JavaScript learning path

Great 👍 — let's now go **one step deeper** into **real-world use cases + more tricky pitfalls/interview-style problems** for the loops we already covered (`while`, `do…while`, `for`, `break`, `continue`, `labels`).

🔁 Loops – Real-World Usage + Interview Pitfalls

✅ Real-World Use Cases

1. `while` loop

- **Best for unknown iteration counts** (runs until condition met).
  Example: keep asking user input until valid number:

```
js

let value;
while (true) {
  value = prompt("Enter a number greater than 10:");
  if (value > 10) break;
}
console.log("You entered:", value);
```

---

2. `do…while` loop

- **Guaranteed execution at least once**.
  Example: menu-driven program.

```
js

let choice;
do {
  choice = prompt("1. Add  2. Exit");
  if (choice == 1) console.log("Adding...");
} while (choice != 2);
```

---

3. `for` loop

- **Best for known iteration counts.**
  Example: printing multiplication table.

```
js

for (let i = 1; i <= 10; i++) {
  console.log(`5 x ${i} = ${5 * i}`);
}
```

---

4. `break` in real use

- **Early termination when condition met.**

```
js

let arr = [3, 6, -2, 7, 9];
for (let i = 0; i < arr.length; i++) {
  if (arr[i] < 0) {
    console.log("Found negative:", arr[i]);
    break;
  }
}
```

---

5. `continue` in real use

- **Skip processing certain cases.**

```
js

for (let i = 1; i <= 10; i++) {
  if (i % 2 === 0) continue; // skip even
  console.log("Odd:", i);
}
```

---

6. Labels in real use

- **Breaking out of nested loops** (rare, but sometimes needed).

```js
outer: for (let i = 0; i < 5; i++) {
  for (let j = 0; j < 5; j++) {
    if (i * j > 6) break outer;
    console.log(i, j);
  }
}
```

⚠ Common Pitfalls + Interview Brain-Venders

❓ Q1. What's the output?

```js
let i = 0;
while (i++ < 5) {
  console.log(i);
}
```

👉 **Output:** `1 2 3 4 5`

Reason: `i++` compares first, then increments.

❓ Q2. Guess the result

```js
for (let i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 100);
}
```

👉 **Output:** `0 1 2`

Because `let` makes `i` block-scoped (new variable each iteration).

⚡ Tricky twist: if it was `var i`, output = `3 3 3`.

❓ Q3. Can this loop end?

```js
for (let i = 0; i != 10; i += 0.2) {}
```

👉 **No guarantee!** Floating-point numbers may skip 10 (e.g., 9.8 → 10.0 may never be exact).

❓ Q4. Nested break confusion

```js
for (let i = 0; i < 2; i++) {
  for (let j = 0; j < 2; j++) {
    if (j === 1) break;
    console.log(i, j);
  }
}
```

👉 **Output:**

```
0 0
1 0
```

Because `break` only exits inner loop.

❓ Q5. Infinite loop subtlety

```js
let x = 0.1;
while (x !== 1.0) {
  x += 0.1;
}
```

👉 This is **infinite**.
Because floating-point addition may never produce exactly `1.0`.

❓ Q6. Scope trap

```js
for (var i = 0; i < 3; i++) {}
console.log(i); // ?
```

👉 Prints `3` (because `var` is function-scoped).
With `let`, would be ReferenceError.

🧠 Interviewer Tricks

- "Why is `do…while` rarely used in production?"
  👉 Because we usually want condition checked before execution. It's useful in rare cases like menu-driven apps.
- "How to safely terminate a loop that depends on floating-point comparison?"
  👉 Use tolerance:

```js
while (Math.abs(x - 1.0) > 1e-9) { ... }
```

- "Why is `for(;;)` dangerous?"
  👉 It's an **infinite loop** unless controlled with `break`. Used in event loops, but risky for beginners.

✅ That's the **extended deep dive with real-world cases + pitfalls + brain-venders**.

Do you want me to **next cover "loop optimizations (performance tricks, micro-optimizations, interview hacks)"** or should we go ahead to **special loops (`for…in`, `for…of`)**?