React `useState` Notes

1. Value Updation (How React Stores & Updates State)

- In React, **global variables** cannot be used for reactive data because they do not trigger re-renders.
- Every component has its own internal memory stored inside a **fiber node**.
- A fiber node holds two key parts:
  - **Current Value** → Stores the latest, up-to-date state.
  - **Updation Queue** → Stores pending state updates.

How `useState` Works Internally

- `useState()` is directly connected to the fiber node's **current value** (for reading the state).
- `setState()` is connected to the fiber node's **updation queue** (for writing the state).
- When you call a setter function (like `setCount()`):
  1. It sends the update to the **updation queue**.
  2. React processes the queue and compares the **current value** with the **new value**.
  3. If they are **different** → React updates the current value in the fiber node.
  4. The component is then scheduled to **re-render**.

Why React Doesn't Update Current Value Directly

- Direct updates would bypass the crucial **comparison step**.
- The comparison check is what prevents **unnecessary re-renders** (see Section 2).
- The Fiber architecture must track the previous and new state to ensure component stability during the render process.

Why State Doesn't Reset

- During every re-render, the `useState()` call reads the **stored value** from the fiber node, not the initial value passed to `useState(initialValue)`.
- This persistence ensures the state remains updated across renders instead of resetting.

---

2. Computation Efficiency & Optimized Re-Rendering

Example

1. **Current value:** `"Rayyan"`
2. You update it to: `"M. Rayyan"` (Different $\implies$ **Re-render**)
3. You then update it back to: `"Rayyan"`

- React checks: *Is the new value* (`"Rayyan"`) *the same as the current value* (`"Rayyan"`)?
- Since both are the **same**, React does **not** re-render the component.

Conclusion

- React only re-renders when the new state value is **actually different** from the current state value.
- This optimization prevents:
  - Wasted CPU cycles
  - Unnecessary DOM updates

---

3. Batch Processing (Multiple State Updates = One Render)

Vanilla JavaScript Behavior

- Functions are executed **synchronously** and **one-by-one**:

```
 increase(); // runs
decrease(); // runs
reset();    // runs
```

React's Batch Processing Behavior

- When multiple state setter calls occur within the **same event handler** (or promise, timeout, etc. in modern React):

```
 setName("John");
setAge(25);
setCity("New York");
```

- React groups them together into one process (Batching) and performs:
  - **One** combined state update
  - **One** Virtual DOM comparison
  - **One final** re-render
- **Without Batch Processing (Hypothetical):** The DOM would be compared and updated three times (after each setter call), leading to slow performance.
- **With Actual Batch Processing:** React makes **one comparison** between the Old DOM and the final New DOM, making the application significantly faster and more optimized.

Without Batch Processing

```
Old DOM              New DOM
{pic1, 21, Chakwal}    {pic2, 21, Chakwal}

Old DOM              New DOM
{pic2, 21, Chakwal}    {pic2, 30, Chakwal}

Old DOM              New DOM
{pic2, 30, Chakwal}    {pic2, 30, Lahore}
```

With Batch Processing

```
 setName("John");
setAge(25);
setCity("New York");
```

```
Old DOM              New DOM
{pic1, 21, Chakwal}    {pic2, 30, Lahore}
```

Final Summary

React state works efficiently because:

1. State is stored persistently inside the **fiber node**.
2. `useState` accesses the **current value**; `setState` updates via the **queue**.
3. React **compares** old vs. new values before updating to prevent unnecessary work.
4. **Same values** do not trigger re-renders (performance optimized).
5. Multiple state updates are combined using **batch processing** for a single, fast render.