Advanced React State Management Concepts

1. State Lifting

Lift state up to a common parent when multiple components need access to the same data.

```
const Parent = () => {
  const [text, setText] = useState('');
  return (
    <>
      <InputComponent text={text} setText={setText} />
      <DisplayComponent text={text} />
    </>
  );
};

const InputComponent = ({ text, setText }) => (
  <input value={text} onChange={(e) => setText(e.target.value)} />
);

const DisplayComponent = ({ text }) => <p>{text}</p>;
```

2. Managing Complex State with useReducer

Use useReducer for complex state logic involving multiple actions or nested objects.

```
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: return state;
  }
}

const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </>
  );
};
```

3. Derived State

Compute state based on props or other state values instead of storing derived data directly.

```
const PriceDisplay = ({ price, discount }) => {
  const finalPrice = price - discount;
  return <p>Final Price: ${finalPrice}</p>;
```

```
};
```

4. Persisting State with localStorage

Use localStorage to persist state across page reloads.

```
useEffect(() => {
  const saved = localStorage.getItem('name');
  if (saved) setName(saved);
}, []);

useEffect(() => {
  localStorage.setItem('name', name);
}, [name]);
```

5. Cleaning Up with useEffect

Clean up side effects like timers or subscriptions to avoid memory leaks.

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log('Tick');
  }, 1000);

  return () => clearInterval(timer); // Cleanup
}, []);
```

6. Custom Hooks for Reusable State Logic

Encapsulate state logic into reusable custom hooks.

```
const useCounter = (initial = 0) => {
  const [count, setCount] = useState(initial);
  const increment = () => setCount((c) => c + 1);
  return { count, increment };
};

const Counter = () => {
  const { count, increment } = useCounter();
  return <button onClick={increment}>Count: {count}</button>;
};
```