CipherSchools

Conditional Rendering

In ReactJS, conditional rendering means showing or hiding UI elements based on a condition (like a boolean or a state value). React renders components based on the logic you provide.

1. **Using if statements**

   Useful when you want to include logic before returning JSX.
   Example:

   ```
   function Greeting(props) {
     const isLoggedIn = props.isLoggedIn;

     if (isLoggedIn) {
       return <h1>Welcome back!</h1>;
     } else {
       return <h1>Please sign in.</h1>;
     }
   }
   ```

   Use Case

   - Best when the condition is complex or you have to return early.

2. **Ternary Operator (? :)**

   Best for inline conditionals inside JSX.
   Example:

   ```
   function Greeting(props) {
     return (
       <h1>
         {props.isLoggedIn ? "Welcome back!" : "Please sign in."}
       </h1>
   ```

```
  );
}
```

Use Case:

- Great for simple conditions.
- Avoid if the condition or returned JSX is too long – it gets messy.

3. Short-Circuit Evaluation (&&)

Shows something only if the condition is true. If false, nothing renders.

Example:

```
function Notification(props) {
  return (
    <div>
      {props.unreadCount > 0 && (
        <p>You have {props.unreadCount} unread messages.</p>
      )}
    </div>
  );
}
```

Use Case:

- Best when you want to conditionally show something or nothing.
- Not useful when you want an else condition.

4. IIFE (Immediately Invoked Function Expression)

For more complex logic inline, you can use:

Example:

```
{
  (() ⇒ {
    if (condition) return <A />;
    else if (other) return <B />;
    else return <C />;
  })()
}
```

Use Case:

- Advanced use only when ternary becomes unreadable.

5. Switch Case Rendering

For multiple options:

Example:

```
function StatusMessage({ status }) {
  switch (status) {
    case "loading":
      return <p>Loading...</p>;
    case "success":
      return <p>Success!</p>;
    case "error":
      return <p>Error occurred.</p>;
    default:
      return <p>Idle...</p>;
 }
}
```

What is “Lifting State Up”?

In React, "Lifting State Up" means moving the state from a child component to a common parent component. This is done so that multiple components can share and sync the same data.

Why Lift State Up?

Let’s say:

1. You have 2 or more child components.
2. They need to communicate or sync data.
3. If they maintain separate states, it can get out of sync.
4. Instead, we lift the state to their common parent, and pass data + callbacks down via props.

Example Scenario

Without lifting:

```
function Child() {
  const [count, setCount] = useState(0);
  return <button onClick={() ⇒ setCount(count + 1)}>Click:
{count}</button>;
}
```

Each child maintains its own state - not shared.

With lifting:

Step-by-Step:

1. Move the state to the parent.
2. Pass state & state-updating function (callback) to children.

```
// Parent Component
function Parent() {
  const [count, setCount] = useState(0);

  return (
    <>
      <Child count={count} setCount={setCount} />
      <Display count={count} />
    </>
  );
}

// Child Component (updates state)
function Child({ count, setCount }) {
  return (
    <button onClick={() ⇒ setCount(count + 1)}>
      Click Me
    </button>
  );
}
```

```
// Display Component (reads state)
function Display({ count }) {
  return <h2>Count is: {count}</h2>;
}
```

Now both components share the same state from the parent.

When Should You Lift State?

- When two sibling components need to share state
- When child needs to inform parent
- When a component wants to trigger something in another via shared data

Callback Pattern for Lifting

Child can send data back to parent using a callback:

```
// Parent
function Parent() {
  const [name, setName] = useState("");

  const handleNameChange = (newName) => {
    setName(newName);
  };

  return <Child onNameChange={handleNameChange} />;
}

// Child
function Child({ onNameChange }) {
  return (
    <input
      onChange={(e) => onNameChange(e.target.value)}
      placeholder="Enter name"
    />
  );
}
```

Parent now controls the state, but the child can trigger updates.

Best Practices

- Always lift the state up to the lowest common ancestor that needs it.
- Avoid lifting state too high if it makes components bloated.
- Use callbacks for child → parent communication.

Controlled vs Uncontrolled Inputs

In React, when working with forms (<input>, <textarea>, <select>, etc.), there are two ways to manage their state:

1. Controlled Components

   In a controlled component, form data is handled by the React component state.

   How it works:

   - The value of the input is controlled via useState().
   - Every input change updates the state using an onChange handler.

   Example:

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

   function ControlledInput() {
     const [value, setValue] = useState('');

     const handleChange = (e) ⇒ {
   ```

```
    setValue(e.target.value);
  };

  return (
    <main>
      <h1>Student Form</h1>
      <input placeholder=”Name” type="text" value={value}
onChange={handleChange} />
    </main>
  );
}
```

Benefits:

- Full control over form behavior.
- Easier validation, formatting, disabling, etc.
- Best suited for dynamic or conditional rendering.

2. Uncontrolled Components

In an uncontrolled component, the DOM handles the form data. React doesn't track the input's state directly.

How it works:
You use ref to access the DOM node and read the value directly.

Example:

```
import { useRef } from 'react';

function UncontrolledInput() {
  const inputRef = useRef();

  const handleSubmit = () ⇒ {
    alert(`You typed: ${inputRef.current.value}`);
  };
  return (
    <div>
      <input type="text" ref={inputRef} />
      <button onClick={handleSubmit}>Submit</button>
    </div>
  );
}
```

Benefits:

- Less code if you don’t need frequent access to the input value.
- Useful for quick forms or when integrating with non-React libraries.

Rendering Behaviour:

1. Controlled Inputs:
    - Re-render on every keystroke:
        - Every time a user types, the state updates → component re-renders.
    - Because React controls the value, rendering stays in sync with state.

    Pros:
    - Ensures consistent UI with React state.
    - Better integration with features like form validation, conditional rendering, and custom formatting.

    Cons:
    - Slight performance cost if you're rendering many inputs.
2. Uncontrolled Inputs:
    - Render once, input value changes don't cause re-renders unless manually accessed.
    - React doesn't re-render on input value changes unless a ref.current.value is explicitly used.

    Pros:
    - More performant in large forms or quick tasks.

    Cons:
    - Harder to sync with UI logic or conditional rendering.

Key Differences:

Feature	Controlled Input	Uncontrolled Input
Data Source	React State	DOM
Controlled by	value + onChange	ref
Real-time validation	Easy	Difficult
Initial value	Set via useState()	Set via defaultValue
Performance	Slightly heavier for many inputs	More performant in large forms

List Rendering – map, filter & key

List rendering means displaying a list of items dynamically from an array in JSX.
React uses the JavaScript array methods like map() and filter() to transform and render lists.

Using .map() in React
map() is most commonly used to render lists.

Syntax:

```
{array.map((item, index) ⇒ (
  <div key={index}>{item}</div>
))}
```

Example:

```
const fruits = ['Apple', 'Banana', 'Mango'];

return (
  <ul>
    {fruits.map((fruit, index) ⇒ (
      <li key={index}>{fruit}</li>
    ))}
  </ul>
);
```

map returns a new array of JSX elements.
We use it inside JSX curly braces {}.

Using .filter() Before Rendering
You can use filter() to render a subset of items.

Example:

```
const students = [
  { id: 1, name: 'Alice', passed: true },
  { id: 2, name: 'Bob', passed: false }
];

return (
  <ul>
    {students
      .filter(student ⇒ student.passed)
      .map(student ⇒ (
        <li key={student.id}>{student.name}</li>
      ))}
  </ul>
);
```

1. .filter() first filters out data.
2. Then .map() is used to render JSX from the filtered list.

Why is key Important?

Virtual DOM Optimization

- React uses key to match elements during diffing.

- It avoids re-rendering unchanged components, improving performance.

Common Bug Without Proper key:

- Imagine you're rendering a list of inputs. If keys are incorrect or reused:
    - You might type in one input, but it appears in another field after re-render.
    - This is because React can’t correctly track which item changed.

React Fragment and Empty Tags

React Fragments allow you to group a list of children elements without adding extra nodes to the DOM. It is useful when you want to return multiple elements from a component but don’t want to wrap them in an extra <div> or similar tag.

Why not just use a <div>?

- Using extra <div> can mess up your styling or layout, especially in:
- Flex/Grid systems (adds unintended containers)
- Semantic HTML (extra nodes can confuse screen readers)

1. React Fragment Syntax

   a) Explicit Fragment Syntax

   ```
   import React from "react";

   function App() {
     return (
       <React.Fragment>
         <h1>Hello</h1>
         <p>Welcome to React</p>
       </React.Fragment>
     );
   }
   ```

   b) Short Syntax (Empty Tags)

```
function App() {
  return (
    <>
      <h1>Hello</h1>
      <p>Welcome to React</p>
    </>
  );
}
```

This is functionally the same as React.Fragment.

2. When to use React.Fragment instead of <> </>?
   Use React.Fragment if you want to:
   - Add keys to fragments (e.g., inside .map()):

```
items.map(item ⇒ (
  <React.Fragment key={item.id}>
    <h2>{item.title}</h2>
    <p>{item.description}</p>
  </React.Fragment>
));
```

You cannot use the shorthand <> if you're assigning a key.