useState

useState is a built-in hook in React that allows functional components to manage state. It is a fundamental and widely used hook in React for managing component-level state. useState is typically used when you need to store and update some piece of data within a component. It returns an array with two elements: the current state value and a function to update it.

The argument you pass to useState is the initial state value, and it's only used during the initial render. Subsequent renders will use the current state value, and you can update it with the corresponding setter function.

You can use useState to manage multiple pieces of state in a single component, and it's a fundamental building block for building dynamic and interactive React components.

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

function Counter()
{
  // The useState hook takes an initial state value as an argument.
  // It returns an array with two elements: the current state value and a
function to update it.
  const [count, setCount] = useState(0);

  // You can use the state variable (count) in your component's JSX.
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
    </div>
  );
}

export default Counter;
```

useState should not be used conditionally

useState should not be used conditionally inside functions or loops in a React component. The reason for this is that hooks in React should always be called at the top level of your functional component.

Here's why you should avoid conditional usage of useState.

Order of Execution

React relies on the order in which hooks are called to maintain the state between re-renders. Hooks like useState should always be called in the same order on every render. If you conditionally call hooks, React won't be able to determine the correct state updates, leading to unexpected behavior and bugs.

Consistency

Hooks like useState are designed to be used consistently throughout the component. Placing them conditionally inside functions can make your code less predictable and harder to reason about.

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

function Counter()
{
  // This is incorrect! useState is called conditionally.
  if (someCondition)
  {
    const [count, setCount] = useState(0);
  }

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

To conditionally manage state in a React functional component, you should define your state variables using useState at the top level of your component and then conditionally render different parts of your component based on the state.

```jsx
import React, { useState } from 'react';

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

  const handleIncrement = () =>
  {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleIncrement}>Increment</button>
    </div>
  );
}
```

In this corrected example, the state variable count is defined at the top level of the component, and conditional rendering is achieved by conditionally rendering different parts of the component based on the value of count or other state variables. This approach ensures that hooks like useState are always called in the same order on every render and maintains the expected behaviour of React components.

```jsx
import React, { useState } from 'react';

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

  if (count === 5)
  {
    const [message, setMessage] = useState('You reached 5!');
  }

  const handleIncrement = () =>
  {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      {count === 5 && <p>{message}</p>}
      <button onClick={handleIncrement}>Increment</button>
    </div>
  );
}
```

In this example, we have a Counter component that attempts to conditionally show a message when the count reaches 5. However, this code demonstrates an incorrect order of execution of hooks. Let's break down what's happening:

1. const [count, setCount] = useState(0); is called at the top level of the component. This initializes the count state variable with an initial value of 0.

2. Inside the component's render function, we conditionally render the message state variable when count equals 5: {count === 5 && {message} }.

3. However, the const [message, setMessage] = useState('You reached 5!'); is also called conditionally inside the if (count === 5) block. This is an incorrect usage of hooks because hooks must always be called in the same order on every render.

The incorrect order of execution can lead to issues:

1. React relies on the order of hooks to maintain the state between re-renders. In this case, when count reaches 5, React doesn't know where to insert the message state, as it was not called during the initial render when count was 0. This can lead to unpredictable behaviour.

2. Additionally, if you increment the count beyond 5, you will encounter a "Rendered more hooks than during the previous render" error because the number of hooks calls within a component must remain consistent between renders.

To correct the order of execution, you should always call hooks at the top level of your component. Here's the corrected example:

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

function Counter() {
  const [count, setCount] = useState(0);
  const [message, setMessage] = useState(''); // Moved to the top level

  if (count === 5) {
    setMessage('You reached 5!');
  }

  const handleIncrement = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      {count === 5 && <p>{message}</p>}
      <button onClick={handleIncrement}>Increment</button>
    </div>
  );
}
```

In this corrected example, we have moved the message state to the top level of the component, ensuring that hooks are called in the same order on every render. This avoids issues with the order of execution and provides the expected behaviour when count reaches 5.

In summary, the "Order of Execution" in React is essential to maintain the consistency of hooks and state management. Hooks should always be called at the top level of your component to ensure predictable behaviour and avoid errors related to the incorrect order of hooks.

Don’t set state manually.

In React, it's generally recommended not to update the state manually using direct assignment or mutation. Instead, you should use the state updater functions provided by React, such as setState in class components or the state update function returned by the useState hook in functional components.

Here's why it's important not to update the state manually:

Reactivity

React relies on its internal mechanism to track state changes and trigger re-renders. When you update the state manually, React may not be aware of the change, and the component's UI won't reflect the updated state correctly.

Predictable State Updates

Using state updater functions ensures that state updates are batched and applied in the correct order, which helps prevent race conditions and ensures that the component behaves as expected.

Component Lifecycle

State updates trigger component lifecycle methods (e.g., render, componentDidUpdate, useEffect in functional components). Manually updating state can disrupt the component's lifecycle and lead to unexpected behaviour.