React Hooks, State, Props & Forms – Structured Notes

---

1. What are Hooks

Hooks are a feature in React that let you use **state, lifecycle behavior, and other React features inside functional components**, without writing class components.

Hooks are special functions provided by React that allow functional components to use state, lifecycle methods, and other React features.

---

2. Why Hooks Were Introduced

Before Hooks

- State & lifecycle → only in class components
- Functional components → stateless & simple

Problems with Class Components

- Confusing `this` keyword
- Hard to reuse logic (HOCs, render props)
- Long, complex lifecycle methods

How Hooks Help

- Allow logic reuse
- Cleaner and simpler code
- No classes required

---

3. Class Component vs Functional Component (Hooks)

Class Component Example

```
class Counter extends React.Component {
  state = { count: 0 };

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

  render() {
    return (
      <button onClick={this.increment}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

Functional Component with Hooks

```
import { useState } from "react";

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

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

---

4. Types of Hooks

Core Hooks

- useState
- useEffect
- useContext
- useRef
- useReducer

Latest Hooks

- useId
- useTransition
- useDeferredValue
- useSyncExternalStore

Optimization Hooks

- useMemo
- useCallback
- useLayoutEffect
- React.memo

---

5. React Lifecycle

Lifecycle refers to the different phases a component goes through from creation to removal.

Phase	Meaning
Mounting	Component is created and added to the DOM
Updating	Component re-renders due to state/props changes
Unmounting	Component is removed from the DOM

---

6. Lifecycle in Class Components

Mounting Phase (Birth)

Method	Use
constructor()	Initialize state, bind methods
render()	Returns JSX
componentDidMount()	Runs once after first render (API calls, subscriptions)

Updating Phase (Growth)

Happens when: - State changes - Props change

Method	Use
render()	Re-renders UI
componentDidUpdate()	Runs after update

```
componentDidUpdate(prevProps, prevState) {
  if (prevProps.id !== this.props.id) {
    console.log("Props changed");
  }
}
```

Unmounting Phase (Death)

Method	Use
componentWillUnmount()	Cleanup (timers, listeners, subscriptions)

```
componentWillUnmount() {
  console.log("Component removed");
}
```

7. Lifecycle in Functional Components (Hooks)

In modern React, we mainly use **useEffect**.

`useEffect` covers ALL lifecycle phases.

```
useEffect(() => {
  console.log("Mounted");

  return () => {
    console.log("Unmounted");
  };
}, []);
```

Mapping Lifecycle to Hooks

Lifecycle Phase	Hook Equivalent
Mounting	useEffect(() => {}, [])
Updating	useEffect(() => {}, [deps])
Unmounting	Cleanup function in useEffect

8. What is State in React

State is **internal data** of a component.

- Owned and managed by the component itself
- Mutable (can change)
- When state changes → component re-renders

```
const Counter = () => {
  const [count, setCount] = React.useState(0);

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

Key Points About State

- Local to the component
- Can change using setState / setCount
- Causes re-render
- Private unless passed down

9. What are Props in React

Props (properties) are **data passed from parent to child component**.

- Read-only
- Child cannot modify props
- Used to make components reusable

```
const Child = ({ name }) => {
  return <h1>Hello {name}</h1>;
};

const Parent = () => {
  return <Child name="Rohit" />;
};
```

Key Points About Props

- Passed from parent → child
- Immutable inside child
- Used for communication
- Makes component configurable

10. State vs Props

Feature	State	Props
Owned by	Component itself	Parent component
Mutable	✅Yes	❌No
Who can change	Same component	Only parent
Purpose	Manage internal data	Pass data to child
Re-render	On change	On receiving new props

11. State Passed as Props (Ownership Rule)

State does **NOT lose its identity** when passed as props.

👉Ownership is the REAL difference.

Component	Role
Parent	Owns count as state
Child	Receives count as props
Child	❌Cannot change count

---

12. Can Child Change Parent State?

- ❌Directly → NO
- ✅Indirectly → YES (via function props)

```
const Parent = () => {
  const [count, setCount] = React.useState(0);

  return <Child count={count} increment={setCount} />;
};

const Child = ({ count, increment }) => {
  return (
    <button onClick={() => increment(count + 1)}>
      {count}
    </button>
  );
};
```

👉Child requests change 👉Parent actually changes state

---

13. Lifting State Up

Lifting state up means:

Moving state from a child component to its nearest common parent so that multiple components can share the same data.

Why We Need It

- State should live in the component that needs to control it
- If multiple components need same state → place it in common parent

The component that needs to share data should not own the state — the parent should.

---

14. Controlled and Uncontrolled Components

Key Question

Who stores the value you type?

- Browser stores it → Uncontrolled
- React state stores it → Controlled

---

15. Uncontrolled Components (Browser is Boss)

React does NOT know the input value until you ask for it.

```
function UncontrolledForm() {
  const nameRef = React.useRef();

  const submit = () => {
    console.log(nameRef.current.value);
  };

  return (
    <>
      <input ref={nameRef} />
      <button onClick={submit}>Submit</button>
    </>
  );
}
```

What's Happening

- You type → browser stores value
- React doesn't re-render
- On submit → React reads value

☸️React is not controlling the input

Real Production Use: File Input

```
function UploadFile() {
  const fileRef = React.useRef();

  const upload = () => {
    const file = fileRef.current.files[0];
    console.log(file);
  };

  return <input type="file" ref={fileRef} />;
}
```

File inputs cannot be controlled properly ⛎This is standard production practice

---

16. Controlled Components (React is Boss)

React state is the **only place** where input value lives.

```
function ControlledForm() {
  const [name, setName] = React.useState("");

  return (
    <input
      value={name}
      onChange={(e) => setName(e.target.value)}
    />
  );
}
```

What’s Happening

- You type → onChange fires
- React updates state
- State updates → input updates

☸️Browser cannot change value directly

---

17. Why Controlled Components Are Needed

Controlled components store form data in React state, giving full control over validation and UI behavior.

Uncontrolled components rely on the DOM to manage state.

---

18. Controlled vs Uncontrolled (Final Comparison)

Criteria	Controlled	Uncontrolled
Default choice	✅YES	❌NO
React control	Full	Minimal

Criteria	Controlled	Uncontrolled
Validation	Easy	Hard
Performance	Slight overhead	Faster
File input	❌Not ideal	✅Required
Complexity	More code	Less code

19. Final Summary

- State is owned and managed by a component
- Props are read-only and passed from parent
- Ownership of state never changes
- Lifting state up enables shared data
- Controlled components are preferred in most production apps
- Uncontrolled components are used for specific cases like file inputs