# JavaScript learning path

Perfect 🚀
We've compared **JS vs PHP** and **JS vs Java**. Now let's move to the **next natural part**:

👉 **How Browsers Run JavaScript (Step by Step)**

This is a **core foundation**, because unless you know how your code *actually runs under the hood*, you'll miss many optimizations and debugging insights.
We'll go **bit by bit** with proper representation.

---

## 🖥️ How Browser Runs JavaScript

When you open a webpage:

1. **HTML Loading (DOM Creation)**
   - Browser first downloads & parses the **HTML file**.
   - Builds the **DOM Tree** (Document Object Model).
   - Encounters a `<script>` tag → pauses HTML parsing → sends JS to the JS Engine.

---

2. **JavaScript Engine Takes Over**
   Every modern browser has its own **JavaScript engine**:
   - Chrome → **V8**
   - Firefox → **SpiderMonkey**
   - Safari → **JavaScriptCore (Nitro)**
   - Edge → also **V8**

   Inside an engine like **V8**:
   - **Parsing** → Converts JS code into an **AST (Abstract Syntax Tree)**.
   - **Interpreter (Ignition)** → Quickly executes code (line by line).
   - **Compiler (TurboFan JIT)** → Optimizes "hot" code (runs many times) into **machine code**.
   - **Garbage Collector (Orinoco & Oilpan in V8)** → Cleans unused memory.

---

3. **Execution Context & Call Stack**

- JS is **single-threaded** → only **one call stack**.
- Each function call → new **execution context** pushed on stack.
- When function finishes → popped from stack.
- Error like *"Maximum call stack size exceeded"* happens if recursion never ends.

---

4. **Web APIs (provided by Browser)**
   - JavaScript itself does **NOT** have `setTimeout`, `DOM`, `fetch`, `XMLHttpRequest`.
   - These come from **Web APIs** (part of Browser, not JS Engine).
   - Examples:
     - `DOM API` → `document.querySelector`, `getElementById`
     - `Timer API` → `setTimeout`, `setInterval`
     - `AJAX / Fetch API` → server communication
     - `Geolocation API`, `Canvas API` …

---

5. **Event Loop & Callback Queue**
   - JS is single-threaded, but still **asynchronous** thanks to Event Loop.
   - Flow:
     - JS code calls `setTimeout` → browser timer starts → after delay, callback goes into **Callback Queue**.
     - Event Loop checks: "Is Call Stack empty?"
       - If YES → pushes queued callback into stack.
       - If NO → waits.
   - This is why `setTimeout(fn, 0)` still runs *after current code finishes*.

---

6. **Microtask Queue (Promises)**
   - Promises (`.then`, `await`) are **higher priority** than normal callbacks.
   - Two queues:
     - **Microtask Queue** → Promises, MutationObserver (runs first).
     - **Callback Queue** → setTimeout, setInterval, DOM events.
   - Event Loop always empties **Microtask Queue first** before moving to Callback Queue.

---

7. **Rendering / Repainting (UI Thread)**
    - After each event loop cycle, browser checks if **UI needs update**.
    - E.g., DOM changes, CSS changes, animations.
    - **Optimization tip** → too many DOM changes per cycle = *layout thrashing* = poor performance.

---

## 📌 Representation (Simplified Flow)

```
vbnet

HTML Parsing → DOM Tree
↓
<script> encountered → Send to JS Engine
↓
Engine parses & executes → Call Stack
↓
If async → Browser Web API handles it
↓
Callback / Promise → Event Loop decides when to push
↓
Render UI updates
```

---

## 💡 Out-of-the-Box Insights

- **Why JS is Single-Threaded?**
  To avoid complexity of shared memory (like in Java/C++ threads). Instead, JS uses async with event loop.
- **Why Microtask Queue exists?**
  To give promises priority → avoids “callback hell” and race conditions.
- **Optimization Tip:**
    - Use **requestAnimationFrame** for smooth animations (sync with screen refresh rate).
    - Avoid blocking the call stack with heavy loops (use Web Workers for CPU-heavy tasks).

---

## 🔥 Interview Tricky Questions

1. Why does `setTimeout(fn, 0)` not run immediately?
   (Because it goes to Callback Queue → Event Loop waits until stack empty).
2. Between `setTimeout` and `Promise`, which runs first?
   (Promise → Microtask Queue has higher priority).
3. Is JavaScript multi-threaded?
   - Core JS → **No** (single-threaded).
   - Browser provides async APIs + event loop simulation.

---

✅ Now you have a **clear big picture** of how JS runs in a browser.

Do you want me to **zoom in on Event Loop** next (with step-by-step simulation of code execution), or on **JS Engine internals (Ignition, TurboFan, hidden classes, memory management)**?

Great choice 🔥 — the **Event Loop** is one of the most misunderstood but most important parts of JavaScript.
Let's go step by step, **one event loop cycle** at a time.

---

## 🌀 What is the Event Loop?

- JavaScript itself is **single-threaded** → it has only **one call stack**.
- But browsers (and Node.js) add **concurrency** through the Event Loop.
- The Event Loop decides *when* pending work (like timers, promises, fetch responses, DOM events) should move from **queues** into the **call stack**.

---

## ⚙️ The Cycle of Event Loop

Think of one **tick (cycle)** of the Event Loop as:

1. **Check Call Stack**
   - If the stack is not empty → keep executing code (synchronous code always runs first).
   - If the stack is empty → continue to next step.

2. **Process Microtask Queue (High Priority)**
   - Microtasks = Promises (`.then`, `catch`, `finally`), `queueMicrotask`, `MutationObserver`.
   - The event loop will **empty the entire microtask queue** before moving on.
   - ⚠ If microtasks keep adding new microtasks → infinite loop risk → browser may crash.

---

3. **Process Callback (Macrotask) Queue**
   - Macrotasks (or just tasks) = `setTimeout`, `setInterval`, `setImmediate` (Node.js), UI rendering callbacks, I/O events.
   - Only **one task** is taken from the macrotask queue per cycle → then the loop restarts.

---

4. **Render / Repaint (if needed)**
   - After microtasks & one macrotask are done, the browser checks:
     - "Does UI need updating?" (e.g., DOM changes, CSS animations).
   - If yes → browser repaints the screen.

---

5. **Repeat → Next Event Loop Cycle**
   - Then it starts again from step 1.

---

## 📊 Visual Timeline (Simplified)

```sql
Cycle Start
  ↓
[Call Stack Empty?] → No → Keep running sync code
  ↓
Yes → Run ALL Microtasks (Promises, MutationObserver)
  ↓
Take ONE Callback (Macrotask) → e.g., setTimeout
  ↓
```

```
Repaint / Render UI
  ↓
Next Cycle...
```

---

## 📌 Example Simulation

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

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

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

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

👉 Step by step:

1. `console.log("A")` → prints **A**.
2. `setTimeout(..., 0)` → timer finishes quickly → callback goes to **macrotask queue**.
3. `Promise.resolve().then(...)` → `.then` goes to **microtask queue**.
4. `console.log("D")` → prints **D**.
5. Call stack empty → event loop checks **microtask queue** → prints **C**.
6. Then event loop checks **macrotask queue** → prints **B**.

✅ Final output:

```
A
D
C
B
```

---

## 💡 Key Insights (Developers Miss These)

- **Microtasks always beat Macrotasks** → Promises are faster than setTimeout.

- **UI rendering happens after microtasks** → if you keep flooding microtasks, UI may freeze.
- **Timers (`setTimeout`) are not guaranteed exact** → if the stack is busy, they'll run later.
- **Infinite microtask chains can block rendering** → e.g., a `Promise.then()` that keeps adding another `.then()`.

---

## 🔥 Interview Traps

1. Why does `setTimeout(fn, 0)` not execute immediately?
   → Because it's a macrotask, runs only after all microtasks are done & current cycle finishes.
2. Can microtasks starve macrotasks?
   → Yes! If you keep scheduling new promises in `.then`, the event loop may **never reach macrotasks**.
3. What's the difference between **microtask queue** and **macrotask queue**?
   → Priority: microtasks run first & all of them are emptied before a macrotask executes.

---

✅ That's the **Event Loop cycle in detail**.

Would you like me to next **deep dive into microtask starvation (with an example that blocks setTimeout forever)**, or should I **move on to JS Engine internals (Ignition, TurboFan, hidden classes, garbage collection)**?