### React Hooks Overview

#### Introduction

* Hooks to enable function components to manage state and access other React features.
* **Purpose**: Hooks allow function components to "hook" into React features like state and lifecycle methods, reducing the need for class components.

#### Key Rules for Using Hooks

1. **Function Components Only**: Hooks must be used inside React function components. They cannot be used in regular JavaScript functions or class components.
2. **Top-Level Calls**: Hooks should be called at the top level of the component, not inside loops, conditions, or nested functions.
3. **No Conditional Usage**: Hooks must be called in the same order on every render. Avoid calling them conditionally.

### React useState Hook

#### 1. Importing useState

To use useState, import it from React:

import React, { useState } from 'react';

#### 2. Basic Syntax

useState returns an array with two elements:

const [state, setState] = useState(initialValue);

* state: Current state value.
* setState: Function to update the state.
* initialValue: Initial state value (can be any data type).

#### 3. Example: Counter

A basic counter example:

import React, { useState } from 'react';

function Counter() {

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

const increment = () => setCount(count + 1);

const decrement = () => setCount(count - 1);

return (

<div>

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

<button onClick={increment}>Increment</button>

<button onClick={decrement}>Decrement</button>

</div>

);

}

export default Counter;

#### 4. Handling State Updates

* State updates are asynchronous (don’t wait for some one).
* Use setState to update state, never modify state directly.

#### 5. Updating State Based on Previous State

Use a callback (A callback is a function passed as an argument to another function.) with setState when the update depends on the previous state:

setState(prevState => prevState + 1);

#### 6. Lazy Initialization

For expensive initializations, pass a function to useState:

const [state, setState] = useState(() => {

// Expensive computation

return computedValue;

});

#### 7. Multiple useState Calls

Use multiple useState hooks to manage different states:

function Form() {

const [username, setUsername] = useState('');

const [password, setPassword] = useState('');

}

#### 8. Using useState with Objects and Arrays

* **Objects**: Manually merge properties using the spread operator:

const [user, setUser] = useState({ name: '', age: '' });

setUser(prevUser => ({ ...prevUser, age: 30 }));

* **Arrays**: Handle array updates manually:

const [items, setItems] = useState([]);

setItems([...items, newItem]); // Adding an item

#### 9. Common Pitfalls

* **Re-rendering Issues**: Avoid storing derived data in state.
* **Infinite Loops**: Be cautious with side effects that depend on state.

#### 10. Summary

useState provides state management for functional components. It returns the current state and a function to update it, making functional components more powerful.

### React useEffect Hook

The useEffect Hook allows you to perform side effects in your components.

A React side-effect occurs when we use something that is outside the scope of React.js in our React components.

#### 1. Basic Usage

useEffect takes two arguments:

1. A function (effect).
2. An optional dependency array. Syntax:

useEffect(() => {

// Effect code here

return () => {

// Cleanup code here (optional)

};

}, [dependencies]);

Example:

import React, { useEffect } from 'react';

function ExampleComponent() {

useEffect(() => {

document.title = 'Hello, World!';

}, []); // Runs once on mount

return <div>Hello, World!</div>;

}

#### 2. How It Works

* **Effect Function**: Runs after the component renders.
* **Cleanup Function**: Optionally returned to clean up resources.

Example with Cleanup:

javascript

Copy code

useEffect(() => {

const timer = setInterval(() => {

console.log('Tick');

}, 1000);

return () => clearInterval(timer);

}, []);

#### 3. Dependency Array

* **No Dependency Array**: Effect runs after every render.
* **Empty Array ([])**: Effect runs once after initial render.
* **Specific Dependencies**: Effect runs when specified dependencies change.

Example with Dependencies:

useEffect(() => {

console.log('This runs when `count` changes');

}, [count]); // Runs when `count` changes

#### 4. Common Use Cases

* **Fetching Data**:

useEffect(() => {

async function fetchData() {

const response = await fetch('https://api.example.com/data');

const data = await response.json();

console.log(data);

}

fetchData();

}, []);

* **Subscribing to Events**:

useEffect(() => {

const handleResize = () => console.log('Window resized');

window.addEventListener('resize', handleResize);

return () => window.removeEventListener('resize', handleResize);

}, []);

* **Updating Document Title**:

useEffect(() => {

document.title = `You clicked ${count} times`;

}, [count]); // Runs whenever `count` changes

#### 5. Best Practices

* **Avoid Unnecessary Dependencies**: Include only necessary dependencies.
* **Handle Cleanup Properly**: Always clean up side effects.
* **Use Multiple useEffect Hooks**: Separate concerns into different hooks.

#### 6. Advanced Concepts

* **Conditionally Firing Effects**: Use conditional logic inside the effect function if needed.
* **Performance Optimization**: Use React.memo, useCallback, and useMemo to optimize performance.
* **React Strict Mode**: Effects might run multiple times in development mode for debugging.

#### 7. Common Pitfalls

* **Infinite Loops**: Avoid effects that cause infinite loops by correctly managing dependencies.
* **Stale Closures**: Ensure variables used in effects are included in the dependency array.
* **Async Functions**: Define async functions inside the effect and call them.

#### 8. Troubleshooting

* **Debugging**: Use console logs and React DevTools to debug effects.
* **Reactivity Issues**: Ensure accurate dependency tracking.

#### 9. Summary

useEffect manages side effects in functional components and replaces lifecycle methods from class components. Proper use of useEffect ensures efficient and maintainable code.

### React useMemo Hook

#### 1. Purpose

useMemo helps optimize performance by memoizing the result of expensive computations and preventing unnecessary recalculations.

#### 2. Syntax

const memoizedValue = useMemo(() => {

// Expensive computation

return computedValue;

}, [dependencies]);

* memoizedValue: The cached result of the computation.
* The computation only runs when dependencies change.

#### 3. Example

import React, { useMemo } from 'react';

function ExpensiveComponent({ value }) {

const computedValue = useMemo(() => {

// Expensive calculation

return value \* 2;

}, [value]);

return <div>{computedValue}</div>;

}

#### 4. When to Use

* Use useMemo to optimize performance when dealing with expensive computations or functions that depend on props or state.

#### 5. Common Pitfalls

* **Overuse**: Avoid using useMemo for simple computations or small components. It adds complexity without significant performance benefits.

### React useCallback Hook

#### 1. Purpose

useCallback memoizes callback functions to prevent unnecessary re-renders of child components that rely on those callbacks.

#### 2. Syntax

const memoizedCallback = useCallback(() => {

// Callback function

}, [dependencies]);

* memoizedCallback: The memoized function that only changes when dependencies change.

#### 3. Example

import React, { useCallback } from 'react';

function ParentComponent() {

const handleClick = useCallback(() => {

console.log('Button clicked');

}, []); // Empty dependency array

return <ChildComponent onClick={handleClick} />;

}

#### 4. When to Use

* Use useCallback when passing callbacks to child components to prevent unnecessary re-renders.

#### 5. Common Pitfalls

* **Overuse**: Like useMemo, overusing useCallback can lead to unnecessary complexity. Use it where it provides a clear benefit in preventing re-renders.