React: Lists, Keys, and Hooks

Lists and Keys

Q1: How can we render a list in React, and why are keys essential?

Rendering Lists:
To display a list in React, use the `.map()` method to loop over an array and return JSX:

js
CopyEdit

```
const fruits = ['apple', 'banana', 'orange'];
const FruitList = () => {
  return (
    <ul>
      {fruits.map((fruit, index) => (
        <li key={index}>{fruit}</li>
      ))}
    </ul>
  );
};
```

Why Keys Matter:

- 🔄 **Improves Rendering Performance:** Keys help React update only changed elements, avoiding unnecessary re-renders.
- ⚙️ **Better Virtual DOM Matching:** React uses keys to track elements across renders, optimizing updates.
- 🧠 **State Preservation:** Helps maintain component state (like input values) during reordering.
- 🚫 **Avoids UI Bugs:** Without proper keys, React may mismatch elements, causing rendering glitches.

Q2: What are keys in React? What happens if keys aren't unique?

What Are Keys?

- Special string identifiers that uniquely tag each element in a list.
- They must be **consistent, unique**, and **stable**.

Good Practice:

js
CopyEdit

```
users.map(user => <User key={user.id} user={user} />)
```

Avoid This (for dynamic lists):

js
CopyEdit

```
users.map((user, index) => <User key={index} user={user} />)
```

Without Unique Keys:

- ⚠️ React re-renders inefficiently
- 🧩 State can get mixed up or lost
- 🔁 UI inconsistencies after reordering
- 🛠️ Development warnings in console
- ✍️ Form inputs may retain incorrect data

React Hooks

Q1: What are React Hooks? How do useState and useEffect work?

Hooks:
Functions that let functional components use features like state and lifecycle events without needing class components.

`useState` Example:

js
CopyEdit

```
import { useState } from 'react';
const Counter = () => {
  const [count, setCount] = useState(0);

  return (
    <>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
      <button onClick={() => setCount(prev => prev - 1)}>-</button>
    </>
  );
};
```

- Returns `[currentValue, setter]`
- Setter can update using a new value or a callback with the previous state

`useEffect` Example:

js
CopyEdit

```
import { useEffect, useState } from 'react';
const UserProfile = ({ userId }) => {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetchUser(userId).then(setUser);
  }, [userId]); // Runs when userId changes

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

```
    return () => clearInterval(timer); // Clean-up
  }, []);

  return <div>{user?.name}</div>;
};
```

Q2: What problems do Hooks solve? Why are they useful?

Hooks solve:

1. 🔁 Complex class syntax & binding issues
2. ♻️ Difficult code reuse across components
3. 🔄 Confusing lifecycle methods
4. 🧩 Excessive HOCs and render props
5. 📦 Larger bundles due to boilerplate code

Hooks bring:

- Simpler, functional coding style
- Grouped logic by behavior, not lifecycle
- Easier testing of isolated logic
- Potential for performance gains
- Base for upcoming concurrent features

Q3: What is `useReducer` and when should you use it?

`useReducer`:
An alternative to `useState` for managing complex state logic. Similar to Redux but built-in.

When to Use:

- State depends on previous values
- Multiple actions or nested state updates
- Logic gets complicated

Example:

js
CopyEdit

```
const reducer = (state, action) => {
  switch (action.type) {
    case 'INCREMENT': return { count: state.count + 1 };
    case 'DECREMENT': return { count: state.count - 1 };
    case 'RESET': return { count: 0 };
    default: throw new Error('Invalid action');
  }
};
const Counter = () => {
  const [state, dispatch] = useReducer(reducer, { count: 0 });
  return (
    <>
      <p>{state.count}</p>
      <button onClick={() => dispatch({ type: 'INCREMENT'
})}>+</button>
      <button onClick={() => dispatch({ type: 'DECREMENT'
})}>-</button>
      <button onClick={() => dispatch({ type: 'RESET'
})}>Reset</button>
    </>
  );
};
```

Form Example:

js
CopyEdit

```
const formReducer = (state, action) => {
  switch (action.type) {
```

```
    case 'SET_FIELD':
      return { ...state, [action.field]: action.value };
    case 'SET_ERROR':
      return { ...state, errors: { ...state.errors, [action.field]:
action.error } };
    case 'RESET':
      return { name: '', email: '', errors: {} };
    default:
      return state;
  }
};
```

Q4: What do `useCallback` and `useMemo` do in React?

`useCallback`:

- Caches function references to avoid unnecessary re-renders.
- Only recreates function when dependencies change.

`useMemo`:

- Caches return values of expensive computations.
- Only recalculates if dependencies change.

Why Use Them?

- Boosts performance in large apps
- Prevents re-renders of memoized child components
- Reduces function/object memory usage

Q5: Difference Between `useCallback` vs `useMemo`

Feature	`useCallback`	`useMemo`
Purpose	Cache functions	Cache values/computations
Returns	Memoized function	Memoized value
Syntax	`useCallback(fn, deps)`	`useMemo(() => fn(), deps)`
Use Case	Stable functions as props	Expensive calculations optimization

Example - `useCallback`:

js
CopyEdit

```
const memoizedClick = useCallback(() => {
  console.log("Clicked");
}, []);
```

Example - `useMemo`:

js
CopyEdit

```
const filteredItems = useMemo(() => {
  return items.filter(item => item.includes(filter));
}, [items, filter]);
```

Q6: What is `useRef` and how does it work?

`useRef`:
Returns a mutable object that persists between renders. It doesn't trigger a re-render when updated.

Use Cases:

1. **Access DOM elements:**

js
CopyEdit

```
const inputRef = useRef();
<input ref={inputRef} />
```

2. **Track mutable values:**

js
CopyEdit

```
const timerRef = useRef();
timerRef.current = setInterval(...);
```

3. **Store previous values:**

js
CopyEdit

```
const usePrevious = (value) => {
  const ref = useRef();
  useEffect(() => { ref.current = value; });
  return ref.current;
};
```

Key Points:

- Doesn’t cause re-renders
- Useful for accessing DOM or persisting values
- Works like an instance variable in a functional component