Hooks (Functional Components):

- useState
- useEffect
- useReducer
- useRef
- useCallback
- useMemo

useState:

useState is a React Hook used in functional components to manage state. It returns an array with the current state value and a function to update it.

Example:

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

useEffect:

The useEffect Hook in functional components serves a similar purpose to lifecycle methods in class components. It allows you to perform side effects in your components, such as data fetching, subscriptions, or manually changing the DOM. Here's a comparison between useEffect in functional components and its counterparts in class components:

useEffect in Functional Components:

Syntax:

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

useEffect takes two arguments: a function containing the effect code and an optional array of dependencies. The effect runs after every render unless the dependencies change.

Mounting and Updating:

- The useEffect with an empty dependency array (useEffect(() => {}, [])) mimics componentDidMount in class components.
- The effect with dependencies is similar to componentDidUpdate and can be used for cleanup logic when dependencies change.

Unmounting:

- Cleanup logic, like unsubscribing or clearing intervals, can be included in the effect function, optionally returning a cleanup function.

Ex:

```
useEffect(() => {
  // Effect code
  return () => {
    // Cleanup code
  };
}, [dependencies]);
```

Class Component Lifecycle Methods:

componentDidMount:

- Called after the component has been inserted into the DOM.
- Equivalent to useEffect(() => {}, []).

componentDidUpdate:

- Called after the component's updates are flushed to the DOM.
- Equivalent to useEffect(() => {}, [dependencies]).

componentWillUnmount:

- Called immediately before a component is unmounted and destroyed.
- Equivalent to cleanup logic in useEffect with a cleanup function.

Comparison Summary:

- useEffect simplifies the organization of side effects in functional components compared to the scattered nature of lifecycle methods in class components.
- It provides a clear separation of concerns by combining mount, update, and unmount logic in a single function.
- The dependency array helps control when the effect runs and ensures proper cleanup.
- Class components require multiple lifecycle methods for similar behavior, leading to potentially more verbose code.

useReducer:

useReducer is a Hook that manages state in functional components with a more complex logic using a reducer function. It returns the current state and a dispatch function.

Example:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

useRef:

useRef is a Hook used for accessing and interacting with the DOM or for persisting values across renders without causing re-renders.

Example:

```
const inputRef = useRef(null);
```

useCallback:

useCallback is a Hook that memoizes a callback function to prevent unnecessary re-renders. It takes a callback function and a dependency array.

Example:

```
const handleClick = useCallback(() => {
  setCount(count + 1);
}, [count]);
```

useMemo:

useMemo is a Hook that memoizes a value to prevent unnecessary recalculations. It takes a function for the value calculation and a dependency array.

Example:

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

These hooks are crucial for managing state, effects, and performance optimizations in functional components within a React application.