**Day-04 React-Class(State Management)**

**What are Hooks?** Hooks are a powerful addition to React that allows you to use state and other React features without writing a class. They let you use React features in functional components, making code more readable and reusable.

**Example:**

javascriptCopy code

import React, { useState } from 'react';

const Counter = () => {

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

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

};

export default Counter;

In this example, **useState** is a Hook that allows **Counter** component to use state. It initializes **count** state to 0, and **setCount** function to update it.

Hooks make it easier to manage state and lifecycle events in functional components, improving code organization and readability.

**Benefits of Hooks:**

1. **Simplified Code**: Eliminates the need for class components and reduces boilerplate code.
2. **Improved Readability**: Makes it easier to understand and reason about component logic.
3. **Reusability**: Encourages the creation of custom Hooks for sharing logic between components.
4. **Better Performance**: Optimizes performance by avoiding unnecessary re-renders.

In conclusion, Hooks revolutionize React development by offering a more intuitive and functional approach to building components. Embrace Hooks to write cleaner, more efficient React code

[**useState**](https://react.dev/reference/react/useState)

const [state, setState] = useState(initialState)

### **Usage**

### **Adding state to a component**

**Call useState at the top level of your component to declare one or more state variables.**

import { useState } from 'react';

function MyComponent() {

const [age, setAge] = useState(42);

const [name, setName] = useState('Taylor');

// ...

The convention is to name state variables like [something, setSomething] using array destructuring.

**useState returns an array with exactly two items:**

**The current state of this state variable, initially set to the initial state you provided. The set function that lets you change it to any other value in response to interaction. To update what’s on the screen, call the set function with some next state:**

function handleClick() {

setName('Robin');

}

**Note**

Calling the set function does not change the current state in the already executing code:

function handleClick() {

setName('Robin');

console.log(name); // Still "Taylor"!

}

It only affects what useState will return starting from the next render.

When the button is clicked, the **handleClick** function is invoked, which changes the value of the **name** state variable to **'Robin'**. However, React state updates are asynchronous, meaning that when you call **setName('Robin')**, React doesn't immediately update the **name** variable. Therefore, when you log **name** immediately after calling **setName('Robin')**, you'll still see the previous value of **name**, which is **'Taylor'**. React schedules an update to re-render the component with the new state values, but it doesn't happen synchronously within the same execution context.

**Examples:**

**Counter**

import { useState } from 'react';

export default function Counter() {

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

function handleClick() {

setCount(count + 1);

}

return (

<button onClick={handleClick}>

You pressed me {count} times

</button>

);

}

**Text**

import { useState } from 'react';

export default function MyInput() {

const [text, setText] = useState('hello');

function handleChange(e) {

setText(e.target.value);

}

return (

<>

<input value={text} onChange={handleChange} />

<p>You typed: {text}</p>

<button onClick={() => setText('hello')}>

Reset

</button>

</>

);

}

**Checkbox**

import { useState } from 'react';

export default function MyCheckbox() {

const [liked, setLiked] = useState(true);

function handleChange(e) {

setLiked(e.target.checked);

}

return (

<>

<label>

<input

type="checkbox"

checked={liked}

onChange={handleChange}

/>

I liked this

</label>

<p>You {liked ? 'liked' : 'did not like'} this.</p>

</>

);

}

**two states**

import { useState } from 'react';

export default function Form() {

const [name, setName] = useState('Taylor');

const [age, setAge] = useState(42);

return (

<>

<input

value={name}

onChange={e => setName(e.target.value)}

/>

<button onClick={() => setAge(age + 1)}>

Increment age

</button>

<p>Hello, {name}. You are {age}.</p>

</>

);

}

### **Reference**

**useState(initialState)useState is a Hook, so you can only call it at the top level of your component or your own Hooks. You can’t call it inside loops or conditions. If you need that, extract a new component and move the state into** [**it.In**](http://it.In) **Strict Mode, React will call your initializer function twice in order to help you find accidental impurities. This is development-only behavior and does not affect production. If your initializer function is pure (as it should be), this should not affect the logic of your component. The result from one of the calls will be ignored.**

### **Troubleshooting**

**I’ve updated the state, but logging gives me the old value Calling the set function does not change state in the running code:**

function handleClick() {

console.log(count); // 0

setCount(count + 1); // Request a re-render with 1

console.log(count); // Still 0!

setTimeout(() => {

console.log(count); // Also 0!

}, 5000);

}

**This is because states behaves like a snapshot. Updating state requests another render with the new state value, but does not affect the count JavaScript variable in your already-running event handler.**

**If you need to use the next state, you can save it in a variable before passing it to the set function:**

const nextCount = count + 1;

setCount(nextCount);

console.log(count); // 0

console.log(nextCount); // 1

**I’ve updated the state, but the screen doesn’t updateI’m getting an error: “Too many re-renders”**

**You might get an error that says: Too many re-renders. React limits the number of renders to prevent an infinite loop. Typically, this means that you’re unconditionally setting state during render, so your component enters a loop: render, set state (which causes a render), render, set state (which causes a render), and so on. Very often, this is caused by a mistake in specifying an event handler:**

// 🚩 Wrong: calls the handler during render

return <button onClick={handleClick()}>Click me</button>

// ✅ Correct: passes down the event handler

return <button onClick={handleClick}>Click me</button>

// ✅ Correct: passes down an inline function

return <button onClick={(e) => handleClick(e)}>Click me</button>

**If you can’t find the cause of this error, click on the arrow next to the error in the console and look through the JavaScript stack to find the specific set function call responsible for the error.**

**My initialiser or updater function runs twice**

**In Strict Mode, React will call some of your functions twice instead of once:**

**This is expected and shouldn’t break your code.**

**This development-only behaviour helps you keep components pure. React uses the result of one of the calls, and ignores the result of the other call.**

**I’m trying to set the state to a function, but it gets called instead**

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

const ListComponent = () => {

const [items, setItems] = useState(['Apple', 'Banana', 'Orange']);

return (

<>

<input type="text" />

<ul>

{items.map((item, index) => (

<li key={index}>

{item}

</li>

))}

</ul>

</>

);

};

export default ListComponent;

Explanation:

1. **Lists**: The **items** state is an array containing items. We map over this array to render a list of items in the **<ul>** element.
2. **Keys**: In the list rendering, each **<li>** element has a **key** attribute set to the **index** of the item in the array. Keys help React identify which items have changed, are added, or are removed. Using the index as a key is suitable for static lists, but for dynamic lists where items can be added, removed, or rearranged, it's better to use unique IDs.
3. **Fragments**: Fragments are represented by **<>...</>**. In this example, we use a fragment to wrap the input, button, and list elements since a component can only return a single root element.