Table of Contents

6. **Custom Hooks**
    - Why create custom hooks? (Reusing logic)
    - Naming convention (use...)
    - Example: Creating a useWindowSize hook
7. **Conclusion**
    - Recap of key hooks
    - Best practices and common mistakes
    - The future of React development with Hooks

Core Content

1. Introduction

What are React Hooks? React Hooks are a set of functions that let you "hook into" React state and lifecycle features from function components. They were introduced in React version 16.8 as a way to use state and other features without writing a class component.

Why were they introduced? Before Hooks, to manage state or handle lifecycle methods, you had to write a class component. This often led to code that was difficult to reuse and test. Hooks were designed to simplify components and provide a more intuitive way to manage state and side effects.

2. State Management with useState

The useState hook is a fundamental tool for adding state to your functional components. It returns an array with two elements: the current state value and a function to update it.

JavaScript

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


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


  const increment = () => {
```

```
  setCount(count + 1);
 };

 return (
  <div>
   <p>You clicked {count} times</p>
   <button onClick={increment}>Click me</button>
  </div>
 );
}
```

3. Side Effects with useEffect

The useEffect hook allows you to perform side effects in functional components. Examples of side effects include data fetching, manually changing the DOM, and setting up subscriptions.

The hook takes a function and an optional dependency array. The dependency array controls when the effect is re-run.

- **No dependency array:** The effect runs after every render.
- **Empty array []:** The effect runs only once, on the initial render. This is equivalent to componentDidMount.
- **Array with dependencies [count]:** The effect runs on the initial render and whenever the value of count changes.

JavaScript

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

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

```
 useEffect(() => {
  document.title = `You clicked ${count} times`;
 }, [count]); // This effect will re-run only when 'count' changes

 return (
  <button onClick={() => setCount(count + 1)}>
   Click to change the document title
  </button>
 );
}
```

4. Managing Context with useContext

The useContext hook allows you to subscribe to React Context without introducing a consumer wrapper. This is a cleaner way to pass data through the component tree without manually passing props at every level (a problem known as **"prop drilling"**).

JavaScript

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

// Step 1: Create the Context
const ThemeContext = React.createContext('light');

// Step 2: Provide the Context
function App() {
 return (
  <ThemeContext.Provider value="dark">
   <Toolbar />
  </ThemeContext.Provider>
 );
```

```
}


// Step 3: Consume the Context
function Toolbar() {
  const theme = useContext(ThemeContext);
  return <div style={{ background: theme === 'dark' ? '#333' : '#eee', color: theme === 'dark' ? 'white' : 'black' }}>...</div>;
}
```

5. Performance Optimization

- **useMemo:** Use this hook to cache the result of an expensive calculation. React will only re-calculate the value if one of the dependencies in the array has changed.
- **useCallback:** Use this hook to memoize a function. It's especially useful for passing functions to child components to prevent unnecessary re-renders.

6. Custom Hooks

Custom hooks are JavaScript functions whose names start with use. They allow you to extract and reuse logic from your components, promoting a clean and organized codebase.

Example: useWindowSize

JavaScript

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


function useWindowSize() {
  const [size, setSize] = useState([window.innerWidth, window.innerHeight]);


  useEffect(() => {
    const handleResize = () => {
      setSize([window.innerWidth, window.innerHeight]);
    };
```

```
  window.addEventListener('resize', handleResize);
  return () => {
    window.removeEventListener('resize', handleResize);
  };
 }, []); // Empty array ensures this effect runs only once


 return size;
}
```

7. Conclusion

Hooks have fundamentally changed how we write React components by providing a more powerful and flexible way to manage state, side effects, and logic. By mastering them, you can create cleaner, more maintainable, and highly reusable code. The key is to understand when to use each hook and to create custom hooks for logic that needs to be shared across your application.