Hooks

React Hooks are special functions introduced in React 16.8 that allow you to use state and lifecycle features inside **functional components**. Before hooks, state and lifecycle methods were only available in **class components**, and functional components were stateless and simpler. Hooks bring the benefits of managing state and side effects into functional components, making React code more reusable, modular, and readable.

Why Hooks?

1. **Reusability**: Hooks allow the reuse of logic between components without complex patterns like higher-order components (HOCs) or render props.
2. **Simplification**: Class components require you to manage this and lifecycle methods (componentDidMount, componentDidUpdate, componentWillUnmount). Hooks simplify this by combining these lifecycle methods.
3. **Function Components**: Hooks enable functional components to manage state and side effects, making function components more powerful and preferred in modern React apps.

Types of Hooks

React provides two types of hooks:

1. **Basic Hooks**: Used most frequently in React applications.
2. **Additional Hooks**: Used for more advanced use cases or optimizations.

---

1. Basic Hooks

1.1 useState: State Management

The useState hook lets you add local state to functional components.

Syntax:

javascript

Copy code

```
const [state, setState] = useState(initialState);
```

Example:

jsx

Copy code

```
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>
  );
}
```

- useState returns two values:
    1. count: The current state.
    2. setCount: A function to update the state.

1.2 useEffect: Side Effects

The useEffect hook handles side effects like fetching data, directly manipulating the DOM, or setting timers. It replaces lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount.

Syntax:

javascript

Copy code

```
useEffect(() => {
  // Code to execute (effect)

  return () => {
    // Cleanup code (optional)
  };
}, [dependencies]);  // Optional array of dependencies
```

Example:

jsx

Copy code

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

function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds(seconds + 1);
```

```
  }, 1000);

  // Cleanup when the component is unmounted
  return () => clearInterval(interval);
 }, [seconds]); // Runs whenever 'seconds' changes

 return <p>Timer: {seconds}s</p>;
}
```

- If the dependency array [] is empty, the effect only runs once (on mount).
- If dependencies are provided, the effect runs whenever any dependency changes.

1.3 useContext: Context API

The useContext hook allows you to access React context values directly without needing to wrap components with Context.Consumer.

Syntax:

javascript

Copy code

```
const value = useContext(SomeContext);
```

Example:

jsx

Copy code

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

const UserContext = createContext();

function Greeting() {
 const user = useContext(UserContext);
 return <h1>Hello, {user.name}!</h1>;
}

function App() {
 const user = { name: 'Alice' };
 return (
```

```
  <UserContext.Provider value={user}>
    <Greeting />
  </UserContext.Provider>
 );
}
```

1.4 useReducer: Complex State Management

The useReducer hook is an alternative to useState and is used when the state logic is more complex, such as multiple sub-values or state transitions. It's particularly useful for managing state that depends on previous state values.

Syntax:

javascript

Copy code

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

Example:

jsx

Copy code

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

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

function Counter() {
 const [state, dispatch] = useReducer(reducer, { count: 0 });

 return (
```

```
  <div>
    <p>Count: {state.count}</p>
    <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    <button onClick={() => dispatch({ type: 'increment' })}>+</button>
  </div>
 );
}
```

- useReducer accepts a reducer function (that handles state transitions) and an initial state.
- It returns the current state and a dispatch function to trigger state transitions.

1.5 useRef: Accessing DOM Elements or Persisting Values

The useRef hook is used for accessing DOM elements and storing mutable values that do not cause re-renders when changed.

Syntax:

javascript

Copy code

```
const ref = useRef(initialValue);
```

Example:

jsx

Copy code

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

function TextInputWithFocusButton() {
 const inputEl = useRef(null);

 const onButtonClick = () => {
  inputEl.current.focus();
 };

 return (
  <div>
   <input ref={inputEl} type="text" />
   <button onClick={onButtonClick}>Focus the input</button>
```

```
  </div>
 );
}
```

1.6 useMemo: Memoization of Expensive Calculations

The useMemo hook memoizes expensive calculations and returns a memoized value only when its dependencies change. It helps optimize performance.

Syntax:

javascript

Copy code

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

Example:

jsx

Copy code

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

function ExpensiveCalculation({ number }) {
 const expensiveComputation = useMemo(() => {
  console.log('Computing...');
  return number * 2;
 }, [number]);

 return <p>Result: {expensiveComputation}</p>;
}
```

1.7 useCallback: Memoization of Functions

The useCallback hook memoizes a function reference and returns the same function instance unless its dependencies change. It's useful for passing callbacks to child components without causing unnecessary re-renders.

Syntax:

javascript

Copy code

```
const memoizedCallback = useCallback(() => doSomething(a, b), [a, b]);
```

Example:

jsx

Copy code

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


function Button({ onClick }) {
 return <button onClick={onClick}>Click me</button>;
}


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


 const increment = useCallback(() => {
  setCount((prevCount) => prevCount + 1);
 }, []);


 return (
  <div>
   <p>Count: {count}</p>
   <Button onClick={increment} />
  </div>
 );
}
```

---

2. Additional Hooks

2.1 useLayoutEffect

Similar to useEffect, but it fires synchronously after all DOM mutations. It is used when you need to interact with the DOM before the browser repaints.

2.2 useImperativeHandle

This hook customizes the instance value that is exposed to parent components when using ref. It's useful for exposing a controlled subset of functionality of a component to its parent.

Conclusion

React hooks are powerful and provide flexible ways to handle state, side effects, and more, all while keeping functional components clean and easy to understand. Hooks like useState, useEffect, and useContext are

widely used, while more advanced hooks like useMemo, useCallback, and useReducer help optimize and manage complex logic.

Here are some common interview questions related to React Hooks, along with brief explanations or answers that could help you prepare:

1. What are React Hooks?

Answer: React Hooks are functions that let you use state and other React features in functional components. They were introduced in React 16.8 to allow functional components to manage state, lifecycle methods, and other features without using class components.

2. Can you name some commonly used hooks in React?

Answer: Some commonly used hooks include:

- useState: For managing local state.
- useEffect: For performing side effects, like data fetching and subscriptions.
- useContext: For accessing context values without using Context.Consumer.
- useReducer: For managing complex state logic with a reducer function.
- useRef: For accessing DOM elements or persisting mutable values.
- useMemo: For memoizing expensive computations.
- useCallback: For memoizing functions to avoid unnecessary re-renders.

3. How does useState work, and what does it return?

Answer: The useState hook takes an initial state value and returns an array containing two elements: the current state value and a function to update that state. The state updates trigger a re-render of the component.

javascript

Copy code

const [state, setState] = useState(initialState);

4. Explain the useEffect hook and its dependencies.

Answer: The useEffect hook lets you perform side effects in functional components. It runs after every render by default but can be controlled with a dependency array. If dependencies are provided, the effect only runs when one of those dependencies changes. If an empty array is passed, it runs only once after the initial render (componentDidMount).

5. What is the difference between useEffect and useLayoutEffect?

Answer:

- useEffect runs asynchronously after the render is painted to the screen, which can be used for side effects that don't require immediate reflection in the DOM.

- useLayoutEffect runs synchronously after all DOM mutations but before the browser paints. This is useful for reading layout from the DOM and synchronously re-rendering before the browser paints.

6. How can you prevent a function from being recreated on every render using hooks?

Answer: You can use the useCallback hook to memoize a function. By providing a dependency array, useCallback returns the same function instance until one of the dependencies changes.

javascript

Copy code

```
const memoizedCallback = useCallback(() => {
  // Function logic
}, [dependency1, dependency2]);
```

7. What is the purpose of useMemo, and how is it different from useCallback?

Answer:

- useMemo is used to memoize a computed value, returning the memoized result of a function only when the dependencies change. It's helpful for optimizing performance when you have expensive calculations.

javascript

Copy code

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

- useCallback is specifically for memoizing function references, while useMemo is for memoizing values.

8. How does the useContext hook work, and why would you use it?

Answer: The useContext hook allows you to access the value of a context directly in a functional component without needing to wrap components with Context.Consumer. It simplifies accessing context values, making the code cleaner and more readable.

9. Explain the use case of the useReducer hook.

Answer: useReducer is ideal for managing complex state logic in functional components. It is especially useful when state updates depend on previous state values or when managing multiple sub-values. It takes a reducer function and an initial state, similar to how Redux works.

10. What is the cleanup function in useEffect, and when is it called?

Answer: The cleanup function in useEffect is returned from the effect callback function and is executed when the component unmounts or before the effect runs again. It's used to clean up subscriptions, timers, or other resources to prevent memory leaks.

11. How do you test components that use hooks?

Answer: You can use libraries like **React Testing Library** or **Enzyme** for testing components with hooks. They provide utilities to render components, trigger events, and assert state changes effectively.

12. Can you use hooks in class components?

Answer: No, hooks can only be used in functional components. Class components can only use lifecycle methods and this for managing state and side effects.

Hooks

13. What are some rules of hooks?

Answer:

1. **Only Call Hooks at the Top Level**: Don't call hooks inside loops, conditions, or nested functions to ensure consistent execution order.
2. **Only Call Hooks from React Functions**: Call hooks from functional components or custom hooks, not from regular JavaScript functions.

14. What are custom hooks?

Answer: Custom hooks are JavaScript functions that use built-in hooks to encapsulate and reuse stateful logic. They follow the naming convention of starting with use and can be shared across multiple components.

15. How can you share logic between components using hooks?

Answer: You can create custom hooks that encapsulate shared logic. This allows you to use the same logic in multiple components without repeating code, enhancing reusability and maintainability.

These questions should give you a comprehensive understanding of React Hooks and help you prepare effectively for an interview. Good luck!