React Interview Notes (Core Concepts)

These notes are **interview-only**, concise, and cover concepts **up to `useState` and `useEffect`**, exactly what most React interviews expect at junior–mid level.

---

1. What is React?

React is a **JavaScript library for building user interfaces**, primarily single-page applications. It uses a **component-based architecture** and focuses on efficient UI updates.

---

2. What is the Virtual DOM?

The **Virtual DOM** is a lightweight JavaScript representation of the real DOM.

How it works:

1. React creates a virtual copy of the UI
2. When state/props change, a **new Virtual DOM tree** is created
3. React compares the new tree with the previous one
4. Only the changed parts are updated in the real DOM

Why Virtual DOM?

- Real DOM updates are expensive
- Virtual DOM minimizes direct DOM manipulation
- Improves performance

---

3. What is Reconciliation?

Reconciliation is the process React uses to: - Compare old Virtual DOM with new Virtual DOM - Identify differences (diffing) - Update only the changed nodes in the real DOM

👉Reconciliation makes React fast and efficient.

---

4. What are Components?

Components are **reusable, independent UI pieces**.

Types: - Functional Components - Class Components

---

5. What are Hooks?

Hooks are special functions introduced in **React 16.8** that allow **functional components to use features that were previously available only in class components**, such as state and lifecycle methods.

Before Hooks (Old Approach)

- Only **class components** could use:
- State (`this.state`)
- Lifecycle methods (`componentDidMount` , `componentDidUpdate` , etc.)
- Functional components were called **stateless components**
- Logic reuse was difficult and often required patterns like HOCs or render props

After Hooks (New Approach)

- Functional components can now:
- Manage state using `useState`
- Handle lifecycle and side effects using `useEffect`
- No need to convert a component into a class just to use state

Why Hooks Were Introduced (Simple Explanation)

Hooks were introduced to: - Reduce complexity of class components - Remove confusion around the `this` keyword - Make logic reuse easier - Write cleaner and more readable components

In Simple Words ⭐

> Hooks allow us to write **stateful and lifecycle-aware logic inside functional components**, without using classes.

---

6. Why Hooks Were Introduced?

Problems with class components: - Complex lifecycle methods - `this` keyword confusion - Harder to reuse logic - Large and less readable components

Hooks solve these problems by: - Simplifying logic - Improving readability - Encouraging reusable logic

---

7. Functional Components vs Class Components

Architectural Difference (Internal View)

Class Components - Each component instance is a JavaScript class - State is stored on `this.state` - Lifecycle methods are separate entry points - React manages class instances and binds lifecycle calls

Functional Components with Hooks - Components are plain functions - State is stored in React's internal fiber tree - Hooks register state and effects during render - No instance, no `this`, fewer memory allocations

Why Hooks Are Preferred (Internals)

- Better tree shaking
- Easier concurrent rendering
- More predictable execution model
- Enables React features like **Concurrent Mode & Server Components**

---

8. What is State?

State is **mutable data** owned by a component that determines how the UI behaves.

- Local to component
- Changes cause re-render

---

9. What is useState?

`useState` is a hook that allows functional components to manage state.

Syntax

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

- `state` → current value
- `setState` → updates state and triggers re-render

---

10. Why State Updates Cause Re-render?

React re-renders because: - UI depends on state - State change → new Virtual DOM - Reconciliation updates UI

---

11. What is useEffect?

`useEffect` is a hook used to handle **side effects** in functional components.

Side effects include: - API calls - DOM updates - Subscriptions - Timers

---

12. Component Lifecycle (Simplified)

Phase	Description
Mount	Component appears

Phase	Description
Update	State/props change
Unmount	Component removed

`useEffect` can manage all these phases.

---

13. useEffect Syntax

```
useEffect(() => {
  // side effect
  return () => {
    // cleanup
  };
}, [dependencies]);
```

---

14. Dependency Array Scenarios

1. `[]` → runs once (on mount)
2. `[value]` → runs when value changes
3. No array → runs on every render

---

15. Cleanup Function in useEffect

Used to: - Remove event listeners - Clear timers - Cancel subscriptions

Prevents **memory leaks**.

---

16. Lifting State Up

From an internal React perspective: - State updates trigger a re-render of the owning fiber - Child components cannot safely share mutable state - Lifting state up ensures **one fiber owns the data**

This reduces: - State divergence - Unnecessary reconciliation paths

---

17. Controlled vs Uncontrolled Components

Controlled Components

A **controlled component** is one where: - Form data is stored in React state - React is the single source of truth - Every input change triggers a state update

Internally: - Input value → state → Virtual DOM → reconciliation - Guarantees predictable UI

Used when: - Validation is required - Dynamic UI updates are needed - Form data must be synced across components

Uncontrolled Components

An **uncontrolled component** is one where: - Form data is managed by the DOM itself - React accesses values using refs

Important clarification (senior-level):

> Inputs are **not uncontrolled just because they change**. They are uncontrolled **only if React does not manage their value**.

Example: - Using `defaultValue` - Reading value via `ref.current.value`

Internally: - React does not track value changes - No re-render on input change

Why Controlled Components Are Preferred

- Better predictability
- Easier debugging
- Works naturally with reconciliation

---

18. Common Interview Questions (Deep Understanding)

Q1. Why does React enforce hook call order?

React maps hook calls by position in the fiber's hook list. Changing order breaks state association.

Q2. Why are uncontrolled components faster?

They avoid state updates and reconciliation, but sacrifice control and predictability.

Q3. Why can't hooks be used conditionally?

Because React relies on deterministic hook order during render.

Q4. Why are class components harder to optimize?

They create instances, bind methods, and fragment lifecycle logic.

Q5. What happens internally when setState/useState is called?

- Update is queued
- Fiber marked dirty
- Reconciliation scheduled
- Commit phase updates DOM

---

19. Senior-Level One-Liners ⭐

- Hooks align React with functional programming principles
- Controlled components trade performance for correctness
- Reconciliation minimizes DOM mutations, not renders
- State lives in fibers, not components

Q2. What happens when state changes?

- New Virtual DOM created
- Reconciliation runs
- UI updates

Q3. Can we use multiple useState hooks?

Yes. Each manages independent state.

Q4. Why is useEffect asynchronous?

To avoid blocking rendering and improve performance.

Q5. What is prop drilling?

Passing props through multiple layers unnecessarily.

Q6. How to avoid prop drilling?

- Context API
- State management libraries

---

18. Interview One-Liners ⭐

- **Hooks:** Enable state and lifecycle features in functional components
- **Virtual DOM:** Lightweight copy of real DOM
- **Reconciliation:** Process of syncing Virtual DOM with real DOM

- **useState:** Manages component state
- **useEffect:** Handles side effects

Final Tip for Interviews

> If you explain **Virtual DOM → Reconciliation → State → useEffect**, interviewers know you understand React fundamentals deeply.

Visual Diagrams (Textual – Interview Friendly)

Virtual DOM vs Real DOM (Conceptual Diagram)

```
User Action
    ↓
setState / setProps
    ↓
React Render Phase
    ↓
Virtual DOM (New Tree)
    ↓           🥹
Old Virtual DOM   Diffing (Reconciliation)
                    ↓
              Minimal Changes
                    ↓
              Real DOM Update
```

Key Interview Points

- Virtual DOM is **in-memory JavaScript object**
- Real DOM updates are **expensive**
- React minimizes DOM mutations using reconciliation

Reconciliation Process (Step-by-Step Diagram)

```
State / Props Change
        ↓
Component Re-renders
        ↓
New Virtual DOM Tree Created
        ↓
Diffing Algorithm Compares Trees
        ↓
Identifies Changed Nodes Only
```

```
        ↓
Commit Phase Updates Real DOM
```

Important Clarification ⭐

> **Reconciliation is NOT re-rendering.** Re-render creates a new Virtual DOM; reconciliation decides what actually changes in the Real DOM.

Execution Flow: useState & Rendering (Very Important)

Common Interview Confusion 🦡

> “useState runs after rendering”

Correct Execution Order 🦜

```
Initial Render
  ↓
Function Component Executes (Top to Bottom)
  ↓
useState() is READ (not executed like a function)
  ↓
JSX Returned
  ↓
Virtual DOM Created
  ↓
Commit Phase (DOM Painted)
```

When setState / setCount is Called

```
setState Called
   ↓
Update Queued (Not Immediate)
   ↓
Component Scheduled for Re-render
   ↓
Function Component Re-executes
   ↓
New Virtual DOM Created
   ↓
Reconciliation
   ↓
DOM Updated
```

Key Senior-Level Insight ⭐

- `useState` does **not run later**
- It is executed **during render**
- State updates are **batched and async**

Props in React (Step-by-Step – Interview Flow)

1. What are Props?

Props (short for *properties*) are **inputs passed from a parent component to a child component**.

- Props are **read-only**
- Props flow in **one direction (parent → child)**
- Props help make components **reusable and configurable**

> Think of props as function arguments passed to a component.

---

2. How Props Work Internally

From React's internal perspective: - Props are part of the **render input** of a component - When props change: - Component re-renders - New Virtual DOM is created - Reconciliation determines DOM updates

> Props change → render → reconciliation → commit

---

3. Why Props Are Immutable

Props are immutable because: - Predictable one-way data flow - Prevents accidental data mutation - Easier debugging and reasoning

If a child needs to update data: - Parent passes a **callback function** as a prop - Child calls the function

---

4. Common Props Interview Questions

Q1. Can a child component modify props?
👈No. Props are read-only.

Q2. How can a child update parent data?
By calling a function passed as a prop.

Q3. What happens when props change?
Component re-renders and reconciliation runs.

State in React (Next Step in Interview Flow)

5. What is State?

State is **data owned and managed by a component itself**.

- State is **mutable**
- State changes trigger re-render
- State controls dynamic UI behavior

> Props come from outside, state lives inside the component.

---

6. Why State Exists

State exists because: - UI changes over time - User interactions modify data - Components need to remember values

Internally: - State updates create a new Virtual DOM - Reconciliation updates the UI

---

7. Props vs State (Very Important – After State)

Feature	Props	State
Ownership	Parent	Component itself
Mutability	Immutable	Mutable
Purpose	Pass data	Manage data
Update Trigger	Parent re-render	setState / useState
Scope	External	Internal

Interview One-Liner ⭐

> Props are **passed into** a component, while state is **managed within** a component.

Final Senior-Level Summary ⭐⭐⭐

- Props define **what a component receives**
- State defines **how a component behaves**
- Props change → re-render
- State update → re-render
- Both participate in Virtual DOM and reconciliation

> Explaining React in this order shows **clear mental models and senior-level understanding**.