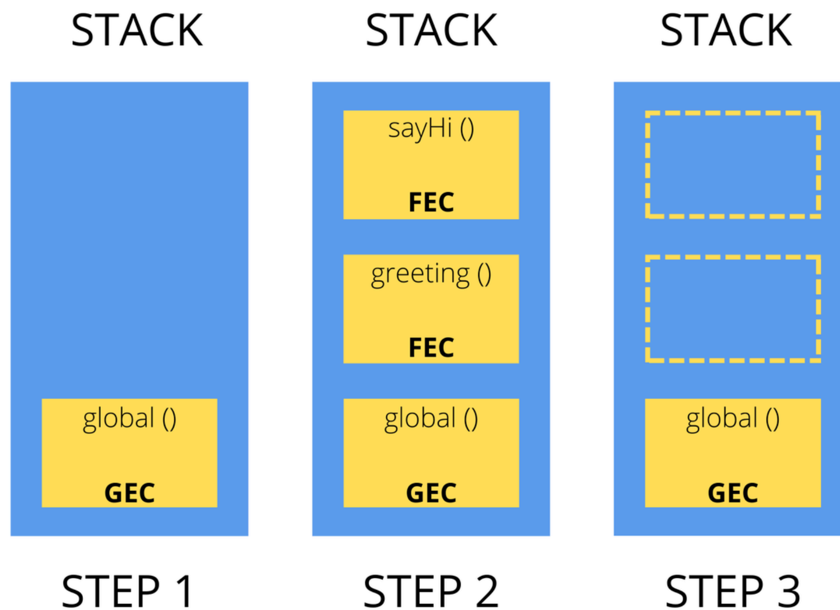
```
function add(x, y) {
    var result = x + y;
    return result;
}

add(2,3);
```

Inside `add` execution context:

- `x` → `2`
- `y` → `3`
- `result` → `undefined` (creation phase)
- `result` → `5` (execution phase)

Execution Context & Call Stack Relationship





How they work together

- Global Execution Context is pushed first
- Each function call:
  - Creates a new Function Execution Context
  - Pushes it onto the call stack

- When function returns:
  - Its execution context is popped off the stack

Example

```
function one() {
  two();
}

function two() {
  three();
}

function three() {
  console.log("Done");
}

one();
```

Call Stack flow:

1. Global Execution Context
2. `one()` Execution Context
3. `two()` Execution Context
4. `three()` Execution Context
5. `three()` completes → popped
6. `two()` completes → popped
7. `one()` completes → popped

Important Clarifications (Very Interview-Relevant)

- Execution Context ≠ Call Stack
  - Execution context is **what**

    - Call stack is **where**

- Only **one execution context executes at a time**
- Execution contexts are **created synchronously**
- Async callbacks create **new execution contexts later**

Common Misconceptions (Clear These Explicitly)

❌ "Execution context is created for every line"

✅ Created only for **global code and function calls**

❌ "Async code runs in the same execution context"

✅ Async callbacks run in **new execution contexts**

❌ "Hoisting happens during execution"

✅ Hoisting happens during **memory creation phase**

Quick Summary

- JavaScript runs code inside **execution contexts**
- Global Execution Context is created first
- Each function call creates a new Function Execution Context
- Execution contexts are managed using the call stack
- Each execution context has:
    - Memory Creation Phase
    - Code Execution Phase

What is it physically?

Physically, an execution context is:

- A **C++ object / struct**

- Allocated in **native memory** (heap)
- Created and destroyed by the **JS engine (e.g., V8)**
- Referenced by the **call stack**

You can think of it as:

```
// Conceptual (NOT real V8 code)
struct ExecutionContext {
LexicalEnvironment* lexicalEnv;
VariableEnvironment* variableEnv;
ThisBinding thisValue;
CodePointer* instructionPointer;
};
```

📌 This is not the real implementation, but it is **conceptually accurate**.

Where does it live?

In memory, not in JavaScript land

- Execution contexts live in **engine-managed memory**
- Not accessible from JavaScript
- Not visible in DevTools
- Exist only while code is executing

The **Call Stack** is essentially:

> A stack of pointers to these execution context objects.

How the call stack fits physically

Visualize this:

```
Call Stack (native memory)
┌──────────────────────────┐
│ Pointer → EC forfoo()    │
│ Pointer → EC forbar()    │
│ Pointer → Global EC      │
└──────────────────────────┘
```

Each stack frame points to:

- One execution context object
- Which contains references to scopes, variables, and instructions

One-Line Wrap-Up

> “Execution context explains where variables live and how JavaScript executes code step by step.”