Frontend interview questions

> Provide a list of 15 questions most frequently asked in frontend interviews of javascript engine , javascript run time and their and for interview preparation

Here's a **solid list of 15 most frequently asked frontend interview questions** about the **JavaScript Engine**, **JavaScript Runtime**, and their **inner workings**, along with short, clear explanations for each — perfect for interview prep 👇

🔹 JavaScript Engine Questions

1. What is a JavaScript Engine?

A **JavaScript Engine** is a program that executes JavaScript code.
Examples: **V8 (Chrome, Node.js), SpiderMonkey (Firefox), JavaScriptCore (Safari)**.
It parses, compiles, and executes JS code — converting it to **machine code** that your CPU understands.

---

2. How does the JavaScript Engine work internally?

Basic steps:

1. **Parsing** → Converts JS code into an **AST (Abstract Syntax Tree)**.
2. **Compilation** → Uses **JIT (Just-In-Time Compiler)** to convert AST into **bytecode** or **machine code**.
3. **Execution** → Runs optimized machine code on the CPU.

---

3. What is the difference between Interpreter and Compiler in JavaScript?

- **Interpreter (Ignition in V8)** → Executes code line by line, producing bytecode.
- **Compiler (TurboFan in V8)** → Converts hot code paths into optimized machine code for better performance.
  Together they provide **fast startup + optimized execution**.

---

4. What is Bytecode in JavaScript Engines?

Bytecode is a **low-level representation** of JS code generated by the interpreter (Ignition).
It's more compact than source code and faster for the engine to execute.

---

5. What is JIT Compilation?

JIT (Just-In-Time) compilation happens during runtime.
The engine monitors frequently executed code and compiles it into **optimized machine code** for speed.

6. What are Ignition and TurboFan in V8?

- **Ignition** → V8's interpreter that produces and runs bytecode.
- **TurboFan** → V8's optimizing compiler that converts frequently run bytecode into efficient machine code.

---

7. What is the Call Stack?

The **Call Stack** keeps track of function calls.
When a function runs, it's pushed on the stack. When it returns, it's popped off.
If the stack grows too large → **Stack Overflow error**.

🔹 JavaScript Runtime Questions

8. What is the JavaScript Runtime Environment?

A **Runtime Environment** provides everything needed for JS code to run —
It includes:

- **JS Engine (V8)**
- **Web APIs (DOM, fetch, setTimeout, etc.)**
- **Callback Queue**
- **Event Loop**

Example: **Browser Runtime** or **Node.js Runtime**.

---

9. What's the difference between JS Engine and JS Runtime?

Feature	JavaScript Engine	JavaScript Runtime
Role	Executes JS code	Provides environment + APIs
Example	V8, SpiderMonkey	Chrome, Node.js
Includes	Parser, Interpreter, Compiler	Engine + Web APIs + Event Loop

---

10. What are Web APIs?

Web APIs are **browser-provided features** (not part of JS itself), such as:

- `setTimeout()`
- `fetch()`
- `DOM manipulation`
  They run **outside the JS engine** and return results via **callback queue**.

---

11. What is the Event Loop in JavaScript?

The **Event Loop** continuously checks if the **Call Stack** is empty.
If empty, it pushes pending callbacks (from Web APIs, Promises, etc.) from the **Queue** into the **Call Stack** to execute.

---

12. How do Microtasks and Macrotasks differ?

Type	Examples	Priority
Microtasks	Promises, `process.nextTick()`	Runs **before** next render
Macrotasks	`setTimeout`, `setInterval`, `setImmediate`	Runs **after** microtasks

Order of execution:
1️⃣ JS code
2️⃣ Microtasks
3️⃣ Macrotasks

---

13. What is the difference between Browser and Node.js runtime?

Feature	Browser	Node.js
APIs	DOM, fetch	File system, OS, HTTP
Engine	V8	V8
Event loop	Implemented via libuv	Implemented via libuv
Usage	Client-side	Server-side

---

14. What is a Memory Heap?

The **Heap** is an area in memory where objects, arrays, and functions are stored dynamically.
The **Garbage Collector (GC)** frees unused memory from the heap automatically.

---

15. What is a Memory Leak in JS?

A **Memory Leak** occurs when unused objects remain in memory because something still references them.
Common causes:

- Global variables not cleared
- Event listeners not removed
- Forgotten timers (`setInterval` not cleared)

🔹 Bonus Tip

If asked to **draw or explain the runtime architecture**, show this flow:

```
scss

Source Code
    ↓
JavaScript Engine (V8)
    ↓
[Parser → AST → Bytecode → Machine Code]
    ↓
Runtime Environment (Browser/Node)
    ↓
[Call Stack + Heap + Web APIs + Event Loop + Queues]
```

Would you like me to make a **visual diagram (runtime + engine flow)** for quick revision and better interview explanation? It helps a lot in face-to-face interviews.

> provide 5 more of javascript engine related good question and 5 more good of run time environment questions

Perfect 👌 Vipul — here are **10 more advanced-yet-frequently-asked frontend interview questions**, divided neatly into two sets:
🧠 **JavaScript Engine (5)** and ⚙️ **JavaScript Runtime Environment (5)** — with short, clear explanations ideal for interviews.

🧠 JavaScript Engine – Advanced Interview Questions

1. What is an Abstract Syntax Tree (AST)?

- After parsing JS code, the engine creates an **AST** — a **tree structure** representing code syntax.
- It helps the engine understand *what* your code does and is used in **optimizations**, **compilation**, and **linting tools** like ESLint.
- Example:

  ```
  js

  const a = 5 + 2;
  ```

  becomes a tree:
  → **`VariableDeclaration → Identifier(a) → BinaryExpression(5 + 2)`**

---

2. How does the Garbage Collector work in JavaScript Engines like V8?

- Garbage Collection removes **unreachable objects** from memory.
- V8 uses algorithms like:
  - **Mark-and-Sweep** → Marks reachable objects; unmarked ones are deleted.
  - **Generational GC** → Separates memory into *young* and *old* generations for faster cleanup.

- Purpose: Prevent memory leaks and optimize performance.

---

3. What are hidden classes and inline caching in V8?

- **Hidden Classes:**
  V8 internally assigns "hidden classes" to objects for faster property access — like a C++ class layout.
- **Inline Caching:**
  When a property is accessed repeatedly, V8 caches its memory location → boosts performance for repeated lookups.
  Example:

```
js

user.name  // cached after first lookup
```

---

4. What is Deoptimization in V8?

- Sometimes, V8 optimizes code based on assumptions (like variable types).
  If assumptions break, it **deoptimizes** and falls back to the interpreter.
  Example:

```
js

function add(a, b) { return a + b; }
add(5, 10);     // optimized for numbers
add("5", "10"); // deoptimized when strings appear
```

---

5. What are Inline Functions and Function Inlining in JS Engines?

- **Inlining** = replacing a function call with its actual body to avoid overhead.
- Example:

```
js

function square(x) { return x * x; }
square(2);  // Engine may inline → 2 * 2 directly.
```

- This makes execution faster for short, frequently called functions.

⚙️ JavaScript Runtime Environment – Advanced Interview Questions

1. What is the role of the Callback Queue and Microtask Queue?

- **Callback Queue (Macrotask Queue)** → stores tasks like **`setTimeout`**, **`setInterval`**.
- **Microtask Queue** → stores tasks from **`Promises`**, **`MutationObserver`**.

- **Event Loop** always processes *microtasks first*, then macrotasks.

---

2. How does the Event Loop behave with async/await?

- `async/await` uses **Promises** under the hood.
- When `await` is hit:
  1. The current function pauses.
  2. The rest of the code continues executing.
  3. Once the promise resolves, its continuation is pushed to the **microtask queue**.

---

3. What happens when you use `setTimeout(fn, 0)`?

- Even with `0` delay, the callback doesn't run immediately.
- It's placed in the **macrotask queue** and runs **after** the current call stack and **all microtasks** are done.

```
js

console.log(1);
setTimeout(() => console.log(2), 0);
Promise.resolve().then(() => console.log(3));
console.log(4);
// Output: 1, 4, 3, 2
```

---

4. How does Node.js implement the Event Loop differently from Browsers?

- Node.js uses **libuv** library to manage its event loop.
- Phases:
  1. **Timers** (`setTimeout`)
  2. **I/O callbacks**
  3. **Idle/prepare**
  4. **Poll**
  5. **Check** (`setImmediate`)
  6. **Close callbacks**
- Browser loop focuses on **rendering + user events**, while Node focuses on **I/O + async file/network tasks**.

---

5. What is the role of the Execution Context in a JS runtime?

- Every time code runs, an **Execution Context** is created.
  Types:
  1. **Global Context** – for top-level code.

2. **Function Context** – for each function call.
3. **Eval Context** – when `eval()` is used.

- Each context has:
  - **Variable Environment**
  - **Lexical Environment**
  - **this binding**
- All these contexts form the **Call Stack** during execution.

✅ Summary Table

Category	Concept	Key Idea
Engine	AST	Tree representation of code
Engine	GC	Frees unreachable objects
Engine	Hidden Classes	Faster property access
Engine	Deoptimization	Reverts optimizations if assumptions fail
Engine	Function Inlining	Replace call with code
Runtime	Callback Queue	Macrotasks like **`setTimeout`**
Runtime	Microtask Queue	Promises, async/await
Runtime	Event Loop	Handles task scheduling
Runtime	Node.js Loop	libuv-based phases
Runtime	Execution Context	Scope + environment for execution

Would you like me to create a **visual diagram** showing how
“Engine (V8) + Runtime (Browser/Node) + Event Loop + Queues + Heap + Stack” interact together?
It’s an excellent visual for interviews — you can explain everything in <2 minutes using it.