

Event loop questions



provide a list of questions of 15 Output based questions 5 easy 5 meedium, 5 hard of event loop, task queue, micro task queue concepts questions.

Perfect, Vipul 🌟 — here's a **comprehensive, 15-question set** (with difficulty separation) focused on **Event Loop, Task Queue (macrotask queue)**, and **Microtask Queue** concepts.

Each question gives a **short code snippet** and asks you to **predict the output** — this is exactly the kind of problem interviewers use to test real understanding of async execution order in JavaScript.

EASY LEVEL (5 Questions)

Q1

```
js

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

Concept: setTimeout is a macrotask, logs after current call stack.

Q2

```
js

console.log("Start");
Promise.resolve().then(() => console.log("Promise resolved"));
console.log("End");
```

Concept: Microtasks (Promise.then) run before macrotasks.

Q3

```
js

setTimeout(() => console.log("setTimeout"), 0);
Promise.resolve().then(() => console.log("Promise"));
console.log("Script end");
```

Concept: Microtasks (Promise) execute before next macrotask.

Q4

```
js

console.log("1");
setTimeout(() => console.log("2"), 1000);
setTimeout(() => console.log("3"), 0);
console.log("4");
```

 *Concept: Multiple macrotasks with different delays.*

Q5

```
js

Promise.resolve().then(() => console.log("Microtask 1"));
Promise.resolve().then(() => console.log("Microtask 2"));
console.log("Synchronous");
```

 *Concept: Multiple microtasks are queued and processed in order.*

MEDIUM LEVEL (5 Questions)

Q6

```
js

console.log("Start");

setTimeout(() => {
  console.log("Timeout 1");
  Promise.resolve().then(() => console.log("Promise inside Timeout"));
}, 0);

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

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

 *Concept: Microtask inside a macrotask.*

Q7

```
js

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

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

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

🧠 Concept: Promises create microtasks; nested macrotasks run later.

Q8

```
js

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

🧠 Concept: Microtask queue inside another microtask.

Q9

```
js

setTimeout(() => console.log("1"), 0);
queueMicrotask(() => {
  console.log("2");
  queueMicrotask(() => console.log("3"));
});
console.log("4");
```

🧠 Concept: queueMicrotask always executes before macrotasks.

Q10

```
js

Promise.resolve().then(() => console.log("Microtask 1"));
setTimeout(() => console.log("Macrotask 1"), 0);
Promise.resolve().then(() => {
  console.log("Microtask 2");
  setTimeout(() => console.log("Macrotask 2"), 0);
});
console.log("Main");
```

🧠 Concept: Execution order across nested micro/macrotasks.

HARD LEVEL (5 Questions)

Q11

```
js

console.log("Start");

setTimeout(() => {
  console.log("Timeout 1");
  Promise.resolve().then(() => console.log("Promise 1"));
}, 0);

Promise.resolve().then(() => {
  console.log("Promise 2");
  setTimeout(() => console.log("Timeout 2"), 0);
});

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

🧠 Concept: Mixed nesting of micro and macrotasks.

Q12

```
js

Promise.resolve()
  .then(() => {
    console.log("P1");
    setTimeout(() => console.log("T1"), 0);
  })
  .then(() => {
    console.log("P2");
    setTimeout(() => console.log("T2"), 0);
  });

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

🧠 Concept: Chained `.then()` creates multiple microtasks sequentially.

Q13

```
js

setTimeout(() => console.log("T1"), 0);

Promise.resolve().then(() => {
  console.log("P1");
  queueMicrotask(() => console.log("Q1"));
  setTimeout(() => console.log("T2"), 0);
});

queueMicrotask(() => console.log("Q2"));

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

🧠 Concept: Interaction between microtask APIs and `setTimeout`.

Q14

```
js

setTimeout(() => console.log("Macrotask 1"), 0);

Promise.resolve()
  .then(() => {
    console.log("Microtask 1");
    return Promise.resolve();
})
  .then(() => {
    console.log("Microtask 2");
    setTimeout(() => console.log("Macrotask 2"), 0);
  });

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

 Concept: Promise chaining with macrotask scheduling.

Q15

```
js

console.log("Start");

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

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

queueMicrotask(() => {
  console.log("Microtask 1");
  queueMicrotask(() => console.log("Microtask 2"));
});

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

 Concept: Deep nesting — microtask inside microtask and Promise inside Promise.

Would you like me to provide **answers + detailed explanation (execution order)** for all 15 questions next?

That will help you **master the event loop once and for all** with real reasoning.



provide thier oupput's also with explanation

Perfect, Vipul 🙌 — this is one of the **most important concepts** for frontend interviews.

Let's go step-by-step through **all 15 Event Loop questions**, with **outputs** and **detailed explanations** that tell you *why* things happen in that order.

EASY LEVEL

Q1

```
js

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

Output:

css

A
C
B

Explanation:

1. "A" → synchronous → logs immediately.
2. `setTimeout` → schedules a macrotask → goes to task queue.
3. "C" → synchronous → logs immediately.
4. Event loop finishes main thread → executes macrotasks → logs "B".

Q2

```
js

console.log("Start");
Promise.resolve().then(() => console.log("Promise resolved"));
console.log("End");
```

Output:

sql

Start
End
Promise resolved

Explanation:

- Promises' `.then()` callbacks are **microtasks**, executed after current call stack but before macrotasks.
- So "Promise resolved" runs **after** "End" but **before** any `setTimeout`.

Q3

js

```
setTimeout(() => console.log("setTimeout"), 0);
Promise.resolve().then(() => console.log("Promise"));
console.log("Script end");
```

 **Output:**

arduino

```
Script end
Promise
setTimeout
```

 **Explanation:**

1. `setTimeout` → macrotask.
2. `Promise.then` → microtask.
3. Microtasks always execute **before** the next macrotask.

Q4

js

```
console.log("1");
setTimeout(() => console.log("2"), 1000);
setTimeout(() => console.log("3"), 0);
console.log("4");
```

 **Output:**

```
1
4
3
2
```

 **Explanation:**

- Both timeouts are macrotasks, but **3** (0 ms) executes before **2** (1000 ms).
- Synchronous logs first, then tasks execute based on delay.

Q5

js

```
Promise.resolve().then(() => console.log("Microtask 1"));
Promise.resolve().then(() => console.log("Microtask 2"));
console.log("Synchronous");
```

 **Output:**

```
nginx
Synchronous
Microtask 1
Microtask 2
```

 **Explanation:**

- All synchronous code first.
- Microtasks (Promise callbacks) are queued and executed **in the order they were added**.

 **MEDIUM LEVEL****Q6**

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

setTimeout(() => {
  console.log("Timeout 1");
  Promise.resolve().then(() => console.log("Promise inside Timeout"));
}, 0);

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

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

 **Output:**

```
javascript
Start
End
Promise 1
Timeout 1
Promise inside Timeout
```

 **Explanation:**

1. Synchronous → "Start", "End".
2. "Promise 1" → microtask → before any timeout.
3. "Timeout 1" runs after current stack.
4. "Promise inside Timeout" → new microtask inside the timeout → runs **immediately after "Timeout 1"** within that same event loop tick.

Q7

```
js

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

Promise.resolve().then(() => {
  console.log("Promise 1");
  setTimeout(() => console.log("setTimeout 2"), 0);
});

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

Output:

```
javascript

End
Promise 1
setTimeout 1
setTimeout 2
```

Explanation:

- "Promise 1" runs before `setTimeout` tasks.
- `setTimeout 2` was created inside a microtask → added later → runs **after** `setTimeout 1`.

Q8

```
js

console.log("A");

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

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

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

Output:

```
mathematica

A
E
C
D
B
```

Explanation:

1. "A" and "E" are synchronous.
2. "C" is first microtask.
3. Inside that microtask, another promise schedules "D".
4. All microtasks (C, D) complete **before** moving to macrotask queue (B).

Q9

```
js

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

queueMicrotask(() => {
  console.log("2");
  queueMicrotask(() => console.log("3"));
});

console.log("4");
```

 **Output:**

```
4
2
3
1
```

 **Explanation:**

- `queueMicrotask` behaves like `Promise.then`.
- Nested microtasks execute sequentially before any macrotask.

Q10

```
js

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

setTimeout(() => console.log("Macrotask 1"), 0);

Promise.resolve().then(() => {
  console.log("Microtask 2");
  setTimeout(() => console.log("Macrotask 2"), 0);
});

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

 **Output:**

```
css

Main
Microtask 1
Microtask 2
Macrotask 1
Macrotask 2
```

 **Explanation:**

- All microtasks (**Microtask 1**, **Microtask 2**) first.
- Then macrotasks (**Macrotask 1**, then **Macrotask 2**) in order they were queued.

HARD LEVEL

Q11

```
js

console.log("Start");

setTimeout(() => {
  console.log("Timeout 1");
  Promise.resolve().then(() => console.log("Promise 1"));
}, 0);

Promise.resolve().then(() => {
  console.log("Promise 2");
  setTimeout(() => console.log("Timeout 2"), 0);
});

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

Output:

```
javascript

Start
End
Promise 2
Timeout 1
Promise 1
Timeout 2
```

Explanation:

1. "Promise 2" runs before any timeout.
2. "Timeout 1" executes → inside it, another microtask "Promise 1" runs immediately.
3. "Timeout 2" executes last because it was scheduled inside the first microtask.

Q12

```
js

Promise.resolve()
  .then(() => {
    console.log("P1");
    setTimeout(() => console.log("T1"), 0);
  })
  .then(() => {
    console.log("P2");
    setTimeout(() => console.log("T2"), 0);
  });

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

Output:

```
nginx
```

```
S
P1
P2
T1
T2
```

Explanation:

- Promise chaining ensures `.then()` executes sequentially as microtasks.
- Timeouts are queued after each `.then()` — executed in order.

Q13

```
js

setTimeout(() => console.log("T1"), 0);

Promise.resolve().then(() => {
  console.log("P1");
  queueMicrotask(() => console.log("Q1"));
  setTimeout(() => console.log("T2"), 0);
});

queueMicrotask(() => console.log("Q2"));

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

Output:

```
nginx
```

```
S
P1
Q2
Q1
T1
T2
```

Explanation:

1. "S" → sync.
2. "P1" → microtask.
3. "Q2" was queued before "P1" ran, but "Q1" gets queued during "P1", so Q2 → Q1.
4. Then macrotasks: T1 → T2.

Q14

```
js

setTimeout(() => console.log("Macrotask 1"), 0);

Promise.resolve()
  .then(() => {
```

```

    console.log("Microtask 1");
    return Promise.resolve();
})
.then(() => {
    console.log("Microtask 2");
    setTimeout(() => console.log("Macrotask 2"), 0);
});

console.log("Synchronous");

```

Output:

nginx

Synchronous
 Microtask 1
 Microtask 2
 Macrotask 1
 Macrotask 2

Explanation:

- Promises (**Microtask 1**, **Microtask 2**) before all timeouts.
- Macrotasks execute later in the order they were queued.

Q15

```

js

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

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

queueMicrotask(() => {
    console.log("Microtask 1");
    queueMicrotask(() => console.log("Microtask 2"));
});

console.log("End");

```

Output:

javascript

Start
 End
 Promise 1
 Microtask 1
 Promise 2
 Microtask 2
 setTimeout

Explanation:

- Both Promise and `queueMicrotask` callbacks are microtasks, but **Promise jobs go first** as they're queued in registration order.
- "Promise 2" and "Microtask 2" are nested microtasks and run in insertion order.
- Finally, `setTimeout` executes.

Summary Table

Type	Executes When	Examples
Synchronous	Immediately	<code>console.log</code> , loops
Microtasks	After current call stack, before macrotasks	<code>Promise.then</code> , <code>queueMicrotask</code> , <code>MutationObserver</code>
Macrotasks	After microtasks complete	<code>setTimeout</code> , <code>setInterval</code> , I/O, DOM events

Would you like me to create a **visual diagram (event loop flow)** showing how call stack, microtask queue, and macrotask queue interact (like a step-by-step animation explanation for one hard example)?