React State

What is State?

In React, **state** is a built-in object that represents the current condition of a component. It stores data that can change over time and directly affects what is rendered on the UI.

Key points: - State is **local** to a component - State is **mutable** (can change) - When state changes, React **re-renders** the component - State is managed internally by the component

Unlike props (which are passed from parent to child and are read-only), state is owned and controlled by the component itself.

Why Do We Need State?

State allows React components to: - Respond to user interactions (clicks, typing, etc.) - Handle dynamic data - Update the UI automatically when data changes

Examples of state-driven UI: - Counter values - Form inputs - Toggle buttons - API data

---

useState Hook

`useState` is a React hook that allows functional components to manage state.

Syntax

```
const [stateValue, setStateValue] = useState(initialValue);
```

- `stateValue` → current state
- `setStateValue` → function to update state
- `initialValue` → starting value of state

Basic Counter Example

```
import { useState } from "react";

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

  const incrementCount = () => {
    setCount(count + 1);
  };
  const decrementCount = () => {
    setCount(count - 1);
  };

  return (
    <div>
      <h1>Counter App</h1>
      <p>Count: {count}</p>
      <button onClick={incrementCount}>Increment</button>
      <button onClick={decrementCount}>Decrement</button>
    </div>
  );
}

export default App;
```

Explanation

- `useState(0)` initializes `count` with `0`
- Clicking buttons updates state using `setCount`
- UI automatically re-renders when state changes

State vs Props

State

- Local to the component
- Mutable
- Managed using `useState`
- Private to the component

Props

- Passed from parent to child
- Immutable (read-only)
- Used to customize child components

👉 **Rule of Thumb:** - Use **state** for data that changes inside a component - Use **props** to pass data between components

---

State in Form Handling

Forms in React are usually **controlled components**, meaning input values are controlled using state.

Example: Simple Form

```
import { useState } from "react";

const SimpleForm = () => {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log(name, email);
    setName("");
    setEmail("");
  };

  return (
    <form onSubmit={handleSubmit}>
      <input value={name} onChange={(e) => setName(e.target.value)} />
      <input value={email} onChange={(e) => setEmail(e.target.value)} />
      <button type="submit">Submit</button>
    </form>
  );
};
```

Managing Multiple Inputs with One State Object

When a form has many input fields, managing each field with a separate `useState` can become repetitive and hard to maintain. In such cases, it is better to store all form values inside **one state object**.

Example: Form with Multiple Fields

```
import { useState } from "react";

function MultiInputForm() {
  const [formData, setFormData] = useState({
    name: "",
    email: "",
    password: "",
    age: "",
    city: ""
  });

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData((prevData) => ({
      ...prevData,
      [name]: value
    }));
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log(formData);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        name="name"
        placeholder="Name"
        value={formData.name}
        onChange={handleChange}
      />

      <input
        type="email"
        name="email"
        placeholder="Email"
        value={formData.email}
        onChange={handleChange}
      />

      <input
        type="password"
        name="password"
        placeholder="Password"
        value={formData.password}
        onChange={handleChange}
      />

      <input
        type="number"
        name="age"
        placeholder="Age"
        value={formData.age}
        onChange={handleChange}
      />

      <input
        type="text"
        name="city"
        placeholder="City"
        value={formData.city}
        onChange={handleChange}
      />

      <button type="submit">Submit</button>
    </form>
  );
}

export default MultiInputForm;
```

How This Works

- All input values are stored inside **one state object (`formData`)**
- Each input has a `name` attribute that matches a key in the state object
- `handleChange` dynamically updates the correct field using computed property names
- The spread operator (`...prevData`) ensures other fields are not lost during updates

---

Advantages of Using One State Object

1. **Less Code & Better Readability**
   You avoid creating multiple `useState` hooks for every input field.

2. **Easier to Scale Forms**
   Adding a new field only requires:

3. Adding a key in the state object

4. Adding an input element

5. **Single Change Handler**
   One `handleChange` function can manage all input fields.

6. **Better Form Submission Handling**
   All form values are already grouped together, making API calls and validation easier.

7. **Simpler Validation Logic**
   You can validate the entire form using one object instead of multiple variables.

8. **Cleaner State Management**
   The form behaves as a single controlled unit rather than many scattered states.

Adding Radio Buttons and Checkboxes

Real-world forms often include **radio buttons** and **checkboxes**. These can also be handled cleanly using the same state object approach.

Updated State Structure

```
const [formData, setFormData] = useState({
  name: "",
  email: "",
  password: "",
  age: "",
  city: "",
  gender: "",          // radio button
  skills: []            // checkboxes
});
```

Updated handleChange Function

```
const handleChange = (e) => {
  const { name, value, type, checked } = e.target;

  if (type === "checkbox") {
    setFormData((prev) => ({
      ...prev,
      skills: checked
        ? [...prev.skills, value]
        : prev.skills.filter((skill) => skill !== value)
    }));
  } else {
    setFormData((prev) => ({
      ...prev,
      [name]: value
    }));
  }
};
```

Radio Buttons Example (Gender)

```
<label>
  <input
    type="radio"
    name="gender"
    value="male"
    checked={formData.gender === "male"}
    onChange={handleChange}
  />
  Male
</label>
```

```
<label>
  <input
    type="radio"
    name="gender"
    value="female"
    checked={formData.gender === "female"}
    onChange={handleChange}
  />
  Female
</label>
```

Checkboxes Example (Skills)

```
<label>
  <input
    type="checkbox"
    value="React"
    checked={formData.skills.includes("React")}
    onChange={handleChange}
  />
  React
</label>

<label>
  <input
    type="checkbox"
    value="Node"
    checked={formData.skills.includes("Node")}
    onChange={handleChange}
  />
  Node
</label>

<label>
  <input
    type="checkbox"
    value="MongoDB"
    checked={formData.skills.includes("MongoDB")}
    onChange={handleChange}
  />
  MongoDB
</label>
```

Why This Pattern Is Powerful

- Radio buttons store **one selected value**
- Checkboxes store **multiple selected values in an array**

- Entire form remains **fully controlled**
- One state object + one handler manages everything

Interview Tip ⭐

> "For large forms, I prefer using a single state object with a generic change handler. This makes the form scalable, easier to validate, and simpler to submit to APIs."

This approach is **industry standard** for medium to large React forms.

Lifting State Up

What Does "Lifting State Up" Mean?

Lifting state up means moving shared state from child components to their **closest common parent**, so that multiple components can access and update the same data in a synchronized way.

In simple terms:

> When two or more components need the same data, keep the state in their parent instead of duplicating it.

General (Non-React) Scenario

Imagine a **remote control** and a **TV display**: - The remote changes the channel - The TV displays the channel

👉If both keep their own channel value: - Remote shows Channel 5 - TV still shows Channel 2

👉🏼Better solution: - Store the channel value **in one place** - Remote updates it - TV reads it

That single place is like the **parent component** in React.

Problem Without Lifting State Up (React Scenario)

Two components manage their own state independently:

Child 1: Input Component

```
function TemperatureInput() {
  const [temperature, setTemperature] = useState("");

  return (
    <input
```

```
      value={temperature}
      onChange={(e) => setTemperature(e.target.value)}
    />
  );
}
```

Child 2: Display Component

```
function TemperatureDisplay() {
  const [temperature, setTemperature] = useState("");
  const fahrenheit = (temperature * 9) / 5 + 32;

  return (
    <div>
      <p>{temperature}°C</p>
      <p>{fahrenheit}°F</p>
    </div>
  );
}
```

Issues Here 🦡

- Duplicate state
- Components are not synced
- Hard to maintain

Lifting State Up (Correct React Approach)

Step 1: Move State to Parent

```
function App() {
  const [temperature, setTemperature] = useState("");

  return (
    <div>
      <TemperatureInput
        temperature={temperature}
        onTemperatureChange={setTemperature}
      />
      <TemperatureDisplay temperature={temperature} />
    </div>
  );
}
```

Step 2: Use Props in Children

Input Component

```
function TemperatureInput({ temperature, onTemperatureChange }) {
  return (
    <input
      value={temperature}
      onChange={(e) => onTemperatureChange(e.target.value)}
    />
  );
}
```

Display Component

```
function TemperatureDisplay({ temperature }) {
  const fahrenheit = (temperature * 9) / 5 + 32;

  return (
    <div>
      <p>{temperature}°C</p>
      <p>{fahrenheit.toFixed(1)}°F</p>
    </div>
  );
}
```

Why Lifting State Up Works

- 👈🏼Single source of truth
- 👈🏼Always synchronized UI
- 👈🏼Easier debugging
- 👈🏼Better scalability

Real-World Project Scenarios

1. Search Bar + Results List

- Search input updates query
- Results component displays filtered data

👋🏻Lift search query state to parent

---

2. Shopping Cart (E-commerce)

- Product list adds items

- Cart icon shows count
- Checkout page reads same cart

👋🏻Cart state lifted to parent or global store

---

3. Authentication State

- Login form updates user state
- Navbar shows user name
- Protected routes depend on login

👋🏻User state lifted to App level

---

4. Form Wizard (Multi-Step Forms)

- Step 1: Personal info
- Step 2: Address
- Step 3: Review

👋🏻All form data lifted to parent wizard component

Interview Summary ⭐

> "Lifting state up is used when multiple components need access to the same changing data. We move the state to their closest common parent and pass it down via props to keep the UI consistent and maintainable."

Key Takeaway

If you ever feel like you are **copy-pasting the same state in multiple components**, it is a strong signal that you should **lift the state up**.

(We will cover this in detail separately.)

Key Takeaways

- State controls dynamic data in React
- `useState` is the primary hook for managing state
- Updating state triggers re-render
- State is local, props are external
- Proper state management leads to predictable UI behavior