React State Management: Complete Interview Guide

Table of Contents

---

1. What is State in React?

Definition

State is a built-in React object used to contain data or information about a component. When state changes, the component re-renders to reflect those changes in the UI.

Key Characteristics

- **Mutable**: State can be changed over time
- **Triggers Re-renders**: When state updates, React re-renders the component
- **Component-Specific**: Each component instance has its own state
- **Asynchronous Updates**: State updates may be batched for performance

---

2. Class Component State vs useState Hook

Class Component State (Legacy Approach)

Syntax

```javascript
```

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    // Initialize state as an object
    this.state = {
      count: 0,
      name: 'John'
    };
  }

  // Update state using this.setState()
  incrementCount = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <p>Name: {this.state.name}</p>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}
```

Key Features

- State is always an **object**
- Initialize in constructor
- Access via `this.state.propertyName`
- Update via `this.setState()`
- Can update multiple properties at once
- Automatically merges state updates

---

useState Hook (Modern Approach)

Syntax

```
javascript
```

```jsx
import { useState } from 'react';

function Counter() {
  // Declare state variables individually
  const [count, setCount] = useState(0);
  const [name, setName] = useState('John');

  const incrementCount = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <p>Name: {name}</p>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
}
```

Key Features

- State can be **any data type** (number, string, object, array, boolean)
- Declare multiple state variables separately
- Access directly by variable name
- Update via setter function
- Does NOT automatically merge - replaces the entire value
- Cleaner, more readable syntax

---

3. Detailed Comparison: Class State vs useState

Feature	Class Component State	useState Hook
Syntax	`this.state = { count: 0 }`	`const [count, setCount] = useState(0)`
Initialization	In constructor	During function call
Access	`this.state.count`	`count`
Update	`this.setState({ count: 1 })`	`setCount(1)`
Data Type	Must be an object	Any type
Merging	Automatically merges	Replaces entirely
Multiple States	Single object with properties	Multiple separate states

Feature	Class Component State	useState Hook
Binding	Need to bind methods	No binding needed
Component Type	Class components only	Function components only

---

4. State vs Regular Variables

Regular Variables (Don't Use for Dynamic UI)

```javascript
function Counter() {
  let count = 0; // Regular variable

  const increment = () => {
    count = count + 1;
    console.log(count); // This will log updated value
    // BUT UI won't update!
  };

  return (
    <div>
      <p>Count: {count}</p> {/* Always shows 0 */}
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

Problem: UI doesn't re-render when the variable changes!

State Variables (Correct Approach)

```javascript
```

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

  const increment = () => {
    setCount(count + 1);
    // UI automatically re-renders!
  };

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

Solution: State triggers re-render, updating the UI!

Comparison Table: Variables vs State

Aspect	Regular Variables	State Variables
Re-render	❌ No re-render on change	✅ Triggers re-render
Persistence	❌ Resets on every render	✅ Persists between renders
UI Update	❌ Manual DOM manipulation needed	✅ Automatic UI update
Use Case	Static values, calculations	Dynamic data, user interactions
Syntax	`let count = 0`	`const [count, setCount] = useState(0)`
Update	Direct assignment: `count = 5`	Setter function: `setCount(5)`

---

5. Deep Dive Code Examples

Example 1: Multiple State Variables

```
javascript
```

```
function UserProfile() {
  const [name, setName] = useState('Alice');
  const [age, setAge] = useState(25);
  const [isOnline, setIsOnline] = useState(true);

  return (
    <div>
      <p>Name: {name}</p>
      <p>Age: {age}</p>
      <p>Status: {isOnline ? 'Online' : 'Offline'}</p>

      <button onClick={() => setAge(age + 1)}>
        Increase Age
      </button>
      <button onClick={() => setIsOnline(!isOnline)}>
        Toggle Status
      </button>
    </div>
  );
}
```

Example 2: Object State

javascript

```javascript
// ❌ WRONG: Direct mutation
function UserProfile() {
  const [user, setUser] = useState({ name: 'Alice', age: 25 });

  const updateAge = () => {
    user.age = 26; // This won't trigger re-render!
    setUser(user);
  };

  return <button onClick={updateAge}>Update</button>;
}

// ✅ CORRECT: Create new object
function UserProfile() {
  const [user, setUser] = useState({ name: 'Alice', age: 25 });

  const updateAge = () => {
    setUser({ ...user, age: 26 }); // Spread operator creates new object
  };

  return <button onClick={updateAge}>Update</button>;
}
```

Example 3: Functional Updates

javascript

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

  // ❌ May cause issues with async updates
  const incrementThreeTimes = () => {
    setCount(count + 1);
    setCount(count + 1);
    setCount(count + 1);
    // Only increments by 1!
  };

  // ✅ CORRECT: Use functional form
  const incrementThreeTimesCorrect = () => {
    setCount(prev => prev + 1);
    setCount(prev => prev + 1);
    setCount(prev => prev + 1);
    // Correctly increments by 3!
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={incrementThreeTimesCorrect}>+3</button>
    </div>
  );
}
```

Example 4: Array State

javascript

```
function TodoList() {
  const [todos, setTodos] = useState(['Learn React', 'Build Project']);

  const addTodo = () => {
    // ✅ Correct: Create new array
    setTodos([...todos, 'New Todo']);
  };

  const removeTodo = (index) => {
    // ✅ Correct: Filter creates new array
    setTodos(todos.filter((_, i) => i !== index));
  };

  return (
    <div>
      {todos.map((todo, index) => (
        <div key={index}>
          <span>{todo}</span>
          <button onClick={() => removeTodo(index)}>Remove</button>
        </div>
      ))}
      <button onClick={addTodo}>Add Todo</button>
    </div>
  );
}
```

---

6. Interview Questions & Answers

Q1: Why do we need state instead of regular variables?

Answer: Regular variables reset on every render and don't trigger re-renders. State persists between renders and automatically updates the UI when changed. React needs to know when to re-render, and state changes signal this.

Q2: What happens when you call setState or useState setter?

Answer:

1. React schedules a re-render
2. The component function runs again
3. React compares the new virtual DOM with the previous one
4. Only changed elements are updated in the real DOM

5. State updates may be batched for performance

Q3: Why can't we directly mutate state?

Answer: React uses reference comparison to detect changes. If you mutate state directly, the reference stays the same, so React won't know to re-render. Always create new objects/arrays when updating state.

```javascript
// ❌ Wrong
state.count = 5;

// ✅ Correct
setState({ count: 5 });
```

Q4: What's the difference between this.setState in class and useState setter?

Answer:

- `this.setState` **merges** the update with existing state
- `useState` setter **replaces** the entire value

```javascript
// Class: Merges
this.setState({ count: 1 }); // Other properties remain

// useState: Replaces
setUser({ name: 'Bob' }); // age is lost if not included!
setUser({ ...user, name: 'Bob' }); // Correct way
```

Q5: When should you use functional updates?

Answer: Use functional updates when the new state depends on the previous state, especially in async operations or when calling multiple updates in succession.

```javascript
// When dependent on previous state
setCount(prevCount => prevCount + 1);

// When updating multiple times
setCount(prev => prev + 1);
setCount(prev => prev + 1); // Correctly increments by 2
```

Q6: Can you use useState in class components?

Answer: No, hooks (including useState) can only be used in functional components. Class components must use `this.state` and `this.setState`.

Q7: How many times can you call useState in a component?

Answer: You can call useState multiple times in a single component. Each call creates a separate state variable.

```javascript
const [name, setName] = useState('');
const [age, setAge] = useState(0);
const [email, setEmail] = useState('');
// All three are independent state variables
```

Q8: Is state update synchronous or asynchronous?

Answer: State updates are **asynchronous**. React batches multiple setState calls for performance. You can't immediately access the updated state after calling the setter.

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

const increment = () => {
  setCount(count + 1);
  console.log(count); // Still logs 0, not 1!
};
```

---

7. Best Practices

✅ **DO:**

- Use state for data that changes and affects the UI
- Initialize state with appropriate default values
- Use functional updates when new state depends on old state
- Keep state as simple as possible
- Use multiple useState calls for unrelated data

- Create new objects/arrays when updating (immutability)

❌ **DON'T:**

- Use state for values that don't affect rendering
- Mutate state directly
- Use regular variables for dynamic UI data
- Store props in state (usually)
- Have too much state (consider derived values)
- Update state during render

---

8. Quick Reference Cheat Sheet

javascript

```
// Class Component State
class Example extends React.Component {
  state = { count: 0 };

  updateCount = () => {
    this.setState({ count: this.state.count + 1 });
  }
}

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

  const updateCount = () => {
    setCount(count + 1);
  };
}

// State Types
const [string, setString] = useState('hello');
const [number, setNumber] = useState(42);
const [boolean, setBoolean] = useState(true);
const [array, setArray] = useState([1, 2, 3]);
const [object, setObject] = useState({ key: 'value' });

// Updating Patterns
setCount(count + 1);                // Direct
setCount(prev => prev + 1);         // Functional
setArray([...array, newItem]);      // Array
setObject({ ...object, key: 'new' }); // Object
```

Summary

- **State** is for dynamic data that affects the UI
- **Class state** uses `this.state` and `this.setState()` with automatic merging
- **useState** is the modern hook approach with cleaner syntax
- **Regular variables** don't trigger re-renders
- Always update state **immutably**
- Use **functional updates** when depending on previous state
- State updates are **asynchronous**

This guide covers everything you need to know about React state for interviews. Practice the code examples and understand the concepts deeply!