Day 5 — Event Loop | (Foundations)

---

1. JavaScript Runtime Overview

JavaScript is **single-threaded**, meaning it can execute one piece of code at a time. To manage asynchronous operations, JavaScript relies on the **Event Loop**, which coordinates tasks between different queues.

---

2. Call Stack

The **Call Stack** is where JavaScript runs functions in a LIFO (Last In, First Out) order.

Example:

```
function a() { console.log("A"); }
function b() { a(); }
function c() { b(); }

c();
```

Execution order:

```
c()
b()
a()
```

---

3. Memory Heap

The **Heap** is a large memory region for storing objects, arrays, and functions.

Example:

```
const user = { name: "Bruce" }; // stored in heap
```

---

4. Task Queue (Macrotask Queue)

The **Task Queue** contains tasks scheduled to run later.

Examples of tasks:

- `setTimeout`
- `setInterval`
- DOM events
- I/O callbacks

Important: Tasks run **after** all microtasks finish.

---

5. Microtask Queue

The **Microtask Queue** has higher priority than the Task Queue.

Microtasks include:

- `Promise.then`
- `Promise.catch`
- `Promise.finally`
- `queueMicrotask()`

Microtasks always run before tasks.

---

6. Timers (setTimeout)

`setTimeout(fn, 0)` does **not** run immediately.
It schedules `fn` in the **Task Queue**, which executes **after** microtasks.

Example:

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

Timeline:

```
A
B
(timeout later)
```

---

7. Promise Jobs (Microtasks)

Promises schedule microtasks, which run **before any task**, even `setTimeout(0)`.

Example:

```
console.log("1");
setTimeout(() => console.log("timeout"), 0);
Promise.resolve().then(() => console.log("promise"));
console.log("2");
```

Output:

```
1
2
promise
timeout
```

---

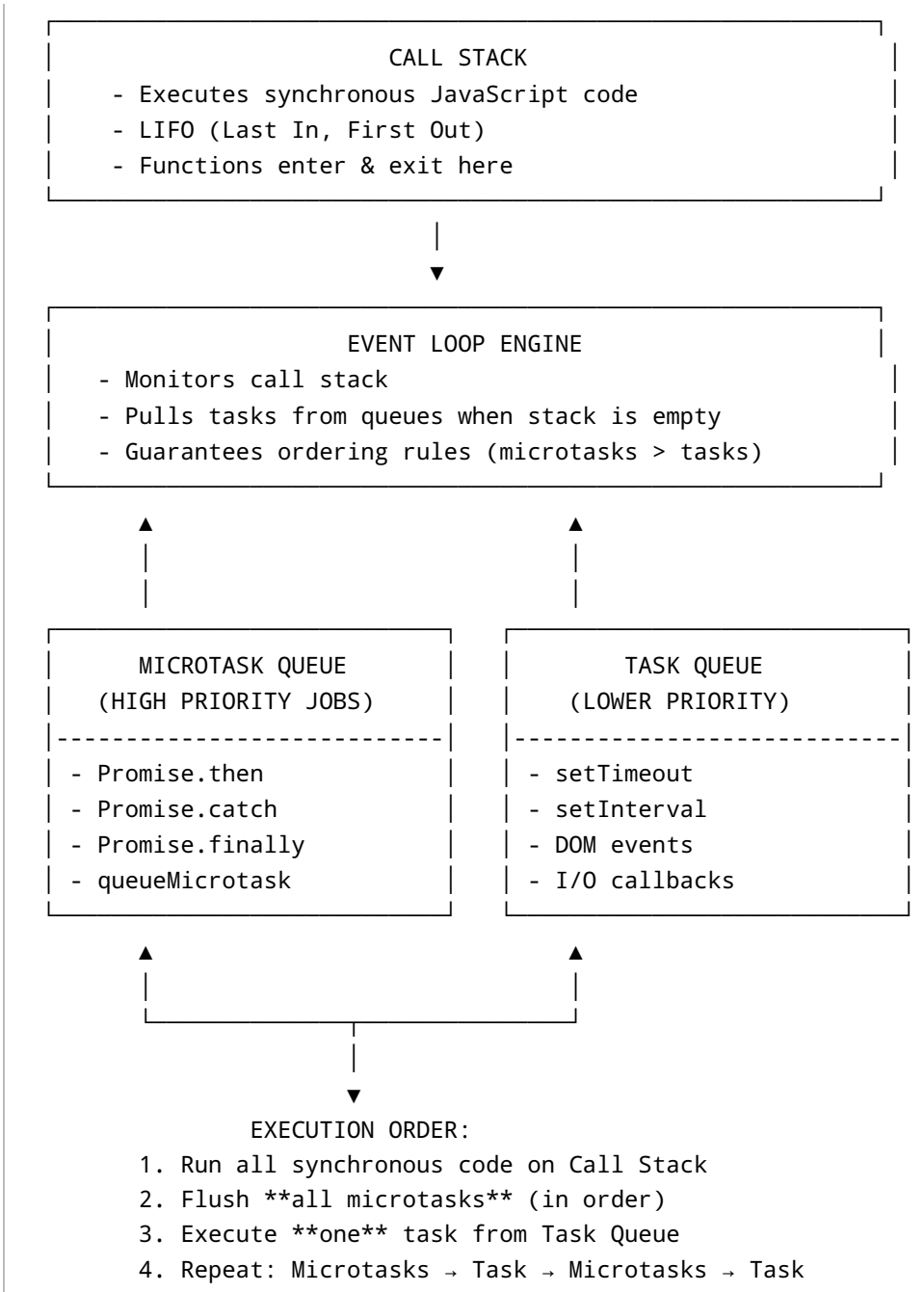
8. Event Loop Flow (Timeline)

Below is an expanded graphical reference showing all major components and how execution flows between them.

High-Level Event Loop Architecture

```
                  JAVASCRIPT RUNTIME
        (Browser / Node.js Environment Layer)

┌──────────────────────────────────────────────────────────┐
│                          HEAP                            │
│   (Stores objects, arrays, functions, closures, etc.)    │
└──────────────────────────────────────────────────────────┘
                             ▲
                             │
                             │
```



What This Diagram Shows

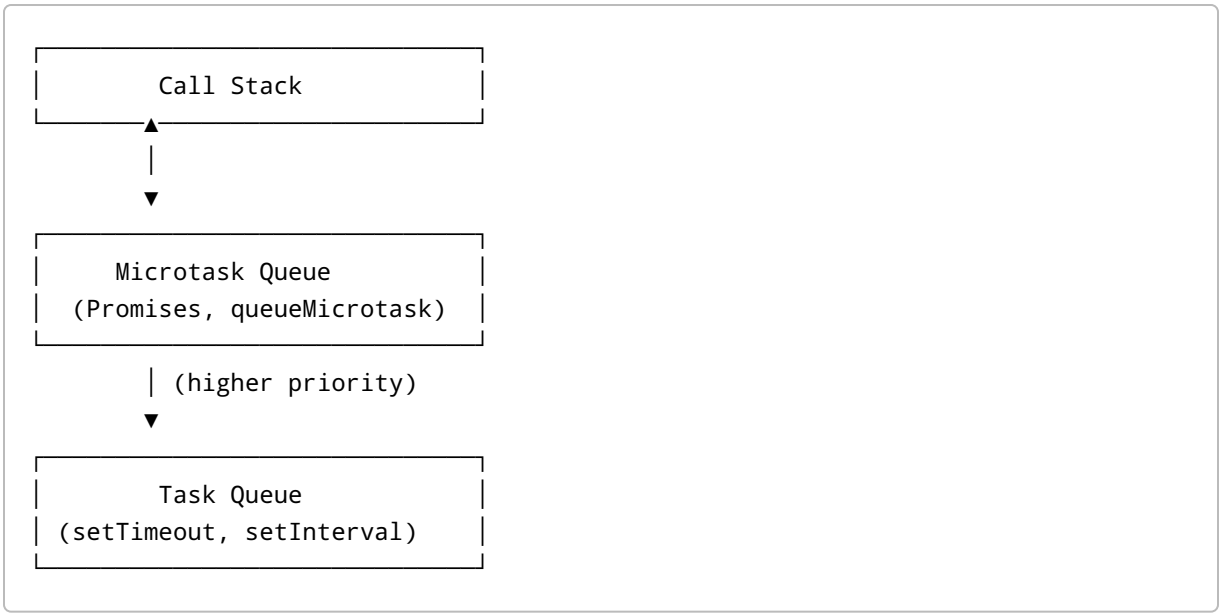
- **Synchronous code** always runs first (Call Stack).
- When the stack becomes empty, the **Event Loop** checks:
- **Microtask Queue** → runs **ALL** microtasks
- **Task Queue** → runs **ONE** task
- Promises (microtasks) always outrun `setTimeout` (tasks).

Visual Timeline of a Typical Execution

```
START EXECUTION
│
├─► Run synchronous code (Call Stack)
│
├─► Microtask Queue:
│       ├─ promise.then
│       ├─ promise.then (chained)
│       └─ queueMicrotask
│
├─► Task Queue:
│       └─ setTimeout callback
│
├─► Microtasks again (if new ones were created)
│
└─► Next Task
```

Key Visualization Insight

> Microtasks are processed in full batches.
> Tasks are processed one at a time.



Execution Order:

1. Run synchronous code
2. Run ALL microtasks
3. Run ONE task

4. Repeat: Microtasks → Task → Microtasks → Task ...

---

9. Predict-the-Output Exercises

Example 1

```
console.log("A");
setTimeout(() => console.log("timeout"), 0);
Promise.resolve().then(() => console.log("promise"));
console.log("B");
```

Output:

```
A
B
promise
timeout
```

Example 2

```
setTimeout(() => console.log("A"));
Promise.resolve().then(() => console.log("B"));
console.log("C");
```

Output:

```
C
B
A
```

Example 3

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

setTimeout(() => {
  console.log("timeout 1");
}, 0);

setTimeout(() => {
  console.log("timeout 2");
```

```
}, 0);

Promise.resolve()
  .then(() => console.log("promise 1"))
  .then(() => console.log("promise 2"));

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

Output:

```
start
end
promise 1
promise 2
timeout 1
timeout 2
```

---

10. Event Loop Timeline Notes

- JavaScript is single-threaded; executes one thing at a time.
- All synchronous code runs first.
- Microtasks always run **before** tasks.
- Promises (microtasks) execute immediately after current stack.
- `setTimeout(0)` does not run at 0ms—it waits until microtasks finish.
- Event Loop cycles: **Stack → Microtasks → Task → Stack → Microtasks → Task**.
- Promise chains (`.then().then()`) run microtask by microtask without interruption.
- Tasks execute one at a time between microtask flushes.