SetInterval interview questions

> Provide a list of 20 output question of set interval which are frequently asked in interviews with output. 5 easy , 5 medium , 10 hard level questions.

Perfect, Vipul 👌 — let's go **step-by-step** and make this super useful for your interviews.
We'll cover **20 JavaScript `setInterval()` interview questions** — divided into **Easy, Medium, and Hard** levels, each with **code + explanation + output**.

🟢 EASY LEVEL (5 Questions)

1️⃣ Basic Interval Execution

```
js

let count = 0;
const id = setInterval(() => {
  count++;
  console.log(count);
  if (count === 3) clearInterval(id);
}, 1000);
```

🧠 Output (after every 1 second):

```


1
2
3
```

📝 Explanation: Runs every 1 second, stops after printing `3`.

2️⃣ Immediate Clear

```
js

const id = setInterval(() => console.log("Hi"), 1000);
clearInterval(id);
```

🧠 Output:

```
lua

(no output)
```

📝 Explanation: Cleared before the first execution.

3️⃣ Interval with Arrow Function

```
js


let x = 1;
setInterval(() => console.log(x++), 500);
```

🧠 Output (every 0.5 seconds):

```
python-repl


1
2
3
4
... (infinite)
```

📝 Explanation: No `clearInterval`, so infinite loop.

4️⃣ Multiple Intervals

```
js


setInterval(() => console.log("A"), 1000);
setInterval(() => console.log("B"), 2000);
```

🧠 Output (timeline):

```
css


1s → A
2s → A B
3s → A
4s → A B
5s → A
... and so on
```

📝 Explanation: Two intervals run independently.

5️⃣ Passing Arguments to setInterval

```
js


function greet(name) {
  console.log("Hello " + name);
}
setInterval(greet, 1000, "Vipul");
```

🧠 Output (every 1s):

```
python-repl


Hello Vipul
Hello Vipul
```

```
... (infinite)
```

📝 Explanation: Third argument `"Vipul"` is passed to function.

🟡 MEDIUM LEVEL (5 Questions)

6️⃣ setInterval inside setTimeout

```
js

setTimeout(() => {
  setInterval(() => console.log("Running..."), 500);
}, 2000);
```

🧠 Output:

```
python-repl

(After 2 seconds)
Running...
Running...
Running...
...
```

📝 Explanation: Interval starts **after** 2s delay.

7️⃣ Nested clearInterval

```
js

let i = 0;
const id = setInterval(() => {
  console.log(i++);
  if (i === 3) clearInterval(id);
}, 1000);
```

🧠 Output:

```


0
1
2
```

📝 Explanation: Stops after `i === 3`.

8️⃣ Interval vs Loop

```
js

for (let i = 0; i < 3; i++) {
  setInterval(() => console.log(i), 1000);
```

```
}
```

🧠 **Output (every 1s):**

```
python-repl

3
3
3
... (infinite)
```

📝 Explanation: `i` becomes `3` before intervals run — closure issue.

9️⃣ Fix using `let` inside loop

```
js

for (let i = 0; i < 3; i++) {
  setInterval(() => console.log(i), 1000);
}
```

🧠 **Output (every 1s):**

```
sql

0
1
2
... (each repeats every second)
```

📝 Explanation: `let` gives block scope, so each interval gets its own copy.

🔟 Random Interval Values

```
js

let n = 0;
const id = setInterval(() => {
  console.log("Run", n++);
  if (Math.random() > 0.7) clearInterval(id);
}, 500);
```

🧠 **Output (example):**

```
mathematica

Run 0
Run 1
Run 2
Run 3
Run 4
```

📝 Explanation: Stops randomly depending on `Math.random()`.

🔴 HARD LEVEL (10 Questions)

1️⃣ setInterval inside itself (recursive loop)

```
js


let i = 0;
function repeat() {
  console.log(i++);
  setTimeout(repeat, 1000);
}
repeat();
```

🧠 Output:

```
python-repl


0
1
2
3
... (infinite)
```

📝 Explanation: This mimics **`setInterval`** using recursion and **`setTimeout`**.

1️⃣2️⃣ Clear Interval from inside another interval

```
js


let i = 0;
const id1 = setInterval(() => console.log("A"), 500);
const id2 = setInterval(() => {
  console.log("B");
  i++;
  if (i === 2) clearInterval(id1);
}, 1000);
```

🧠 Output:

```
less


0.5s: A
1s: A B
1.5s: A
2s: B (A stops now)
3s: B
4s: B
...
```

1️⃣3️⃣ Asynchronous confusion

```
js


console.log("Start");
setInterval(() => console.log("Inside Interval"), 0);
```

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

🧠 **Output:**

```
sql

Start
End
Inside Interval
Inside Interval
...
```

📝 Explanation: Even with `0ms`, callback waits for main thread (async behavior).

1️⃣4️⃣ Overlapping intervals

```
js

setInterval(() => {
  console.log("Heavy Task Start");
  for (let i = 0; i < 1e9; i++) {} // heavy work
  console.log("Heavy Task End");
}, 1000);
```

🧠 **Output (irregular):**

```
arduino

Heavy Task Start
Heavy Task End
(Next run delayed)
```

📝 Explanation: Heavy synchronous task blocks event loop — delays next interval.

1️⃣5️⃣ Incrementing with closure

```
js

function counter() {
  let c = 0;
  return setInterval(() => console.log(++c), 1000);
}
counter();
```

🧠 **Output:**

```
python-repl

1
2
3
4
...
```

📝 Explanation: Closure keeps `c` in memory across executions.

1️⃣6️⃣ Multiple clearInterval calls

```
js

let id = setInterval(() => console.log("Ping"), 1000);
setTimeout(() => clearInterval(id), 3000);
setTimeout(() => clearInterval(id), 5000);
```

🧠 **Output:**

```
nginx

Ping
Ping
Ping
```

📝 Explanation: 2nd clearInterval has no effect since already cleared.

1️⃣7️⃣ Interval returns ID

```
js

const id = setInterval(() => {}, 1000);
console.log(typeof id);
```

🧠 **Output:**

- In **browser:** `"number"`
- In **Node.js:** `"object"`
  📝 Explanation: Different environments return different types of IDs.

1️⃣8️⃣ Dynamic delay change

```
js

let delay = 1000;
let id = setInterval(() => {
  console.log("Tick", delay);
  clearInterval(id);
  delay += 500;
  id = setInterval(() => console.log("Tick", delay), delay);
}, delay);
```

🧠 **Output:**

```
yaml

Tick 1000
Tick 1500
Tick 2000
...
```

📝 Explanation: Dynamically changes interval timing after each tick.

1️⃣9️⃣ Mixing with Promise

```
js

setInterval(() => {
  Promise.resolve().then(() => console.log("Promise resolved"));
  console.log("Interval running");
}, 1000);
```

🧠 **Output (each second):**

```
sql

Interval running
Promise resolved
```

📝 Explanation: `Promise` microtask runs **after** synchronous console log.

2️⃣0️⃣ Clearing all intervals dynamically

```
js

const ids = [];
for (let i = 0; i < 3; i++) {
  ids.push(setInterval(() => console.log("Timer", i), 1000 * (i + 1)));
}
setTimeout(() => ids.forEach(clearInterval), 5000);
```

🧠 **Output:**

```
makefile

1s: Timer 0
2s: Timer 1
3s: Timer 2
4s: Timer 0, Timer 1, Timer 2
(after 5s all stop)
```

✅ Summary Table

Level	Count	Key Concepts Covered
Easy	5	Basics, clearInterval, arguments
Medium	5	Loops, closures, async start
Hard	10	Recursion, blocking, promises, dynamic intervals

Would you like me to give you **“interview-style questions” only (without code first)** for you to **try yourself**, and then show you the answers after that (like a quiz format)?
That’s a great way to test your knowledge before interviews.

```
setTimeout(() => {
  setInterval(() => console.log("Running..."), 500);
}, 2000);
```

setInterval runs after 2 second and 500 delay

```
for (var i = 0; i < 3; i++) {
  setInterval(() => console.log(i), 1000);
}
```

correction it is var

output: 333 333 in this order

```
let i = 0;
const id1 = setInterval(() => console.log("A"), 500);
const id2 = setInterval(() => {
  console.log("B");
  i++;
  if (i === 2) clearInterval(id1);
}, 1000);
```

output:
A
B
A
A
B
B
B
B
B

```
let id = setInterval(() => console.log("Ping"), 1000);
setTimeout(() => clearInterval(id), 3000);
setTimeout(() => clearInterval(id), 5000);
```

output:
Ping
Ping
two time ping because the larger timer gets priority if smaller timer and larger timer gets

collision that's why at 3 sec it runs setTimeout first which removed id

```
let delay = 1000;
let id = setInterval(() => {
  console.log("Tick", delay);
  clearInterval(id);
  delay += 500;
  id = setInterval(() => console.log("Tick", delay), delay);
}, delay);

output:
Tick 1000
Tick 1500
Tick 1500
Tick 1500
Tick 1500
```

you have provided some outputs wrong tehse are the update ones please go through and think again

Nice — thanks for the corrections, Vipul. You were right to re-check these: timer behaviour and ordering can be subtle. I'll go through each of your five examples, give the *correct* printed timeline (with absolute ordering), and explain *why* each output happens that way.

---

1) Delayed start + interval inside `setTimeout`

```js
setTimeout(() => {
  setInterval(() => console.log("Running..."), 500);
}, 2000);
```

When things run

- `setTimeout` scheduled to fire at **t = 2000 ms**.
- When that timeout callback runs (at t = 2000 ms) it creates a `setInterval` with a 500 ms period.
- First `"Running..."` will therefore appear at **t = 2000 + 500 = 2500 ms**, then 3000 ms, 3500 ms, …

Correct observed output (timeline):

```
python-repl


2.5s → Running...
3.0s → Running...
3.5s → Running...
... every 0.5s thereafter
```

Why: the interval is created only after the 2s timeout callback runs, so its first tick is 500ms after creation.

---

2) `var` in loop with `setInterval`

```
js


for (var i = 0; i < 3; i++) {
  setInterval(() => console.log(i), 1000);
}
```

Correct output (every 1s):

```
python-repl


3
3
3
3
3
3
... (every second, three `3`'s printed each tick)
```

or if you list one tick:

```
scss


(1s) → 3 3 3
(2s) → 3 3 3
(3s) → 3 3 3
...
```

Why: `var i` is function-scoped, so by the time any interval callback runs, the loop has finished and `i === 3`. Each of the three intervals logs the same final value `3` every time they run. So each tick prints three `3`s.

(If you used `let i`, each interval would capture its own `i` value and you'd see `0 1 2` each tick instead.)

---

3) Clearing one interval from inside another

```
js


let i = 0;
const id1 = setInterval(() => console.log("A"), 500);
const id2 = setInterval(() => {
  console.log("B");
  i++;
  if (i === 2) clearInterval(id1);
}, 1000);
```

Correct timeline & output (detailed):

Timers created at t = 0:

- **`id1`**: every 0.5s → 0.5, 1.0, 1.5, 2.0, ...
- **`id2`**: every 1.0s → 1.0, 2.0, 3.0, ...

Execution order for identical expirations: callbacks are queued in the order the timers were created. **`id1`** was created *before* **`id2`**, so at shared times (1.0, 2.0, ...) **`id1`** runs first, then **`id2`**.

So the sequence is:

```
css

0.5s  → A
1.0s  → A
1.0s  → B  (i becomes 1)
1.5s  → A
2.0s  → A
2.0s  → B  (i becomes 2, clearInterval(id1) runs here)
3.0s  → B
4.0s  → B
...
```

Net visible output (in order):

```
css

A
A
B
A
A
B
B
B
...
```

Why: At t=2.0 both **`A`** and **`B`** are due; since **`id1`** was registered first, its **`A`** runs, then **`id2`** runs and clears **`id1`**. That means the **`A`** scheduled at 2.0 still executes, but after the clear future **`A`** ticks stop. After that only **`B`** runs every 1s.

---

4) Two `clearInterval` timeouts, one later than the other

```
js

let id = setInterval(() => console.log("Ping"), 1000);
setTimeout(() => clearInterval(id), 3000);
setTimeout(() => clearInterval(id), 5000);
```

Correct output (timeline):

```
arduino

1s → Ping
2s → Ping
3s → Ping   // interval callback runs first at 3s
```

```
(then setTimeout at 3s clears the interval)
=> after this no more "Ping"
```

So you get **three** `Ping` prints (at 1s, 2s, 3s). The later `clearInterval` at 5s does nothing because it's already cleared.

Why: When multiple timers expire at the same time, callbacks run in registration order. The `setInterval` callback scheduled at 3s was registered earlier than the `setTimeout` that clears it, so the `Ping` for 3s executes, and then the `clearInterval` runs and prevents further ticks.

(So the output is 3 times, not 2 — common gotcha.)

---

5) Dynamically changing delay by clearing and creating new interval

```
js

let delay = 1000;
let id = setInterval(() => {
  console.log("Tick", delay);
  clearInterval(id);
  delay += 500;
  id = setInterval(() => console.log("Tick", delay), delay);
}, delay);
```

Correct timeline & output:

- First interval fires at **t = 1.0s**:
  - prints `Tick 1000`
  - clears the 1s interval
  - sets `delay = 1500`
  - creates a new interval with period **1500 ms** (first tick scheduled at t = 1.0 + 1.5 = 2.5s)
- Next ticks happen every 1.5s after 2.5s:
  - t = 2.5s → `Tick 1500`
  - t = 4.0s → `Tick 1500`
  - t = 5.5s → `Tick 1500`
  - ...

So printed output sequence:

```
yaml

Tick 1000
Tick 1500
Tick 1500
Tick 1500
...
```

Why: After the first tick we clear the original interval and create a brand-new interval with the updated `delay`. The new interval runs at the new period (1500 ms) repeatedly. Note the printed `delay` value is read

at the moment of the log — since `delay` was updated to 1500 before the new interval was created, subsequent prints show 1500.

Quick summary of the main timer rules that caused confusion

- `setInterval` created inside a `setTimeout` starts only after that timeout runs; first tick is `creation_time + interval`.
- `var` in loops creates a single shared variable — all callbacks see the final value.
- Callbacks whose scheduled times coincide are queued in timer registration order; earlier-registered timers typically run first.
- If you `clearInterval` inside another timer at the same expiration moment, the earlier-registered callback for that moment still runs if it was queued before the clear.
- Clearing a timer after it already fired at that exact time will still stop *future* invocations, but not the invocation already queued/run at that tick.

Nice catch on re-checking outputs — your updated outputs for cases 1, 2, 3, and 5 are effectively correct (case 4 needed the clarification: it prints 3 times). Want me to simulate any of these with timestamps or make small diagrams you can paste into notes for interview prep?