States

State refers to a built-in object that stores **data** or **values** that can **change over time**, and when it changes, the component **re-renders** to reflect the new values.

Imagine you're building:

- A **counter** that increases on a button click
- A **form** that stores user input
- A **dark/light theme** toggle

In all of these, **values change dynamically**, and you need React to **respond to that**. That's where state comes in.

Syntax:

```
const [stateVariable, setStateFunction] = useState(initialValue);
```

Example 1: Counter

```jsx
import React, { useState } from 'react';

function Counter() {
	const [count, setCount] = useState(0);

	return (
		<div>
		<h2>Count: {count}</h2>
		<button onClick={() => setCount(count + 1)}>Increment</button>
		</div>
	);
}
```

What's Happening:

- useState(0) sets count to 0 initially.
- Clicking the button runs setCount(count + 1).
- React re-renders the component with the new count.

Example 2: Form Input

```
import React, { useState } from 'react';

function Form() {
      const [name, setName] = useState(‘ ’);

      return (
            <div>
                  <input type="text” value={name} onChange={(e) =>
                  setName(e.target.value) }
                  />
                  <p>Hello, {name}</p>
            </div>
      );
}
```

States

State is local and private

Each component manages its **own state**. One component's state does **not affect others**, unless you **explicitly pass** data via props or global state management.

Rules of useState

- Only call useState at the top level of the component
- Don't call it inside loops or conditions
- Always use the setter function (setStateFunction) to update the value

States vs Props

Feature	state	props
Data source	Internal (local to component)	External (passed from parent)
Editable	Yes	No (read-only)
Used for	Interactivity, dynamic UI	Passing data to child components

Use of ... (Spread Operator)

The **...** spread operator is used to **copy all the values** of an object or array into a new object/array.

When updating state that is an **object or array**, we should **never mutate the original state directly**.
React relies on **immutability** to detect changes and re-render the UI.
So instead of changing the original object, we create a **new copy** with updated values — and this is where ... is useful.

Example: Updating Object State

```
const [user, setUser] = useState({ name: "Rahul", age: 25 });

const updateAge = () => {
setUser({ ... user, age: 26 }); // spread previous user, change only age
};
```

Explanation:

- ... user spreads all existing key-value pairs (name: "Rahul")
- Then age: 26 overrides the old age
- You're creating a new object, not mutating the old one

Example: Updating Array State

```
const [items, setItems] = useState(["Java", "Python"]);

const addItem = () => {
setItems([... items, "React"]); // spreads old items, adds new one
};
```

Explanation:

- ... items creates a copy of the existing array
- "React" is added to the end
- No mutation — safe for React to detect changes