

Event loop tick explained



what is event loop tick. explain to me

Great question, Vipul — understanding what an **Event Loop tick** is will make a lot of tricky async behavior (Promises, timers, etc.) clear.

Let's break it down **step by step in a simple, visual, and real-world way**.

First, What Is the Event Loop?

The **Event Loop** is the mechanism that allows JavaScript (which is **single-threaded**) to handle:

- asynchronous tasks (like `setTimeout`, Promises, network requests)
- without blocking the main thread.

It does this by continuously checking:

"Do I have something to execute right now?"

"If not, should I pick something from my queues?"

What Is a "Tick"?

Think of one **tick** as **one full cycle (or iteration)** of the event loop.

👉 In simple terms:

A **tick** = one pass where JavaScript executes all tasks it currently has available (from the call stack and queues) before the next waiting task is picked.

Step-by-Step Breakdown of One Tick

Let's go through what happens in **one event loop tick**:

1. Check the Call Stack

- If there's code (like a function call) already running, finish that first.

2. If Call Stack Is Empty:

- The event loop looks at the **macrotask queue** (also called **task queue**) — this includes things like:
 - `setTimeout`
 - `setInterval`
 - `setImmediate` (in Node.js)

- I/O callbacks, etc.

3. Pick the First Macrotask and Execute It

- Example: run the callback of a `setTimeout`.

4. After That Macrotask Finishes, Run All Microtasks

- Microtasks include:
 - `Promise.then()` and `.catch()`
 - `queueMicrotask()`
 - `process.nextTick()` (Node.js only)
- Important: **All microtasks are run before the next tick starts.**

5.  Once all microtasks are done, the tick ends.

6.  The event loop starts the **next tick** and repeats the same steps.

Example: Visualizing Event Loop Ticks

```
js
console.log("A");
setTimeout(() => console.log("B"), 0);
Promise.resolve().then(() => console.log("C"));
console.log("D");
```

Execution Flow (Tick by Tick)

Tick 1:

- **Call Stack:** executes main script line by line.
 - `console.log("A")` → prints A
 - `setTimeout` → schedules "B" in **macrotask queue**
 - `Promise.then` → schedules "C" in **microtask queue**
 - `console.log("D")` → prints D
- Now the **call stack is empty**.
- **Microtask queue:** contains "C" → runs it now → prints c

 **End of Tick 1**

Tick 2:

- Event loop sees "B" in macrotask queue → runs it → prints B

 **End of Tick 2**

Output:

CSS

A
D
C
B

Each of those steps happened across **two ticks**.

Quick Analogy

Imagine the event loop like a waiter (the JS engine) serving food orders (tasks):

- One **tick** = serving **one customer's order** (a macrotask).
- Before the waiter takes the next order, he finishes **small follow-ups** (microtasks) like refilling water or giving the bill.
- Then, he moves to the **next customer** (next macrotask = next tick).

In Node.js — Two Types of Ticks

Node.js adds one more layer:

- **Event Loop tick** → one full cycle of event loop phases.
- **process.nextTick()** → a *special microtask queue* that runs **before** regular microtasks.

So:

SCSS

```
process.nextTick() → runs before Promise microtasks
Promise.then()    → runs after process.nextTick()
setTimeout()      → runs in next event loop tick
```

Summary Table

Type	Executes In	When Runs	Example
Macrotask	Each tick	After microtasks	setTimeout , I/O
Microtask	Inside same tick	Before next macrotask	Promise.then , queueMicrotask
Tick	Event Loop iteration	Runs macrotasks then microtasks	One full cycle

Would you like me to show you a **diagram** (visual representation) of one event loop tick — showing how macrotasks and microtasks flow inside it?

It'll make the concept even clearer.



but microtask runs before macro you said Pick the First Macrotask and Execute It which is wrong i think

Nice catch — you're right to be suspicious, and thanks for calling that out. My wording was ambiguous; let me fix it cleanly.

Short corrected rule (the important part)

- **Microtasks run after the *current* macrotask finishes and *before* the next macrotask begins.** So microtasks don't preempt a running macrotask, but they *do* run before the next macrotask is picked.

I phrased it as "pick a macrotask and execute it" (which is true when the loop selects a macrotask), but that made it sound like macrotasks always run first *within* the same tick — which is misleading without the microtask follow-up step. Sorry for the confusion.

Concrete example (shows the order)

```
js

console.log("start");

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

Promise.resolve()
  .then(() => console.log("promise1"))
  .then(() => console.log("promise2"));

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

Output:

```
sql

start
end
promise1
promise2
timeout
```

Why:

1. The **initial script** runs (the initial script itself is a macrotask). It prints **start**, schedules a macrotask (**setTimeout**), schedules microtasks (the **.then()** handlers), then prints **end**.
2. When the initial macrotask finishes, the event loop runs the **microtask queue** — so **promise1** and then **promise2** run **before** the **setTimeout** callback.
3. Only after all microtasks are drained does the event loop pick the next macrotask (**timeout**) and run it.

Extra note on Node.js

Node has `process.nextTick()` which is a special queue that runs **before** other microtasks (like Promise callbacks). So in Node the relative order is:

1. current macrotask finishes
2. `process.nextTick()` callbacks
3. other microtasks (`Promise.then`, `queueMicrotask`)
4. next macrotask (e.g., `setTimeout`)

TL;DR

- Microtasks run **between** macrotasks: after the currently running macrotask completes and before the next macrotask starts.
- My earlier wording made it sound otherwise — thanks for pointing it out. Want another example (including `process.nextTick`) or a visual diagram?



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

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

Promise.resolve()
  .then(() => console.log("promise1"))
  .then(() => console.log("promise2"));

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

in the above one you said that

When the initial macrotask finishes but first it finishes synchronous than micro than macro than how after initial macro tasks finish line came

Excellent — 🧠 this is *exactly* the kind of detailed thinking you need for mastering the **event loop**. Let's go step-by-step **precisely** and untangle this confusion.



The confusion

You're saying:

"But the script first runs all synchronous code — so where does 'after initial macrotask finishes' come from? Isn't the initial script also synchronous?"

Yes — and that's the key insight.

The **entire initial script itself is treated as the very first macrotask**.

Let's unpack that carefully.

Step-by-step execution (true event loop view)

Step 1: The Event Loop starts

When your JS file is loaded, the **event loop schedules one macrotask** — called the **initial script task**.

So at the start:

- **Macrotask queue** = [`<your script>`]
- **Microtask queue** = []

Step 2: Execute the initial macrotask (your script)

Now the engine starts running your JS file line-by-line.


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

 prints **start**

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

 schedules this callback in the **macrotask queue** for a future tick (not now)

```
js
Promise.resolve()
  .then(() => console.log("promise1"))
  .then(() => console.log("promise2"));
```

 schedules `.then()` callbacks in the **microtask queue**

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

 prints **end**

Step 3: Your script (the initial macrotask) finishes running

Now there's **no synchronous code left** — the **initial macrotask** is complete.

👉 But before starting the next macrotask, the **event loop rule** says:

“Run all pending microtasks before moving on.”

4 Step 4: Execute microtasks

Microtask queue has:

1. `promise1`
2. Then `promise2` (added during `promise1`'s resolution)

So they run one after another:

- prints `promise1`
- prints `promise2`

5 Step 5: Move to the next macrotask

Now the microtask queue is empty.

The event loop moves on and picks the next macrotask, which is your `setTimeout` callback.

It runs:

```
js
console.log("timeout");
```

➔ prints `timeout`

✓ Final Output Order

```
sql
start
end
promise1
promise2
timeout
```

🔄 Summary Timeline

Phase	Type	What runs	Queue(s) affected
1	Macrotask #1	Your script (sync code)	schedules macrotask + microtasks
2	Microtasks	Promise callbacks	drain microtask queue
3	Macrotask #2	<code>setTimeout</code> callback	runs next

💡 So what does “after initial macrotask finishes” mean?

It means:

After your entire script (the initial macrotask) runs all synchronous lines and the call stack is empty, **the event loop finishes that macrotask.**

Then it runs **microtasks** before moving to any further macrotask (like `setTimeout`).

So yes —

- first, synchronous code runs (inside the first macrotask),
- then microtasks,
- then next macrotasks.

Would you like me to show this **as a visual timeline diagram** (with queues and arrows) for this exact example? It's super clear when you see it visually.