Appendix B: Understanding the Event Loop, Promises, and Microtasks

JavaScript's concurrency model is built around the **event loop**, which allows asynchronous behavior despite JavaScript being single-threaded.

This appendix provides a detailed explanation of:

- The **Call Stack**
- **Web APIs / Background Tasks**
- The **Event Queue**
- The **Microtask Queue**
- And how **Promises** integrate with these mechanisms

🧠 Call Stack

The **call stack** tracks the execution of function calls. It works like a LIFO (Last In, First Out) stack:

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

Execution order:

```
start()
→ greet()
→ console.log()
```

When the current function returns, it's popped off the stack.

🌐 Web APIs / Background Tasks

These are **browser-provided** or **Node.js-provided** APIs that handle operations outside the main thread:

- `setTimeout`, `setInterval`
- `fetch` and network requests
- Event listeners (e.g., `click`)

When such a function is called, it is **offloaded** to the Web API environment. Once complete, it queues a callback in the **Event Queue**.

```
setTimeout(() => console.log("done"), 1000);
```

Here, the delay is tracked by the browser — not by JavaScript itself.

📥 Event Queue (a.k.a. Callback Queue)

This is a FIFO queue that holds callbacks from completed Web API operations. These are **executed only when the call stack is empty**.

Example:

```
console.log("Start");
setTimeout(() => console.log("Timeout"), 0);
console.log("End");
```

Output:

```
Start
End
Timeout
```

⚡ Microtask Queue

The **Microtask Queue** is used for:

- Promise `.then()` callbacks
- `queueMicrotask()`

Microtasks are prioritized **before** anything in the Event Queue.

Example:

```
console.log("Start");
Promise.resolve().then(() => console.log("Promise"));
setTimeout(() => console.log("Timeout"), 0);
console.log("End");
```

Output:

```
Start
End
```

```
Promise
Timeout
```

Promises = Microtasks

Promise callbacks go into the Microtask Queue.

```
Promise.resolve().then(() => console.log("Runs before timeouts"));
```

Deep Function Example:

```
function outer() {
  console.log('outer start');
  inner();
  console.log('outer end');
}

function inner() {
  console.log('inner start');
  Promise.resolve().then(() => console.log('promise in inner'));
  console.log('inner end');
}

outer();
```

Output:

```
outer start
inner start
inner end
outer end
promise in inner
```

The **microtask** (from the promise) waits until the entire call stack finishes.

📌 Summary

Concept	Role in Event Loop
Call Stack	Executes JS code directly
Web APIs	Handle async ops outside the stack
Event Queue	Receives callbacks from Web APIs

Concept	Role in Event Loop
Microtask Queue	Receives Promise and microtask callbacks, runs before Event Queue

Final Note

JavaScript’s single-threaded model works effectively because of:

- Asynchronous behavior
- Event delegation to Web APIs
- Priority of microtasks

Understanding this model is essential for writing performant and predictable JavaScript code.