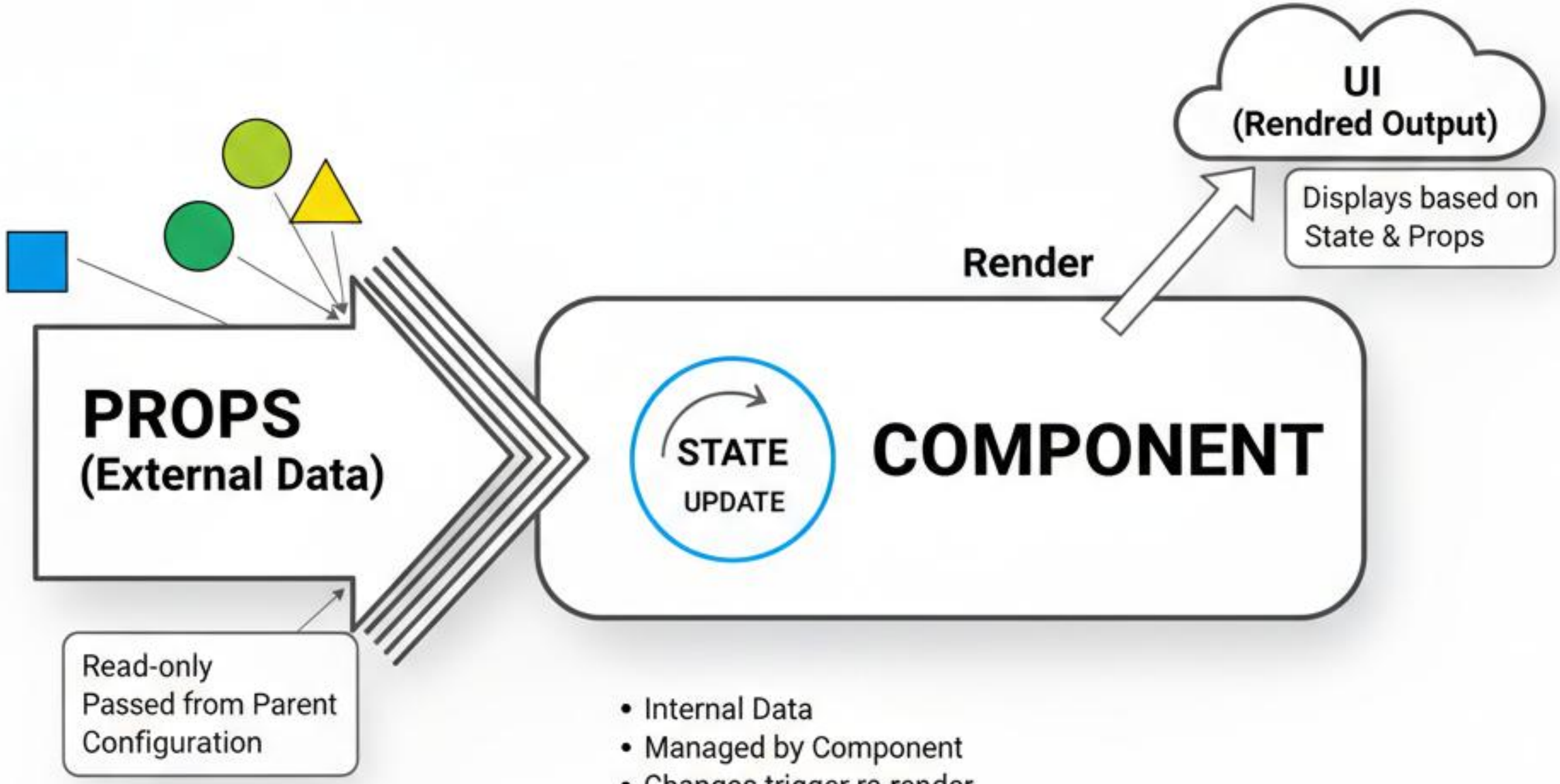
State
heart of React apps.

State

change in state directly leads to a change in the UI


UI
(Rendred Output)
Displays based on
State & Props
Render
PROPS
(External Data)
STATE
UPDATE
COMPONENT
Read-only
Passed from Parent
Configuration
• Internal Data
• Managed by Component
• Changes trigger re-render

Why Do We Need State?

- Dynamic UI.
- Example: In a counter button :
  - without state, UI won't update when clicked.
  - With state, UI will update when clicked.

Understanding useState Hook

- Syntax & destructuring:

```
const [count, setCount] = useState(0);
```

- State variable: A state variable is a special variable in React that holds data that changes over time and should trigger a re-render of the component when it changes.

- State updater function: Requests React to update the state and re-render the component. Ensures changes are tracked by React's internal system.

Note:

Never modify state directly (count = count + 1 ✕). Directly mutating the variable won't trigger re-render.

Initial State Possibilities

- Primitive values (number, string, boolean).
- Complex values (object, array).

Updating State:

- Direct updates:

```
setCount(count + 1);
```

Note:

Never modify state directly (count = count + 1 ❌).
Directly mutating the variable won't trigger re-render.

- Functional updates (when depending on previous state):

```
setCount(prev => prev + 1);
```

- Updating arrays & objects (use spread operator, immutability concept).

- Common mistakes: mutating directly (push, user.age = 22).

how React components are rendered and updated

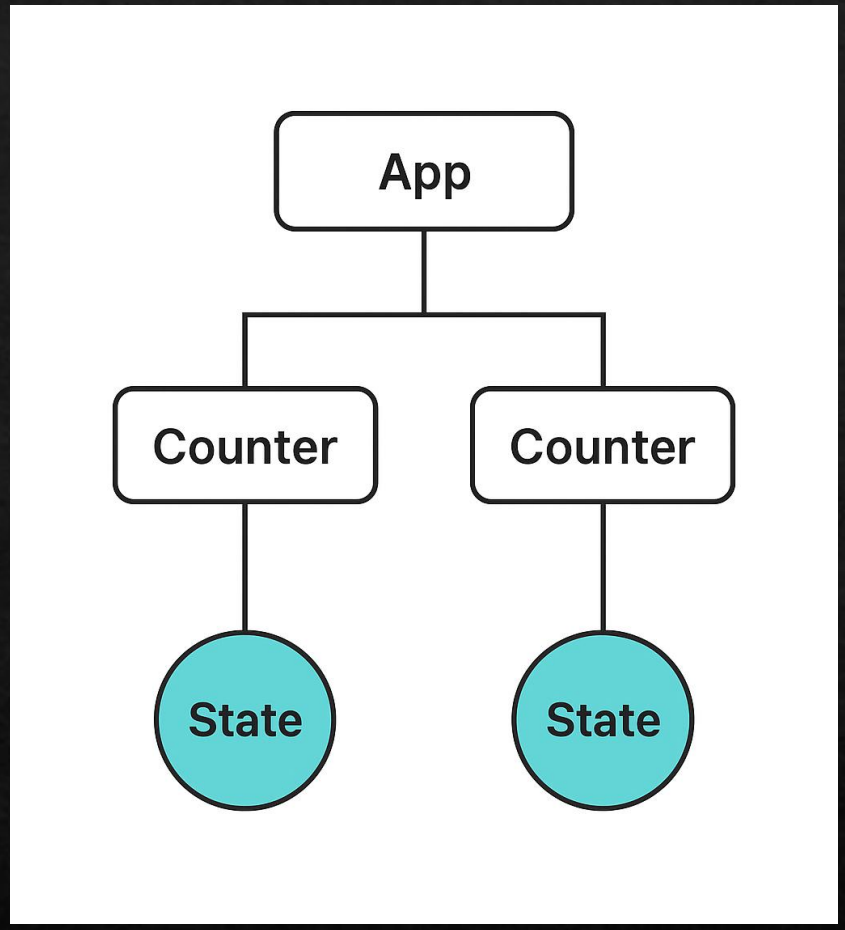
- State Declaration: Components in React manage their data using "state." In functional components, this is typically done using the useState hook.
- Updating State: When an event occurs (e.g., a user clicks a button, data is fetched), you use the state's updater function (e.g., setCount) to change the state value.
- Re-rendering: When the state is updated using its setter function, React detects this change. It then queues a re-render of the component where the state was updated, and any child components that depend on that state.
- UI Update: During the re-render, React re-executes the component's render logic (the JSX). It compares the new virtual DOM tree with the previous one and efficiently updates only the necessary parts of the actual DOM to reflect the new state. This ensures that the UI accurately displays the current state values.

Memory (Storage)

- Each component instance has its own independent state, even if it's the same component used multiple times.

Why It Works Like This ?

- React doesn't attach state to the component function itself.
- Instead, React keeps a state "bucket" for each component instance in the tree.
- When React re-renders, it matches components using their position in the tree and gives them back their own state memory.



Keyed Components

- If you render components in a list:

```
{items.map(item => <Counter key={item.id} />)}
```

- React uses the key prop to decide which state belongs to which component.
- If key changes → React treats it as a new component with a fresh state.

Common Pitfalls

1. Async Nature of State Updates

- Updating state doesn't happen immediately; React schedules it and re-renders later.

```
const [count, setCount] = useState(0);

const handleClick = () => {
  setCount(count + 1);
  console.log(count); // will log old value, not updated
};
```

Batching of Updates

- React batches multiple state updates in the same event to improve performance (avoids unnecessary re-renders).

```
const [count, setCount] = useState(0);

const handleClick = () => {
  setCount(count + 1);
  setCount(count + 1);
  setCount(count + 1);
};
```

Here, count will increase only by 1, not 3 (because React batches them, and each line uses old count).

How React Schedules It ?

When you call setState:

- React adds the update to an internal queue.
- React waits until the current event handler finishes (so it can batch multiple updates).
- Then React decides when to re-render the component.
- During render, React reconciles (diffs) the new virtual DOM with the old one and updates only what changed.

Solution (Use Functional Update)

```
setCount(prev => prev + 1);
setCount(prev => prev + 1);
setCount(prev => prev + 1);
// Now increases by 3
```

Why we can't use setCount(count++)?

Thank You