How Javascript Works? - Call Stack, Event Loop and Queues Explained

🧱 1. JavaScript is Single-Threaded

JavaScript runs one command at a time in a single sequence.

- It reads code **from top to bottom**, **one line at a time**.
- You can't do two things at once (like Python's threading or Java's multithreading).
- It uses a **Call Stack** to manage the code execution.

🗃️ 2. Call Stack — The Main Executor

Think of the call stack as a stack of plates:

- **Last In, First Out (LIFO)**
- The most recent function pushed on the stack is the first one to be popped off.

Example:

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

sayHello();
```

What happens:

Call Stack	Action
`sayHello()`	Pushed

`console.log("Hello")`	Pushed → Executes
`console.log`	Pops after printing
`sayHello()`	Pops

👉 Once stack is **empty**, JavaScript has nothing more to run.

⏰ 3. setTimeout — Delayed Execution

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

setTimeout(() => {
  console.log("Inside Timeout");
}, 2000);

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

Output:

```
Start
End
Inside Timeout
```

> Even though we wrote setTimeout(..., 2000), it doesn't block the next lines. JavaScript moves on immediately.

⚙️ 4. What Happens Internally

Here's what happens when you run the code:

📊 Architecture:

```
+----------------------+
|  Your JavaScript  |
+---------------------+
        │
        v
+--------------------+      +-----------------------+
│   Call Stack      │       │     Web APIs        │
│ (Runs Sync Code)  │       │  (Timer, DOM, etc.)   │
+--------------------+      +-----------------------+
        │                    │
        v                    v
+--------------------+      +-----------------------+
│ Microtask Queue    │       │     Task Queue      │
│ (Promises, etc.)   │       │ (setTimeout, etc.)    │
+--------------------+      +-----------------------+
        │
        v
+--------------------+
│    Event Loop      │ ⟵── Keeps checking these queues
+--------------------+
```

🧪 5. Code Execution Example

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

setTimeout(() => {
  console.log("Timeout");
}, 0);

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

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

📝 Step-by-step Execution:

Step	What Happens	Where It Goes
1	`console.log("Start")`	Call Stack
2	`setTimeout(..., 0)`	Web API → Task Queue
3	`Promise.resolve().then(...)`	Microtask Queue
4	`console.log("End")`	Call Stack
5	Call Stack is now EMPTY	Event Loop wakes up
6	Microtask Queue has callback → Runs it	Prints `Promise`
7	Then Task Queue has callback → Runs it	Prints `Timeout`

🧾 Final Output:

```
Start
End
Promise
Timeout
```

🎯 6. Microtask Queue vs Task Queue

Feature	Microtask Queue	Task Queue
Priority	Higher	Lower
Example APIs	`Promise.then` , `MutationObserver`	`setTimeout` , `setInterval`
Execution Timing	Right after current task ends	After all microtasks finish

⚠️ 7. Starvation Problem

When **microtasks never stop**, **task queue** items (like `setTimeout`) get **starved**.

Example:

```
function repeat() {
  Promise.resolve().then(() => {
    console.log("Microtask");
    repeat(); // Keeps pushing microtasks
  });
}

setTimeout(() => {
  console.log("From setTimeout");
}, 0);

repeat();
```

Output:

```
Microtask
Microtask
Microtask
...(forever)...
```

☠️ `From setTimeout` will **never run.**

🧩 8. Full Visual Example

Code:

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

setTimeout(() => console.log("B"), 0);

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

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

Step-by-Step:

Step	Item	Location	Printed?
1	`console.log("A")`	Call Stack	✅ A
2	`setTimeout(...)`	Web API → Task Queue	❌
3	`Promise.resolve().then(...)`	Microtask Queue	❌
4	`console.log("D")`	Call Stack	✅ D
5	Microtask `console.log("C")`	Microtask → Call Stack	✅ C
6	Task `console.log("B")`	Task Queue → Call Stack	✅ B

✅ Final Output:

```
A
D
C
B
```

🧠 Summary — How JavaScript Executes

```
1. All your code enters the call stack one by one.
2. Async functions (like setTimeout) go to Web API.
3. Promise callbacks go to Microtask Queue.
4. Event Loop:
  - Checks if call stack is empty.
  - Runs Microtasks first.
  - Then runs Tasks (like setTimeout).
```

✅ Memory Tip:

Concept	Think Like
Call Stack	Stack of plates (LIFO)
Web API	Waiter holding delayed tasks
Microtask Q	VIP line (Promise callbacks)

Task Queue	Regular line (Timers, Events)
Event Loop	The doorman — lets tasks in