**Hook Rule - Call Multiple Hooks in Same Sequence**

Hooks should always be called in the same order and at the top level of a React function component or a custom hook. This consistency ensures that hook state is maintained correctly across renders.

Here's an example to illustrate:

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

function ExampleComponent() {

// 1. First hook: useState for count

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

// 2. Second hook: useState for name

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

// 3. Third hook: useEffect for document title

useEffect(() => {

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

}, [count]);

// 4. Fourth hook: useEffect for greeting

useEffect(() => {

console.log(`Hello, ${name}!`);

}, [name]);

return (

<div>

<p>You clicked {count} times</p>

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

<p>Hello, {name}</p>

<button onClick={() => setName('Jane')}>Change Name</button>

</div>

);

}

export default ExampleComponent;

In this example, we have multiple hooks (**useState** and **useEffect**). They're called in a consistent sequence every time the component renders:

1. **useState** for **count**
2. **useState** for **name**
3. **useEffect** for updating the document title
4. **useEffect** for logging a greeting

The sequence is crucial. If, for any reason, hooks are called conditionally or in a different order between renders, React wouldn't know how to correctly preserve the state of each hook between renders, leading to unpredictable behaviors.

A violation of the rule might look like this (which you should avoid):

if (someCondition) {

const [count, setCount] = useState(0); // BAD: Hooks shouldn't be called conditionally

}

In summary, always ensure that:

* Hooks are called at the top level (not inside loops, conditions, or nested functions).
* Hooks are always called in the same order in every component render.

The key takeaway is that once you decide on an order for your hooks, you should maintain that order across all renders. If you were to conditionally render hooks or frequently switch their order, it would break React's internal mechanism for tracking hooks, leading to unpredictable behaviors and potential bugs. In practice, this means that once you've established an order, any modifications should be made carefully, with testing to ensure functionality remains as expected.

In React, you should not call hooks inside loops, conditions, or nested functions. Always use hooks at the top level of your React function. This is one of the core Rules of Hooks.

The reason for this rule is that hooks rely on the order of their calls between multiple renders to properly maintain their internal state and side effects. If you were to call hooks conditionally or inside loops, the order in which they're called could potentially change between renders, leading to unexpected behaviors and bugs.

Here's an incorrect and problematic usage of hooks:

function IncorrectComponent({ items }) {

items.forEach(item => {

// WRONG! Do not call hooks inside loops

const [someState, setSomeState] = useState(item.value);

});

// ... rest of the component

}

If you feel the need to use a hook within a loop or conditional, you may want to rethink your component design or extract the repeated logic into a custom hook or a separate component.

For instance, instead of the incorrect approach shown above, you can do something like this:

function ItemComponent({ value }) {

const [someState, setSomeState] = useState(value);

// ... rest of the component logic

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

}

function CorrectComponent({ items }) {

return (

<div>

{items.map(item => <ItemComponent key={item.id} value={item.value} />)}

</div>

);

}

In the corrected approach, each item gets its own component with its own state, ensuring hooks are called at the top level and in a consistent order.