Introduction to React Hooks

Purpose and Benefits:

- React Hooks enable the use of state and other React features in functional components, eliminating the need for class components.
- Simplifies lifecycle management and encourages functional programming paradigms.
- Enhances code reusability and readability by separating stateful logic into custom hooks.

Key Hooks:

- **useState**: Manages state in functional components.
- **useEffect**: Handles side effects in functional components.
- **useContext**: Provides access to context within functional components.
- **useRef**: Creates mutable references to DOM elements or values that persist across renders.
- **useReducer**: Manages complex state logic.
- **useMemo**: Memoizes values to optimize performance.
- **useCallback**: Memoizes callback functions to optimize performance.
- **Custom Hooks**: Encapsulate reusable logic for better code organization.

The useState Hook

Purpose:

- Adds state to functional components.

Usage:

- Initialize state by calling `useState` with an initial value. It returns an array with the current state and a function to update it.

Example:

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


function Counter() {

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

```
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</button>
    </div>
  );
}
```

Explanation:

- `count` holds the current state value.
- `setCount` is the function to update `count`.

Side Effects Using the useEffect Hook

Purpose:

- Manages side effects such as data fetching, subscriptions, and DOM manipulations.

Usage:

- `useEffect` takes a function that contains the side-effect logic. This function runs after every render by default.

Example:

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

function DataFetcher() {
  const [data, setData] = useState(null);
```

```
  useEffect(() => {
    fetch('<https://api.example.com/data>')
      .then(response => response.json())
      .then(data => setData(data));
  }, []);

  return (
    <div>
      {data ? <pre>{JSON.stringify(data, null, 2)}</pre> : 'Loading...'}
    </div>
  );
}
```

Explanation:

- The empty array `[]` as the second argument makes the effect run only once after the initial render, similar to `componentDidMount`.

The useContext Hook

Purpose:

- Provides a way to consume context values in functional components without using the `Context.Consumer` wrapper.

Usage:

- `useContext` accepts a context object and returns the current context value.

Example:

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

import { ThemeContext } from './ThemeContext';


function ThemedComponent() {

  const theme = useContext(ThemeContext);


  return (

    <div style={{ background: theme.background, color: theme.color }}>

      Themed content

    </div>

  );

}
```

Explanation:

- `ThemeContext` provides theme-related values.
- `useContext(ThemeContext)` gives access to the current theme value.

The useRef Hook

Purpose:

- Creates a mutable object that persists across renders and can reference DOM elements or store mutable values.

Usage:

- `useRef` returns a ref object with a `.current` property.

Example:

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


function FocusInput() {
  const inputRef = useRef(null);


  const handleClick = () => {
    inputRef.current.focus();
  };


  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={handleClick}>Focus the input</button>
    </div>
  );
}
```

Explanation:

- `inputRef.current` holds the reference to the input element.
- `handleClick` sets focus to the input element when the button is clicked.

The useReducer Hook

Purpose:

- Manages more complex state logic compared to `useState`.
- Useful for state that involves multiple sub-values or when the next state depends on the previous one.

Usage:

- `useReducer` accepts a reducer function and an initial state, returning the current state and a dispatch function.

Example:

```
const initialState = { count: 0 };


function reducer(state, action) {
  switch (action.type) {
   case 'increment':
    return { count: state.count + 1 };
   case 'decrement':
    return { count: state.count - 1 };
   default:
    throw new Error();
  }
}


function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
     Count: {state.count}
```

```
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </>
  );
}
```

Explanation:

- `reducer` function specifies how the state should change based on actions.
- `dispatch` function triggers state changes by dispatching actions.

The useMemo Hook

Purpose:

- Memoizes a value to optimize performance, preventing expensive calculations on every render.

Usage:

- `useMemo` takes a function and an array of dependencies. It returns a memoized value.

Example:

```
const memoizedValue = useMemo(() => expensiveCalculation(count), [count]);
```

Explanation:

- `expensiveCalculation(count)` is only recalculated when `count` changes.

The useCallback Hook

Purpose:

- Memoizes a callback function to optimize performance, preventing functions from being recreated on every render.

Usage:

- `useCallback` takes a function and an array of dependencies. It returns a memoized callback.

Example:

```
const memoizedCallback = useCallback(() => {
  doSomething(count);
}, [count]);
```

Explanation:

- `doSomething` is only recreated when `count` changes.

Writing Custom Hooks

Purpose:

- Encapsulates reusable logic in a custom hook for better code organization and reuse.

Usage:

- A custom hook is a function that can use other hooks and returns state or other values.

Example:

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

function useFetch(url) {
```

```
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch(url)
      .then(response => response.json())
      .then(data => {
        setData(data);
        setLoading(false);
      });
  }, [url]);

  return { data, loading };
}

// Usage in a component
import React from 'react';
import { useFetch } from './useFetch';

function DataDisplay() {
  const { data, loading } = useFetch('<https://api.example.com/data>');

  return (
    <div>
```

```
      {loading ? 'Loading...' : <pre>{JSON.stringify(data, null, 2)}</pre>}
    </div>
  );
}
```

Explanation:

- `useFetch` is a custom hook that fetches data from a URL and returns the data and loading state.
- The component `DataDisplay` uses `useFetch` to display the fetched data.

Additional Hooks

useLayoutEffect Hook:

- Similar to `useEffect`, but fires synchronously after all DOM mutations.
- Useful for reading layout from the DOM and synchronously re-rendering.

Syntax:

```
useLayoutEffect(() => {
  // code
}, [dependencies]);
```

useImperativeHandle Hook:

- Customizes the instance value that is exposed when using `ref` in parent components.

Syntax:

```
useImperativeHandle(ref, () => ({
```

```
  // instance value

});
```

useDebugValue Hook:

- Displays a label for custom hooks in React DevTools.

Syntax:

```
useDebugValue(value);
```

Interview Tips

Common Questions:

1. Explain the difference between `useState` and `useReducer`.
2. How does `useEffect` differ from lifecycle methods in class components?
3. When would you use `useMemo` vs `useCallback`?
4. How can you optimize performance using hooks?
5. Explain how custom hooks can be created and why they are useful.

Best Practices:

- Always list dependencies in `useEffect`, `useCallback`, and `useMemo`.
- Avoid using hooks inside loops, conditions, or nested functions.
- Use custom hooks to abstract away complex logic.
- Keep state management simple; use `useReducer` for complex state logic.

Understanding these hooks and their proper usage can significantly improve your efficiency in writing React applications and prepare you well for technical interviews.