React Interview Questions & Answers

For 1.5 Years Experience Level

Core React Concepts

1. What is React and why do we use it?

Answer: React is a JavaScript library for building user interfaces, developed by Facebook. We use it because:

- **Component-Based Architecture**: Build encapsulated components that manage their own state
- **Virtual DOM**: React uses a virtual DOM to efficiently update only what changes
- **Declarative**: You describe what the UI should look like, React handles the updates
- **Reusable Components**: Write once, use anywhere
- **Large Ecosystem**: Huge community and third-party libraries

Example:

```
function Welcome() {
  return <h1>Hello, World!</h1>;
}
```

---

2. What is JSX?

Answer: JSX (JavaScript XML) is a syntax extension for JavaScript that looks like HTML. It makes writing React components more intuitive.

Key Points:

- JSX gets compiled to `React.createElement()` calls
- You can embed JavaScript expressions using `{}`
- Must return a single parent element
- Use `className` instead of `class`, `htmlFor` instead of `for`

Example:

```
const name = "John";
const element = <h1>Hello, {name}!</h1>;
```

```
// This JSX compiles to:
const element = React.createElement('h1', null, 'Hello, ', name, '!');
```

---

3. What is the difference between functional and class components?

Answer:

Functional Components:

- Simpler syntax
- Use Hooks for state and lifecycle
- Better performance (slightly)
- Preferred modern approach

Class Components:

- Use `this.state` and `this.setState()`
- Have lifecycle methods
- More verbose
- Legacy approach

Example:

```
// Functional Component
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

// Class Component
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

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

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

---

4. What are Props?

Answer: Props (properties) are read-only data passed from parent to child components. They allow components to be dynamic and reusable.

Key Points:

- Props are immutable (cannot be changed by child)
- Passed as attributes
- Accessed via `props.propertyName`
- Can pass any data type including functions

Example:

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}

function App() {
  return <Greeting name="Alice" />;
}
```

---

5. What is State?

Answer: State is a built-in object that stores component data that can change over time. When state changes, the component re-renders.

Key Points:

- State is mutable (can be changed)
- State is local to the component
- Use `useState` hook in functional components
- Never mutate state directly, always use setState

Example:

```
function TodoApp() {
  const [todos, setTodos] = useState([]);
```

```
  const [input, setInput] = useState('');

  const addTodo = () => {
    setTodos([...todos, input]); // Create new array
    setInput('');
  };

  return (
    <div>
      <input value={input} onChange={(e) => setInput(e.target.value)} />
      <button onClick={addTodo}>Add</button>
      <ul>
        {todos.map((todo, index) => <li key={index}>{todo}</li>)}
      </ul>
    </div>
  );
}
```

---

6. What is the difference between State and Props?

Answer:

Feature	Props	State
Mutability	Immutable	Mutable
Source	Parent component	Component itself
Change triggers	Re-render when parent changes	Re-render when setState called
Access	Read-only	Read and write

Example:

```
function Child(props) {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Parent says: {props.message}</p> {/* Props from parent */}
      <p>My count: {count}</p> {/* Own state */}
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

```
function Parent() {
  return <Child message="Hello from parent!" />;
}
```

React Hooks

7. What is useState hook?

Answer: `useState` is a Hook that lets you add state to functional components. It returns an array with the current state value and a function to update it.

Syntax:

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

Example:

```
function Counter() {
  const [count, setCount] = useState(0);
  const [name, setName] = useState('Guest');

  return (
    <div>
      <h2>Hello, {name}</h2>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>+</button>
      <button onClick={() => setCount(count - 1)}>-</button>
      <input
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
    </div>
  );
}
```

---

8. What is useEffect hook?

Answer: `useEffect` performs side effects in functional components. It runs after render and can optionally clean up.

Key Points:

- Runs after every render by default

- Can specify dependencies to control when it runs
- Can return a cleanup function
- Replaces lifecycle methods like `componentDidMount`, `componentDidUpdate`, `componentWillUnmount`

Example:

```
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    // Runs after render
    fetch(`/api/users/${userId}`)
      .then(res => res.json())
      .then(data => setUser(data));

    // Cleanup function (optional)
    return () => {
      console.log('Cleanup');
    };
  }, [userId]); // Only re-run if userId changes

  if (!user) return <div>Loading...</div>;

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

Dependency Array Variations:

```
useEffect(() => {}); // Runs after every render
useEffect(() => {}, []); // Runs once on mount
useEffect(() => {}, [dep1, dep2]); // Runs when deps change
```

---

9. What is useContext hook?

Answer: `useContext` lets you access context values without wrapping components in Context.Consumer.

Example:

```
// Create context
const ThemeContext = React.createContext('light');

function App() {
  return (
```

```
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar() {
  return <ThemedButton />;
}

function ThemedButton() {
  const theme = useContext(ThemeContext); // Access context
  return <button className={theme}>I'm {theme} themed!</button>;
}
```

---

10. What is useRef hook?

Answer: `useRef` creates a mutable reference that persists across renders. Common uses: accessing DOM elements and storing mutable values without causing re-renders.

Key Points:

- Returns an object with a `.current` property
- Changing `.current` doesn't trigger re-render
- Useful for DOM manipulation

Example:

```
function TextInput() {
  const inputRef = useRef(null);
  const countRef = useRef(0);

  const focusInput = () => {
    inputRef.current.focus(); // Access DOM element
  };

  const handleClick = () => {
    countRef.current += 1; // Update without re-render
    console.log('Clicked:', countRef.current);
  };

  return (
    <div>
      <input ref={inputRef} />
      <button onClick={focusInput}>Focus Input</button>
      <button onClick={handleClick}>Click Me</button>
```

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

---

11. What is useMemo hook?

Answer: `useMemo` memoizes expensive calculations, only recalculating when dependencies change. It optimizes performance.

Example:

```
function ExpensiveComponent({ items, filter }) {
  const filteredItems = useMemo(() => {
    console.log('Filtering...');
    return items.filter(item => item.includes(filter));
  }, [items, filter]); // Only recalculate if items or filter changes

  return (
    <ul>
      {filteredItems.map(item => <li key={item}>{item}</li>)}
    </ul>
  );
}
```

---

12. What is useCallback hook?

Answer: `useCallback` memoizes functions, returning the same function reference unless dependencies change. Useful for optimizing child components that rely on reference equality.

Example:

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

  // Without useCallback, this creates a new function on every render
  const increment = useCallback(() => {
    setCount(c => c + 1);
  }, []); // Function never changes

  return (
    <div>
      <p>Count: {count}</p>
```

```
      <Child onIncrement={increment} />
      <button onClick={() => setOther(other + 1)}>Other: {other}</button>
    </div>
  );
}

const Child = React.memo(({ onIncrement }) => {
  console.log('Child rendered');
  return <button onClick={onIncrement}>Increment</button>;
});
```

Component Lifecycle

13. What are React lifecycle methods?

Answer: Lifecycle methods are hooks that allow you to run code at specific times in a component's life.

Class Component Lifecycle:

Mounting:

1. `constructor()` - Initialize state
2. `render()` - Return JSX
3. `componentDidMount()` - After first render (API calls, subscriptions)

Updating:

1. `shouldComponentUpdate()` - Optimize rendering
2. `render()` - Re-render
3. `componentDidUpdate()` - After update

Unmounting:

1. `componentWillUnmount()` - Cleanup (clear timers, cancel subscriptions)

Functional Component with Hooks:

```
function MyComponent() {
  useEffect(() => {
    // componentDidMount
    console.log('Mounted');

    return () => {
      // componentWillUnmount
```

```
    console.log('Unmounted');
  };
 }, []);

 useEffect(() => {
  // componentDidUpdate (for specific dependencies)
  console.log('Updated');
 }, [dependency]);

 return <div>Component</div>;
}
```

---

14. What is the difference between componentDidMount and useEffect?

Answer:

componentDidMount:

- Class component method
- Runs once after initial render
- Cannot be conditional

useEffect:

- Functional component hook
- Can run on mount, update, or both (depending on dependencies)
- Can be conditional
- More flexible

Example:

```
// Class
componentDidMount() {
 fetchData();
}

// Functional (equivalent)
useEffect(() => {
 fetchData();
}, []); // Empty array = run once on mount
```

Event Handling

15. How do you handle events in React?

Answer: React uses camelCase event handlers passed as props. Events are wrapped in React's SyntheticEvent system.

Key Points:

- Use camelCase: `onClick`, not `onclick`
- Pass function reference, not call: `onClick={handleClick}`, not `onClick={handleClick()}`
- Prevent default with `e.preventDefault()`

Example:

```
function Form() {
  const [name, setName] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault(); // Prevent page reload
    console.log('Submitted:', name);
  };

  const handleChange = (e) => {
    setName(e.target.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        value={name}
        onChange={handleChange}
        onFocus={() => console.log('Focused')}
        onBlur={() => console.log('Blurred')}
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

---

16. What is event delegation in React?

Answer: React uses event delegation by attaching a single event listener at the root level and delegating events to the appropriate handlers. This is handled automatically by React.

Example:

```
function List() {
  const handleClick = (id) => {
```

```
    console.log('Clicked item:', id);
  };

  return (
    <ul>
      {[1, 2, 3].map(id => (
        <li key={id} onClick={() => handleClick(id)}>
          Item {id}
        </li>
      ))}
    </ul>
  );
}
```

Conditional Rendering

17. How do you conditionally render components?

Answer: There are several ways to conditionally render in React:

1. if/else:

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
  } else {
    return <h1>Please sign in.</h1>;
  }
}
```

2. Ternary operator:

```
function Greeting({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please sign in.</h1>}
    </div>
  );
}
```

3. Logical && operator:

```
function Mailbox({ unreadMessages }) {
  return (
```

```
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 && (
        <h2>You have {unreadMessages.length} unread messages.</h2>
      )}
    </div>
  );
}
```

4. Switch statement:

```
function Status({ status }) {
  switch (status) {
    case 'loading':
      return <Spinner />;
    case 'error':
      return <Error />;
    case 'success':
      return <Data />;
    default:
      return null;
  }
}
```

Lists and Keys

18. Why do we need keys in lists?

Answer: Keys help React identify which items have changed, been added, or removed. They should be stable, unique identifiers.

Key Points:

- Keys must be unique among siblings
- Don't use array index as key if list can reorder
- Keys help React optimize rendering

Example:

```
function TodoList({ todos }) {
  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id}> {/* Use unique ID, not index */}
          {todo.text}
```

```
      </li>
    ))}
  </ul>
 );
}

// BAD - Using index
{todos.map((todo, index) => (
 <li key={index}>{todo.text}</li> // Avoid this
))}
```

Forms

19. What are controlled components?

Answer: Controlled components are form elements whose values are controlled by React state. The state is the "single source of truth."

Example:

```
function LoginForm() {
 const [formData, setFormData] = useState({
  email: '',
  password: ''
 });

 const handleChange = (e) => {
  const { name, value } = e.target;
  setFormData(prev => ({
   ...prev,
   [name]: value
  }));
 };

 const handleSubmit = (e) => {
  e.preventDefault();
  console.log('Form data:', formData);
 };

 return (
  <form onSubmit={handleSubmit}>
   <input
    name="email"
    value={formData.email}
    onChange={handleChange}
```

```
        placeholder="Email"
      />
      <input
        name="password"
        type="password"
        value={formData.password}
        onChange={handleChange}
        placeholder="Password"
      />
      <button type="submit">Login</button>
    </form>
  );
}
```

---

20. What are uncontrolled components?

Answer: Uncontrolled components store their own state in the DOM. You use refs to access their values.

Example:

```
function UncontrolledForm() {
  const emailRef = useRef();
  const passwordRef = useRef();

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Email:', emailRef.current.value);
    console.log('Password:', passwordRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input ref={emailRef} placeholder="Email" />
      <input ref={passwordRef} type="password" placeholder="Password" />
      <button type="submit">Login</button>
    </form>
  );
}
```

Performance Optimization

21. What is React.memo?

Answer: `React.memo` is a higher-order component that memoizes functional components. It prevents re-renders if props haven't changed.

Example:

```
const ExpensiveComponent = React.memo(({ data }) => {
  console.log('Rendering ExpensiveComponent');
  return <div>{data}</div>;
});

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

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>Count: {count}</button>
      <ExpensiveComponent data={data} />
      {/* ExpensiveComponent won't re-render when count changes */}
    </div>
  );
}
```

---

22. How do you optimize React app performance?

Answer:

1. Use React.memo for components:

```
const MemoizedChild = React.memo(Child);
```

2. Use useMemo for expensive calculations:

```
const sortedList = useMemo(() => items.sort(), [items]);
```

3. Use useCallback for functions:

```
const handleClick = useCallback(() => {}, []);
```

4. Code splitting with lazy loading:

```
const LazyComponent = React.lazy(() => import('./Component'));

function App() {
```

```
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <LazyComponent />
    </Suspense>
  );
}
```

5. Virtualize long lists: Use libraries like react-window or react-virtualized.

6. Avoid inline functions and objects:

```
// BAD
<Child onClick={() => doSomething()} />

// GOOD
const handleClick = useCallback(() => doSomething(), []);
<Child onClick={handleClick} />
```

Context API

23. What is Context API?

Answer: Context API allows you to share data across the component tree without passing props through every level (prop drilling).

Example:

```
// Create Context
const UserContext = React.createContext();

// Provider Component
function App() {
  const [user, setUser] = useState({ name: 'John', role: 'admin' });

  return (
    <UserContext.Provider value={{ user, setUser }}>
      <Header />
      <Main />
    </UserContext.Provider>
  );
}

// Consumer Component
function Profile() {
  const { user, setUser } = useContext(UserContext);
```

```
  return (
    <div>
      <h2>{user.name}</h2>
      <p>Role: {user.role}</p>
      <button onClick={() => setUser({ ...user, name: 'Jane' })}>
        Change Name
      </button>
    </div>
  );
}
```

---

24. When should you use Context vs Props?

Answer:

Use Props when:

- Data is needed by only a few components
- Clear parent-child relationships
- Component reusability is important

Use Context when:

- Data is needed by many components at different nesting levels
- Global or app-wide state (theme, auth, language)
- Avoiding prop drilling

Example:

```
// Props - Simple hierarchy
<Parent>
  <Child data={data} />
</Parent>

// Context - Deep hierarchy or global state
<ThemeContext.Provider value={theme}>
  <App>
    <Header /> {/* Can access theme */}
    <Main>
      <Sidebar>
        <Button /> {/* Can access theme without prop drilling */}
      </Sidebar>
    </Main>
  </App>
</ThemeContext.Provider>
```

React Router

25. What is React Router?

Answer: React Router is a library for handling routing in React applications. It enables navigation between different views/pages.

Example:

```
import { BrowserRouter, Routes, Route, Link, useNavigate } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      <nav>
        <Link to="/">Home</Link>
        <Link to="/about">About</Link>
        <Link to="/users/123">User Profile</Link>
      </nav>

      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/users/:id" element={<UserProfile />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </BrowserRouter>
  );
}

function UserProfile() {
  const { id } = useParams(); // Get URL parameter
  const navigate = useNavigate(); // Programmatic navigation

  return (
    <div>
      <h2>User ID: {id}</h2>
      <button onClick={() => navigate('/')}>Go Home</button>
    </div>
  );
}
```

Advanced Concepts

26. What is prop drilling and how do you avoid it?

Answer: Prop drilling is passing props through multiple levels of components that don't need them, just to reach a deeply nested component.

Solutions:

1. Context API
2. State management libraries (Redux, Zustand)
3. Component composition

Example:

```
// Prop Drilling Problem
function App() {
  const [user, setUser] = useState({ name: 'John' });
  return <Parent user={user} />;
}

function Parent({ user }) {
  return <Child user={user} />; // Just passing through
}

function Child({ user }) {
  return <GrandChild user={user} />; // Just passing through
}

function GrandChild({ user }) {
  return <h1>{user.name}</h1>; // Finally used
}

// Solution with Context
const UserContext = React.createContext();

function App() {
  const [user, setUser] = useState({ name: 'John' });
  return (
    <UserContext.Provider value={user}>
      <Parent />
    </UserContext.Provider>
  );
}

function GrandChild() {
  const user = useContext(UserContext); // Direct access
  return <h1>{user.name}</h1>;
```

}

---

27. What is the Virtual DOM?

Answer: The Virtual DOM is a lightweight copy of the actual DOM. React uses it to optimize updates by:

1. Creating a virtual representation of UI
2. Comparing it with previous version (diffing)
3. Calculating minimum changes needed
4. Updating only changed parts in real DOM (reconciliation)

Why it's faster:

- DOM manipulation is slow
- Virtual DOM operations are in-memory (fast)
- Batch updates reduce reflows/repaints

Example Flow:

State changes → Virtual DOM updated → Diffing → Real DOM patched

---

28. What is reconciliation?

Answer: Reconciliation is the process React uses to update the DOM efficiently by comparing the new Virtual DOM with the previous one.

How it works:

1. When state/props change, React creates a new Virtual DOM tree
2. React compares (diffs) new tree with old tree
3. React calculates minimum changes
4. React updates only changed nodes in real DOM

Optimization techniques:

- Keys in lists help identify elements
- Component type changes cause full remount
- Same type components update props

---

29. What are Higher-Order Components (HOC)?

Answer: HOC is a pattern where a function takes a component and returns a new enhanced component. Used for reusing component logic.

Example:

```
// HOC that adds loading functionality
function withLoading(Component) {
  return function WithLoadingComponent({ isLoading, ...props }) {
    if (isLoading) {
      return <div>Loading...</div>;
    }
    return <Component {...props} />;
  };
}

// Original component
function UserList({ users }) {
  return (
    <ul>
      {users.map(user => <li key={user.id}>{user.name}</li>)}
    </ul>
  );
}

// Enhanced component
const UserListWithLoading = withLoading(UserList);

// Usage
function App() {
  const [users, setUsers] = useState([]);
  const [isLoading, setIsLoading] = useState(true);

  return <UserListWithLoading isLoading={isLoading} users={users} />;
}
```

---

30. What are Custom Hooks?

Answer: Custom Hooks are JavaScript functions that use built-in hooks and can contain stateful logic. They let you reuse logic across components.

Rules:

- Name must start with "use"
- Can call other hooks
- Must follow hooks rules

Example:

```
// Custom hook for form handling
function useForm(initialValues) {
  const [values, setValues] = useState(initialValues);

  const handleChange = (e) => {
    const { name, value } = e.target;
    setValues(prev => ({
      ...prev,
      [name]: value
    }));
  };

  const reset = () => {
    setValues(initialValues);
  };

  return { values, handleChange, reset };
}

// Usage
function LoginForm() {
  const { values, handleChange, reset } = useForm({
    email: '',
    password: ''
  });

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log(values);
    reset();
  };

  return (
    <form onSubmit={handleSubmit}>
      <input name="email" value={values.email} onChange={handleChange} />
      <input name="password" value={values.password} onChange={handleChange} />
      <button type="submit">Submit</button>
    </form>
  );
}

// Custom hook for API fetching
function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
```

```
    setLoading(true);
    fetch(url)
      .then(res => res.json())
      .then(data => {
        setData(data);
        setLoading(false);
      })
      .catch(err => {
        setError(err);
        setLoading(false);
      });
  }, [url]);

  return { data, loading, error };
}

// Usage
function Users() {
  const { data, loading, error } = useFetch('/api/users');

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error.message}</div>;

  return (
    <ul>
      {data.map(user => <li key={user.id}>{user.name}</li>)}
    </ul>
  );
}
```

Common Interview Questions

31. What happens when you call setState?

Answer:

1. React schedules a re-render
2. State is updated asynchronously (batched)
3. Component re-renders with new state
4. Virtual DOM is created and compared
5. Real DOM is updated with changes

Important: setState is asynchronous!

```
function Counter() {
```

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

  const handleClick = () => {
    setCount(count + 1);
    console.log(count); // Still 0! (old value)

    // To use updated value, use functional update
    setCount(prev => {
      console.log(prev); // Updated value
      return prev + 1;
    });
  };

  return <button onClick={handleClick}>Count: {count}</button>;
}
```

---

32. What is the difference between createElement and cloneElement?

Answer:

createElement: Creates a new React element from scratch

```
React.createElement('div', { className: 'container' }, 'Hello');
// Equivalent to: <div className="container">Hello</div>
```

cloneElement: Clones an existing element and optionally adds/overrides props

```
const element = <div>Hello</div>;
const cloned = React.cloneElement(element, { className: 'container' });
```

---

33. How do you handle errors in React?

Answer: Use Error Boundaries to catch JavaScript errors in component tree.

Example:

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
```

```
  }

  componentDidCatch(error, errorInfo) {
    console.log('Error:', error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}

// Usage
function App() {
  return (
    <ErrorBoundary>
      <MyComponent />
    </ErrorBoundary>
  );
}
```

---

34. What is lazy loading in React?

Answer: Lazy loading defers loading of components until they're needed, reducing initial bundle size.

Example:

```
import React, { lazy, Suspense } from 'react';

// Lazy load component
const HeavyComponent = lazy(() => import('./HeavyComponent'));

function App() {
  return (
    <div>
      <h1>My App</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <HeavyComponent />
      </Suspense>
    </div>
  );
}
```

---

35. What is the difference between useEffect and useLayoutEffect?

Answer:

useEffect:

- Runs asynchronously after render
- Doesn't block browser paint
- Use for most side effects (API calls, subscriptions)

useLayoutEffect:

- Runs synchronously after DOM mutations but before browser paint
- Blocks visual updates
- Use for DOM measurements or mutations that need to happen before paint

Example:

```
function Tooltip() {
  const [coords, setCoords] = useState({ x: 0, y: 0 });
  const ref = useRef();

  // Use useLayoutEffect to measure DOM before paint
  useLayoutEffect(() => {
    const rect = ref.current.getBoundingClientRect();
    setCoords({ x: rect.left, y: rect.top });
  }, []);

  return <div ref={ref}>Tooltip</div>;
}
```

State Management

36. What is Redux and when would you use it?

Answer: Redux is a predictable state management library. Use it for complex state that needs to be shared across many components.

Core Concepts:

- **Store:** Single source of truth for state
- **Actions:** Plain objects describing what happened
- **Reducers:** Pure functions that specify how state changes
- **Dispatch:** Method to send actions to store

When to use Redux:

- Large apps with complex state
- State needed by many components
- State updates follow complex logic
- Need to track state changes over time

Example:

```
// Action
const increment = () => ({ type: 'INCREMENT' });

// Reducer
function counterReducer(state = { count: 0 }, action) {
  switch (action.type) {
    case 'INCREMENT':
      return { count: state.count + 1 };
    default:
      return state;
  }
}

// Store
import { createStore } from 'redux';
const store = createStore(counterReducer);

// Component
function Counter() {
  const count = useSelector(state => state.count);
  const dispatch = useDispatch();

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

---

37. What is useReducer hook?

Answer: `useReducer` is an alternative to `useState` for complex state logic. It's similar to Redux reducers.

When to use:

- Complex state logic
- Multiple sub-values
- Next state depends on previous
- State updates involve complex logic

Example:

```
// Reducer function
function todoReducer(state, action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [...state, { id: Date.now(), text: action.text, done: false }];
    case 'TOGGLE_TODO':
      return state.map(todo =>
        todo.id === action.id ? { ...todo, done: !todo.done } : todo
      );
    case 'DELETE_TODO':
      return state.filter(todo => todo.id !== action.id);
    default:
      return state;
  }
}

function TodoApp() {
  const [todos, dispatch] = useReducer(todoReducer, []);
  const [input, setInput] = useState('');

  const addTodo = () => {
    dispatch({ type: 'ADD_TODO', text: input });
    setInput('');
  };

  return (
    <div>
      <input value={input} onChange={(e) => setInput(e.target.value)} />
      <button onClick={addTodo}>Add</button>
      <ul>
        {todos.map(todo => (
          <li key={todo.id}>
            <span
              style={{ textDecoration: todo.done ? 'line-through' : 'none' }}
              onClick={() => dispatch({ type: 'TOGGLE_TODO', id: todo.id })}
            >
              {todo.text}
            </span>
            <button onClick={() => dispatch({ type: 'DELETE_TODO', id: todo.id })}>
              Delete
            </button>
```

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

Testing

38. How do you test React components?

Answer: Common tools: Jest (test runner) and React Testing Library (component testing).

Types of tests:

1. **Unit tests:** Test individual components
2. **Integration tests:** Test component interactions
3. **Snapshot tests:** Test UI doesn't change unexpectedly

Example:

```
import { render, screen, fireEvent } from '@testing-library/react';
import Counter from './Counter';

describe('Counter Component', () => {
 test('renders initial count', () => {
  render(<Counter />);
  expect(screen.getByText(/count: 0/i)).toBeInTheDocument();
 });

 test('increments count on button click', () => {
  render(<Counter />);
  const button = screen.getByRole('button', { name: /increment/i });

  fireEvent.click(button);

  expect(screen.getByText(/count: 1/i)).toBeInTheDocument();
 });

 test('snapshot test', () => {
  const { container } = render(<Counter />);
  expect(container).toMatchSnapshot();
 });
});
```

API Integration

39. How do you make API calls in React?

Answer: Common approaches: fetch API, axios, or custom hooks.

Example with fetch:

```
function UserList() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    fetch('https://api.example.com/users')
      .then(response => {
        if (!response.ok) throw new Error('Failed to fetch');
        return response.json();
      })
      .then(data => {
        setUsers(data);
        setLoading(false);
      })
      .catch(err => {
        setError(err.message);
        setLoading(false);
      });
  }, []);

  if (loading) return <div>Loading...</div>;
  if (error) return <div>Error: {error}</div>;

  return (
    <ul>
      {users.map(user => <li key={user.id}>{user.name}</li>)}
    </ul>
  );
}
```

Example with async/await:

```
function UserList() {
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchUsers = async () => {
```

```
      try {
        const response = await fetch('https://api.example.com/users');
        const data = await response.json();
        setUsers(data);
      } catch (err) {
        console.error(err);
      } finally {
        setLoading(false);
      }
    };

    fetchUsers();
  }, []);

  return loading ? <div>Loading...</div> : (
    <ul>
      {users.map(user => <li key={user.id}>{user.name}</li>)}
    </ul>
  );
}
```

---

40. How do you handle async operations in React?

Answer: Use useEffect with async functions, or libraries like React Query.

Best practices:

1. Clean up async operations on unmount
2. Handle loading and error states
3. Avoid setting state on unmounted components

Example:

```
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    let cancelled = false; // Cleanup flag

    const fetchUser = async () => {
      setLoading(true);
      try {
        const response = await fetch(`/api/users/${userId}`);
        const data = await response.json();
```

```
      if (!cancelled) { // Only update if not unmounted
        setUser(data);
      }
    } catch (err) {
      if (!cancelled) {
        console.error(err);
      }
    } finally {
      if (!cancelled) {
        setLoading(false);
      }
    }
  };

  fetchUser();

  return () => {
    cancelled = true; // Cleanup
  };
 }, [userId]);

 if (loading) return <div>Loading...</div>;
 return <div>{user?.name}</div>;
}
```

Styling

41. What are different ways to style React components?

Answer:

1. Inline Styles:

```
const style = { color: 'blue', fontSize: '20px' };
<div style={style}>Text</div>
```

2. CSS Stylesheets:

```
import './App.css';
<div className="container">Text</div>
```

3. CSS Modules:

```
import styles from './App.module.css';
```

```
<div className={styles.container}>Text</div>
```

4. Styled Components (CSS-in-JS):

```
import styled from 'styled-components';

const Button = styled.button`
  background: blue;
  color: white;
  padding: 10px;

  &:hover {
    background: darkblue;
  }
`;

<Button>Click me</Button>
```

5. Tailwind CSS:

```
<div className="bg-blue-500 text-white p-4 hover:bg-blue-700">
  Text
</div>
```

Refs

42. When should you use refs?

Answer: Use refs when you need to:

1. Access DOM elements directly
2. Store mutable values that don't trigger re-renders
3. Integrate with third-party DOM libraries

Don't use refs for:

- Data that should trigger re-renders (use state)
- Anything that can be done declaratively

Example:

```
function VideoPlayer() {
  const videoRef = useRef(null);
  const clickCountRef = useRef(0);
```

```
  const handlePlay = () => {
    videoRef.current.play(); // Direct DOM manipulation
    clickCountRef.current += 1; // Mutable value without re-render
    console.log('Played', clickCountRef.current, 'times');
  };

  const handlePause = () => {
    videoRef.current.pause();
  };

  return (
    <div>
      <video ref={videoRef} src="video.mp4" />
      <button onClick={handlePlay}>Play</button>
      <button onClick={handlePause}>Pause</button>
    </div>
  );
}
```

Fragment

43. What is React Fragment?

Answer: Fragment lets you group multiple elements without adding extra DOM nodes.

Why use it:

- Avoid unnecessary wrapper divs
- Keep DOM clean
- Return multiple elements from component

Example:

```
// Long syntax
function List() {
  return (
    <React.Fragment>
      <li>Item 1</li>
      <li>Item 2</li>
    </React.Fragment>
  );
}

// Short syntax
function List() {
```

```
  return (
    <>
      <li>Item 1</li>
      <li>Item 2</li>
    </>
  );
}

// With key (only long syntax supports key)
function Glossary({ items }) {
  return (
    <dl>
      {items.map(item => (
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
    </dl>
  );
}
```

Portal

44. What are Portals?

Answer: Portals provide a way to render children into a DOM node that exists outside the parent component's DOM hierarchy.

Use cases:

- Modals
- Tooltips
- Dropdowns
- Notifications

Example:

```
import ReactDOM from 'react-dom';

function Modal({ children, isOpen }) {
  if (!isOpen) return null;

  return ReactDOM.createPortal(
    <div className="modal-overlay">
      <div className="modal-content">
```

```
        {children}
      </div>
    </div>,
    document.getElementById('modal-root') // Render outside root
  );
}

function App() {
  const [isOpen, setIsOpen] = useState(false);

  return (
    <div>
      <h1>My App</h1>
      <button onClick={() => setIsOpen(true)}>Open Modal</button>

      <Modal isOpen={isOpen}>
        <h2>Modal Title</h2>
        <p>Modal content</p>
        <button onClick={() => setIsOpen(false)}>Close</button>
      </Modal>
    </div>
  );
}
```

Strict Mode

45. What is React Strict Mode?

Answer: StrictMode is a tool for highlighting potential problems in an application. It activates additional checks and warnings for its descendants.

What it checks:

- Identifies components with unsafe lifecycles
- Warns about legacy string ref API
- Warns about deprecated findDOMNode usage
- Detects unexpected side effects
- Warns about legacy context API

Note: Strict Mode renders components twice in development to detect side effects.

Example:

```
import React from 'react';

function App() {
```

```
  return (
    <React.StrictMode>
      <div>
        <Header />
        <Main />
        <Footer />
      </div>
    </React.StrictMode>
  );
}

// Can wrap entire app or specific components
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Common Patterns

46. What is the Render Props pattern?

Answer: Render Props is a technique for sharing code between components using a prop whose value is a function.

Example:

```
// Component with render prop
function Mouse({ render }) {
  const [position, setPosition] = useState({ x: 0, y: 0 });

  useEffect(() => {
    const handleMove = (e) => {
      setPosition({ x: e.clientX, y: e.clientY });
    };

    window.addEventListener('mousemove', handleMove);
    return () => window.removeEventListener('mousemove', handleMove);
  }, []);

  return render(position);
}

// Usage
```

```
function App() {
  return (
    <Mouse
      render={({ x, y }) => (
        <h1>Mouse position: {x}, {y}</h1>
      )}
    />
  );
}

// Alternative: children as function
function App() {
  return (
    <Mouse>
      {({ x, y }) => <h1>Mouse position: {x}, {y}</h1>}
    </Mouse>
  );
}
```

---

47. What is Component Composition?

Answer: Component Composition is building complex UIs by combining simpler components.

Example:

```
// Base components
function Card({ children }) {
  return <div className="card">{children}</div>;
}

function CardHeader({ children }) {
  return <div className="card-header">{children}</div>;
}

function CardBody({ children }) {
  return <div className="card-body">{children}</div>;
}

function CardFooter({ children }) {
  return <div className="card-footer">{children}</div>;
}

// Composed component
function UserCard({ user }) {
  return (
```

```
    <Card>
      <CardHeader>
        <h2>{user.name}</h2>
      </CardHeader>
      <CardBody>
        <p>Email: {user.email}</p>
        <p>Role: {user.role}</p>
      </CardBody>
      <CardFooter>
        <button>View Profile</button>
      </CardFooter>
    </Card>
  );
}
```

Best Practices

48. What are React best practices you follow?

Answer:

1. Component Structure:

- Keep components small and focused
- Use functional components with hooks
- Extract reusable logic into custom hooks

2. State Management:

- Keep state as local as possible
- Lift state up only when needed
- Use Context for global state, not prop drilling

3. Performance:

- Use React.memo for expensive components
- Use useMemo and useCallback appropriately
- Avoid inline functions and objects in render

4. Code Quality:

- Use meaningful component and variable names
- Add PropTypes or TypeScript for type checking
- Keep consistent file structure

5. Naming Conventions:

- Components: PascalCase (UserProfile.jsx)
- Functions/variables: camelCase (handleClick)
- Constants: UPPER_SNAKE_CASE (API_URL)
- Custom hooks: use prefix (useAuth, useFetch)

Example:

```
// Good: Small, focused component
function UserAvatar({ src, alt, size = 'medium' }) {
  return (
    <img
      src={src}
      alt={alt}
      className={`avatar avatar-${size}`}
    />
  );
}

// Good: Custom hook for reusable logic
function useAuth() {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    // Auth logic
  }, []);

  return { user, loading };
}

// Good: Memoized expensive calculation
function ProductList({ products, filter }) {
  const filteredProducts = useMemo(() => {
    return products.filter(p => p.category === filter);
  }, [products, filter]);

  return <div>{/* render products */}</div>;
}
```

---

49. How do you prevent unnecessary re-renders?

Answer:

1. Use React.memo:

```
const ExpensiveComponent = React.memo(({ data }) => {
```

```
  return <div>{data}</div>;
});
```

2. Use useMemo for values:

```
const sortedList = useMemo(() => items.sort(), [items]);
```

3. Use useCallback for functions:

```
const handleClick = useCallback(() => {
  doSomething();
}, []);
```

4. Split components:

```
// Instead of one large component
function LargeComponent() {
  const [count, setCount] = useState(0);
  const [data, setData] = useState([]);

  return (
    <>
      <button onClick={() => setCount(count + 1)}>{count}</button>
      <ExpensiveList data={data} />
    </>
  );
}

// Split into smaller components
function Counter() {
  const [count, setCount] = useState(0);
  return <button onClick={() => setCount(count + 1)}>{count}</button>;
}

function DataList() {
  const [data, setData] = useState([]);
  return <ExpensiveList data={data} />;
}
```

5. Move state down:

```
// Bad: State at top level causes everything to re-render
function App() {
  const [formData, setFormData] = useState({});
  return (
```

```
    <>
      <Header />
      <Form data={formData} onChange={setFormData} />
      <Footer />
    </>
  );
}

// Good: State only in Form
function App() {
  return (
    <>
      <Header />
      <Form />
      <Footer />
    </>
  );
}
```

---

50. What are common mistakes to avoid in React?

Answer:

1. Mutating state directly:

```
// Bad
state.items.push(newItem);
setState(state);

// Good
setState({ items: [...state.items, newItem] });
```

2. Using index as key:

```
// Bad
{items.map((item, index) => <li key={index}>{item}</li>)}

// Good
{items.map(item => <li key={item.id}>{item}</li>)}
```

3. Not cleaning up effects:

```
// Bad
useEffect(() => {
  const interval = setInterval(() => {}, 1000);
```

```
}, []);

// Good
useEffect(() => {
  const interval = setInterval(() => {}, 1000);
  return () => clearInterval(interval); // Cleanup
}, []);
```

4. Forgetting dependencies in useEffect:

```
// Bad
useEffect(() => {
  fetchData(userId);
}, []); // Missing userId dependency

// Good
useEffect(() => {
  fetchData(userId);
}, [userId]);
```

5. Too many useState calls:

```
// Bad
const [firstName, setFirstName] = useState('');
const [lastName, setLastName] = useState('');
const [email, setEmail] = useState('');
const [phone, setPhone] = useState('');

// Good
const [formData, setFormData] = useState({
  firstName: '',
  lastName: '',
  email: '',
  phone: ''
});
```

Interview Tips

General Advice:

1. **Understand the basics thoroughly:** Props, state, lifecycle, hooks
2. **Practice coding:** Build small projects, solve problems
3. **Know when to use what:** useState vs useReducer, Context vs Redux
4. **Performance matters:** Know optimization techniques

5. **Be honest:** Say "I don't know" rather than guessing
6. **Think aloud:** Explain your thought process
7. **Ask clarifying questions:** Understand requirements before coding

Key Topics to Master:

- ✅ Hooks (useState, useEffect, useContext, useMemo, useCallback)
- ✅ Component lifecycle
- ✅ State management (Context API, basics of Redux)
- ✅ Performance optimization
- ✅ Forms and controlled components
- ✅ API integration
- ✅ Routing (React Router basics)
- ✅ Error handling
- ✅ Testing basics

Good luck with your interview! 🚀