JavaScript Performance & Event Loop Notes

---

1. Efficient & Performance-Oriented Code

Writing efficient code is about **minimizing unnecessary work in the browser**, especially **reflows and repaints**.

Key Points

- Avoid unnecessary DOM manipulations inside loops.
- Batch DOM changes instead of updating it multiple times.
- Use `documentFragment` for multiple element additions.
- Minimize CSS changes that trigger layout recalculation.

Performance Measurement

You can measure execution time of a code block using `performance.now()`:

```
const t1 = performance.now();

for(let i=0; i<1000; i++) {
    Math.sqrt(i);
}

const t2 = performance.now();
console.log("Time taken: ", t2 - t1, "ms");
```

---

2. Reflow vs Repaint

Reflow (Layout)

- Happens when **browser recalculates element positions and sizes**.

- Triggered by: changing element size, position, content, font, etc.

Repaint (Style)

- Happens when **only the visual appearance changes**, without layout changes.
- Triggered by: color, visibility, background changes.

⚡ **Good Practice:** Minimize reflows → group DOM updates, use `classList` instead of inline styles repeatedly.

---

3. Call Stack & Single-Threading

JavaScript is single-threaded

- Only **one piece of code runs at a time**.
- JS executes in a **run-to-completion** manner.

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

function foo() {
  console.log("Inside foo");
}

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

Output:

```
Start
Inside foo
End
```

- JS will **complete the current function** before moving to the next.

- There’s **no parallel execution** in the main thread (unless using Web Workers).

4. Event Loop & Asynchronous Code

JavaScript uses an **event loop** to handle asynchronous operations (timers, promises, DOM events) **without blocking the main thread**.

How It Works

1. **Call Stack:** Executes synchronous code.
2. **Web APIs / Browser APIs:** Handles async functions like `setTimeout`, AJAX, DOM events.
3. **Callback Queue / Task Queue:** Stores callbacks ready to run.
4. **Event Loop:** Checks if the call stack is empty → pushes next callback from the queue.

Example: setTimeout with 0ms

```
function foo() {
    console.log("foo has been called");
}

setTimeout(foo, 0); // async
console.log("After setTimeout"); // sync
```

Output:

```
After setTimeout
foo has been called
```

Explanation:

- `console.log("After setTimeout")` executes immediately (synchronous).
- `foo` is scheduled on the **callback queue**, executed after the current call stack is empty.

Example with multiple timers

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

setTimeout(() => console.log("Timeout 1"), 0);
setTimeout(() => console.log("Timeout 2"), 10);

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

Output:

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

- Even `0ms` timeout is **not immediate**.
- JS ensures **current stack finishes first** before executing queued callbacks.

---

5. Practical Tips for Performance

1. **Batch DOM updates**

```
const fragment = document.createDocumentFragment();
for(let i=0; i<100; i++){
    const p = document.createElement("p");
    p.textContent = `Paragraph ${i}`;
    fragment.appendChild(p);
}
document.body.appendChild(fragment); // single DOM update
```

2. **Minimize layout thrashing**

```
const height = element.offsetHeight; // read
```

```
element.style.height = height + 10 + "px"; // write
```

- Avoid **reading & writing repeatedly in loops**, as it triggers multiple reflows.

3. **Use asynchronous code wisely**

- `setTimeout`, `fetch`, `Promise` → non-blocking
- Keep long-running code off main thread using **Web Workers**.

✅ Summary Table

Concept	What it does	Example/Tip
Reflow	Recalculates layout	Adding new elements, changing width
Repaint	Updates visual styles	Changing color, visibility
Call Stack	Where JS executes code	Single-threaded, run-to-completion
Event Loop	Handles async tasks	`setTimeout`, promises, DOM events
setTimeout	Executes after delay	Delay minimum; actual execution depends on event loop