MODULE: 10 List and Hooks

Class Component Lifecycle

- Class components have several lifecycle methods that can be overridden to run code at specific times during a component's life. Here are the main lifecycle methods:

1. **Mounting**
    - **constructor()**: Called when the component is initialized. It is used for setting up initial state and binding event handlers.
    - **componentDidMount()**: Invoked immediately after the component is mounted (inserted into the tree). It is a good place to initiate network requests or set up subscriptions.
2. **Updating**
    - **shouldComponentUpdate(nextProps, nextState)**: Called to determine if a re-render is necessary. It should return a boolean value. By default, it returns `true`.
    - **componentDidUpdate(prevProps, prevState, snapshot)**: Invoked immediately after updating. It's a good place to operate on the DOM when the component has been updated.
    - **componentWillReceiveProps(nextProps)**: This method is called when the component receives new props and before rendering. This method is now considered UNSAFE for use and may be removed in future versions.
3. **Unmounting**
    - **componentWillUnmount()**: Invoked immediately before the component is unmounted and destroyed. Use it to clean up subscriptions, timers, or any other resource to prevent memory leaks.
4. **Error Handling**
    - **componentDidCatch(error, info)**: Called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

Example:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  componentDidMount() {
    console.log('Component did mount');
  }
```

```
  componentDidUpdate(prevProps, prevState) {
    if (prevState.count !== this.state.count) {
      console.log('Component did update');
    }
  }

  componentWillUnmount() {
    console.log('Component will unmount');
  }

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Increment
        </button>
      </div>
    );
  }
}
```

Functional Component Lifecycle with Hooks

Functional components don't have lifecycle methods but use hooks to achieve the same functionality. The most important hook related to lifecycle events is `useEffect`.

1. **Mounting**
    - **useEffect(callback, [ ])**: The `useEffect` hook with an empty dependency array [ ] acts like `componentDidMount`. It runs only once after the initial render.
2. **Updating**
    - **useEffect(callback, [dependencies])**: The `useEffect` hook with a dependency array runs after every render when one of the dependencies has changed. This is equivalent to `componentDidUpdate`.
3. **Unmounting**
    - **useEffect(callback, [ ])** with a cleanup function: Returning a function from the `useEffect` hook will be invoked when the component is unmounted, similar to `componentWillUnmount`.

Example:

```
import React, { useState, useEffect } from 'react';
function MyComponent() {
const [count, setCount] = useState(0);
useEffect(() => {
console.log('Component did mount');
 return () => {
 console.log('Component will unmount');
  };
  }, [ ]);
  useEffect(() => {
    console.log('Component did update');
  }, [count]);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```