**Difference Between Props and State**

Props = external, read-only, passed into a component.

State = internal, can change, managed within a component.  
  
  
  
  
Why bind?

bind(this) locks the this keyword to always refer to the current class instance.

**Why key is Needed**

React uses key to uniquely identify each item in the list.

Helps React know which items changed, added, or removed → efficient updates.

Without key, React may unnecessarily re-render all items

Lifecyle methods

1 Mounting-- When an instance of a component is begin created and inserted into the DOM

constructor() → good for initializing state, binding methods.

static getDerivedStateFromProps(props, state) → rarely used, update state based on props.

render() → returns JSX.

componentDidMount() → runs after the component is mounted → good for API calls, subscriptions.

2 Updating -- When a component is beging re-rendered as a result of change to either its props.

static getDerivedStateFromProps(props, state) → again runs before render.

shouldComponentUpdate(nextProps, nextState) → decide if re-render is needed (optimization).

render() → returns JSX.

getSnapshotBeforeUpdate(prevProps, prevState) → capture info (like scroll position) before DOM changes.

componentDidUpdate(prevProps, prevState, snapshot) → runs after update → good for API calls after change.

3: Umounting -- When a component is begin removed from the DOM

componentWillUnmount() → cleanup code (remove event listeners, cancel timers, unsubscribe).

4. Error Handling -- When there is an error during rendering in a lifecyle method, or in the constructor of any child component.

**What is a Constructor?**

In JavaScript, a constructor is a special method inside a class that is automatically called when a new object (or component) is created.

In React class components, the constructor is mainly used for:

Initializing state

Binding event handlers

**static getDerivedStateFromProps(props, state)**

It is a static lifecycle method (notice static)

Runs right before render() is called, both on initial mount and on updates.

Its purpose: update state based on props when the props change.

**What is render()?**

The render() method is mandatory in every React class component.

It tells React what to display on the UI.

It runs during mounting (when component first appears) and updating (when state or props change).

✅ Rules of render()

Must return JSX (or null if nothing to render).

Should be a pure function:

Same input (props, state) → same output (UI).

No side effects (don’t call setState, don’t fetch data here).

Can return:

JSX element

String / number

null

Array or fragment (<>...</>) for multiple elements

**componentDidMount()**

✅ What is it?

A lifecycle method that runs only once, after the component is rendered into the DOM (mounted).

Good place for side effects like:

Fetching data from APIs

Setting up subscriptions, timers

Initializing third-party libraries

**Updating Lifecycle Methods**

**1 Static getDerivedStateFromProps(props,state)**---- Method is called every time a components is re-rendered

**2 ShouldComponentUpdated(nextProps,nextState) --**Dictates if the components should re-render or not

A lifecycle method in React class components.

It decides whether the component should re-render when props or state change.

Default behavior = always re-render.

By overriding this, you can prevent unnecessary renders and improve performance.

**getSnapshotBeforeUpdate(prevProps, prevState)**

✅ What is it?

Runs right before the changes from render() are applied to the DOM.

Allows your component to capture some information from the DOM (like scroll position) before it changes.

The value it returns is passed as the third parameter to componentDidUpdate.

**componentDidUpdate(prevProps, prevState, snapshot)**

✅ What is it?

Runs after the component has updated (after render() and after DOM changes).

Useful for:

Performing side effects after state or props change

Making API calls based on prop changes

Updating the DOM manually using refs

A lifecycle method that runs just before a component is removed (unmounted) from the DOM.

Used to clean up resources like:

Timers (setTimeout / setInterval)

Event listeners (window.addEventListener)

Subscriptions (WebSocket, API listeners)

Helps prevent memory leaks.  
  
**Fragments**

A Fragment in React is a wrapper that lets you group multiple elements without adding an extra tag (like <div>) to the DOM.

**Component vs PureComponent in React**

🔹 React.Component

Base class for all class components.

Always re-renders when its parent re-renders, even if props/state didn’t change.

No performance optimization by default.

**React.PureComponent**

A special version of Component.

It does a shallow comparison of props and state in shouldComponentUpdate.

Re-renders only if props or state actually changed.

if there is no differnce the component is not re-rendere - Performance boost

Helps avoid unnecessary re-renders → better performance.

**Normal Component**

Always re-renders when parent re-renders

Even if props/state are the same

👉 Can cause unnecessary renders (slower performance)

🔹 **PureComponent**

Does a shallow comparison of props and state before re-rendering

If nothing changed → it skips re-render

This gives a performance boost by avoiding useless work

**Shallow comparison** means React checks only the first level of props/state values (not nested/deep values) to decide if something has changed.

Primitive values (string, number, boolean) → compared by value

Objects & arrays → compared by reference (not by content)

**Memo Component**

React.memo is a higher-order component that wraps a functional component and makes it “pure”:

It prevents the component from re-rendering if its props haven’t changed (shallow comparison).

Works like PureComponent, but for function components.

**REF**

A Ref (short for reference) in React is a way to directly access a DOM element or a React component instance without using state.

Useful for manually manipulating DOM elements.

Useful for storing mutable values that do not trigger re-renders.

**What are React Portals?**

A React Portal allows you to render a component or element outside the main React app DOM tree.

Normally, React components render inside the root <div id="root"></div>.

With portals, you can render somewhere else in the DOM, such as for modals, tooltips, or popups.

The component remains in the React tree (so events, state, and context still work), but visually it’s placed elsewhere in the DOM.

**What is an Error Boundary?**

An Error Boundary is a special React component that catches JavaScript errors anywhere in its child component tree.

Instead of breaking the entire React app, it:

Catches the error.

Displays a fallback UI.

Prevents the whole app from crashing.

**Why use HOC?**

Code reuse: Share common functionality across multiple components

**Render Props Definition**

A render prop is a function passed as a prop to a component, which lets that component decide what UI to render.

👉 In short:

“It’s a way to share logic between components by passing a function that returns JSX.”

**React Context** — another way to share data between components, without having to pass props manually at every level

Context provides a way to share data globally in a React app..

**What is a Hook in React?**

A Hook is a special function that lets you “hook into” React features like state and lifecycle methods inside functional components.

Before Hooks, only class components could use state or lifecycle methods.

Hooks let functional components have state, side-effects, context, and more.

🔹 Key Points

Hooks always start with use (e.g., useState, useEffect).

Hooks cannot be called inside loops, conditions, or nested functions — they must be at the top level of a functional component.

Hooks allow you to reuse stateful logic without changing your component hierarchy.

🔹 Common Hooks

Hook Purpose

useState Add state to functional components

useEffect Handle side effects like fetching data or DOM updates

useContext Access context values

useReducer Manage complex state logic

useRef Reference DOM elements or persist values

useMemo Memoize expensive computations

**useCallback Memoize functions to avoid unnecessary re-renders  
  
  
What is useState?**

useState is a React Hook.

It lets you add state (data that can change) to a functional component.

Before Hooks, only class components could have state.  
  
  
  
**What is useEffect?**

useEffect is a React Hook.

It lets you run side effects in your component.

👉 Side effects = anything that happens outside React’s rendering process, like:

Fetching data from an API

Updating the DOM manually

Setting up timers (setInterval, setTimeout)

Subscribing/unsubscribing (e.g., WebSocket, events)  
  
**in class components, we have these lifecycle methods:**

componentDidMount → runs once when the component is mounted (inserted into the DOM).

componentDidUpdate → runs when the component updates (state/props change).

componentWillUnmount → runs before the component is removed from the DOM (cleanup)

**useReducer.**

Usereducer is a hook that for used for state management in React.

useRenducer is relate to reducer fuctions

userRenducer(reducer,initialState)

reducer(currentState,action)

**Why use useReducer with useContext?**  
  
To manage global state (like Redux) without installing extra libraries.

useReducer → handles state updates using actions.

useContext → allows any component to access that state without prop-drilling.

**Side-by-Side Comparison**

**Feature useState ✅ useReducer ✅**

Simplicity Easier, less code More boilerplate (reducer, actions)

State Shape Best for primitive/simple state Best for complex state (objects, arrays)

Update Logic Inline with setState Centralized in reducer (cleaner for big logic)

Readability Cleaner for small updates Cleaner for multiple action types

When to Use Local/simple states Complex, shared, or interdependent states

**useCallbac**k prevents unnecessary function re-creations.

Optimization – Prevents unnecessary re-creation of functions on every render.

Memoization – Stores the function reference until dependencies change.

Performance – Reduces re-renders of child components that rely on function props.

Stability – Keeps function references stable across renders.

Reusability – Allows the same function to be reused without breaking React.memo components.

**Without useCallback, every render creates a new function, causing unnecessary child re-renders.**

**What useMemo does**

**useMemo memoizes (caches) the result of a calculation.**

**It recomputes the value only when its dependencies change.**

**Helps avoid expensive calculations on every render.**

**3️⃣ Key Difference**

**Feature useCallback useMemo**

**Returns Function Value (result of function)**

**Purpose Memoize a function Memoize a computed value**

**Use case Pass function to memoized children Avoid expensive recalculation / stable value**

**What is useRef?**

**useRef is a hook that lets you persist values between renders without causing a re-render.**

**It can also access DOM elements directly.  
Controlled component → The input’s value is controlled by React state.**

**Uncontrolled component → The input’s value is controlled by the DOM itself (not React).**

**A controlled component in React is a form element (like <input>, <textarea>, <select>) whose value is managed by React state instead of the browser.  
  
  
  
  
Advantages of HooK**

**Functional components can be used for all React feaatures  
2 NO this problems**

**3 Reusing stateful logic**

**5 reduced use of hoc(Higher order component)  
  
what happen used fuction in ref  
  
Here, React calls your function whenever the element mounts/updates:**

**When mounted: ref(el) → gets the DOM node (<span>...</span>).**

**When unmounted: ref(null) → React clears it.**

**Redux**

**Helps manage state in javascript application  
  
  
  
  
🌐 Definition: Router**

**Routing in React means showing different components (or pages) based on the URL path — without reloading the page.**

**It helps create a Single Page Application (SPA) where navigation feels smooth and fast.**

**🧩 Types of Routing in React**

**BrowserRouter**

**Uses the browser’s history API.**

**Looks like normal URLs (e.g., /home, /about).**

**Example:**

**import { BrowserRouter } from 'react-router-dom';**

**HashRouter**

**Uses the URL hash (#) part for routes.**

**Example URL: example.com/#/home**

**Useful when you don’t have server-side routing setup.**

**MemoryRouter**

**Keeps the URL changes in memory (doesn’t affect the browser address bar).**

**Commonly used in testing or React Native.**

**⚙️ Main Components**

**Component Purpose**

**<BrowserRouter> Wraps the app to enable routing**

**<Routes> Holds all route definitions**

**<Route> Defines path and component to render**

**<Link> Creates clickable navigation links**

**useNavigate() Navigate programmatically (e.g., after a button click)**

**. What is React Router?**

**React Router is a library for routing in React apps. It allows you to:**

**Navigate between different components/pages**

**Keep the UI in sync with the URL**

**Build Single Page Applications (SPAs) without full page reloads**

**2. Key Components**

**Component Usage**

**<BrowserRouter> Wraps your app and enables React Router (uses HTML5 history API)**

**<Routes> Container for all your routes (replaces Switch in v6)**

**<Route> Defines a route with path and element**

**<Link> Navigate between routes without page reload**

**<NavLink> Like Link, but can apply “active” class when route matches**

**useParams Hook to access URL parameters (/user/:id)**

**useSearchParams Hook to access query strings (?class=5)**

**useNavigate Hook to programmatically navigate routes**

**1. Absolute Path**

Starts from the root of your app (/)

Always points to the same location, no matter the current URL

Most common for main navigation

**2. Relative Path**

Calculated relative to the current route

Does not start with /

Useful for nested routes or sub-pages  
  
  
  
  
  
  
  
  
  
useParams

Gets dynamic parameters from the URL.

Returns an object of all route parameters.

Example: For route /users/:userId, useParams() gives { userId: "2" }.

2. useNavigate

Returns a function to programmatically navigate between routes.

Can navigate using path strings (e.g., navigate('/users')) or relative paths.

Can also pass state or replace history.

3. useLocation

Returns the curret location object.

Includes:

pathname → current URL path

search → query string

state → data passed via navigation (Link or navigate)

Useful for conditionally rendering based on where you came from or what data was passed.

Descendant Routes

Routes that are nested deeper inside another route (children of nested routes).

Essentially, grandchildren or further nested routes.

Follows the same principle as nested routes but can be multiple levels deep.

What are Nested Routes?

Nested routes are routes defined inside another route.

The parent route renders its child routes inside an <Outlet /> placeholder.

Useful for layouts like dashboard tabs, user details pages, or multi-step forms.  
  
What is an Index Route?

An index route is a child route that renders by default when the parent route matches but no specific child path is provided.

Think of it as the “default tab” inside a parent route.