**<https://github.com/JhanaR/my_learnReact.git>**

**If need refer below links too**[**https://github.com/JhanaR/CloneMalini.git**](https://github.com/JhanaR/CloneMalini.git)[**https://github.com/JhanaR/UiClass\_React.js-Prereq.git**](https://github.com/JhanaR/UiClass_React.js-Prereq.git)[**https://github.com/JhanaR/React\_MaliniClass.git**](https://github.com/JhanaR/React_MaliniClass.git)

**What is Library?**

A collection of pre-written code that programmers can use to perform specific tasks or add functionality to a program. Libraries are often written in a specific programming language and are designed to be reusable**.**

**What is React?**

React is an **open-source front-end JavaScript library** also called as **component based** front end library that is used for building user interfaces, especially for single-page applications. It is used for handling **VIEW LAYER** for web and mobile applications so that its light weight application.

React helps you manage these components efficiently and keeps your UI in sync with your data.

**What are the major features of React or Purpose of React? Or What are the building blocks of React? Explain with real-time usage.**

The main building blocks of React are:

1. Components

Web consists of many parts; each part is a component.

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML.

Each components have states (interface changes) and props are objects (sharing the access with the components, its immutable we can get the data but unable to modify).

1. JSX

JSX is a XML-like syntax extension to ECMAScript

**JSX** stands for **JavaScript XML**.  
It allows you to **write HTML-like code inside JavaScript** — which makes writing UI components easy and readable.function MyFirstComponent() {

|  |
| --- |
| return (  < >  <h1> My First React Component! </h1>  </ >  ) |

1. Prop

Props allow you to pass data and behavior (via functions) from a parent component to a child component.

1. State

State is a built-in object in React used to store dynamic data that can change over time and affect the UI rendering.

When state changes, the component re-renders automatically with the new data.

1. Hooks

**Hooks** are special functions in React that let you **use state, lifecycle methods, and other React features** inside **functional components**.

1. Virtual DOM

The major features of React are:

* It uses **VirtualDOM** instead of RealDOM considering that RealDOM manipulations are expensive.

**what is DOM?**

Document object model which represents the web page as a treelike structure which allows JavaScript to dynamically access and manipulate the content and structure of web page, we can achieve everything using Real DOM but the performance will be low.

**What is the Virtual DOM and how does it work in React?**

The Virtual DOM is a performance optimization technique in React.  
It creates a virtual copy of the real DOM, and when changes occur, React uses **Reconciliation** and the **Diffing Algorithm** to identify what needs updating.  
The **React Fiber architecture** handles rendering efficiently, breaking it into manageable chunks.  
This leads to faster updates and smoother UIs, even in large-scale applications.

**Reconciliation**: The process of comparing the **previous Virtual DOM** with the **new Virtual DOM** to determine **what has changed**.

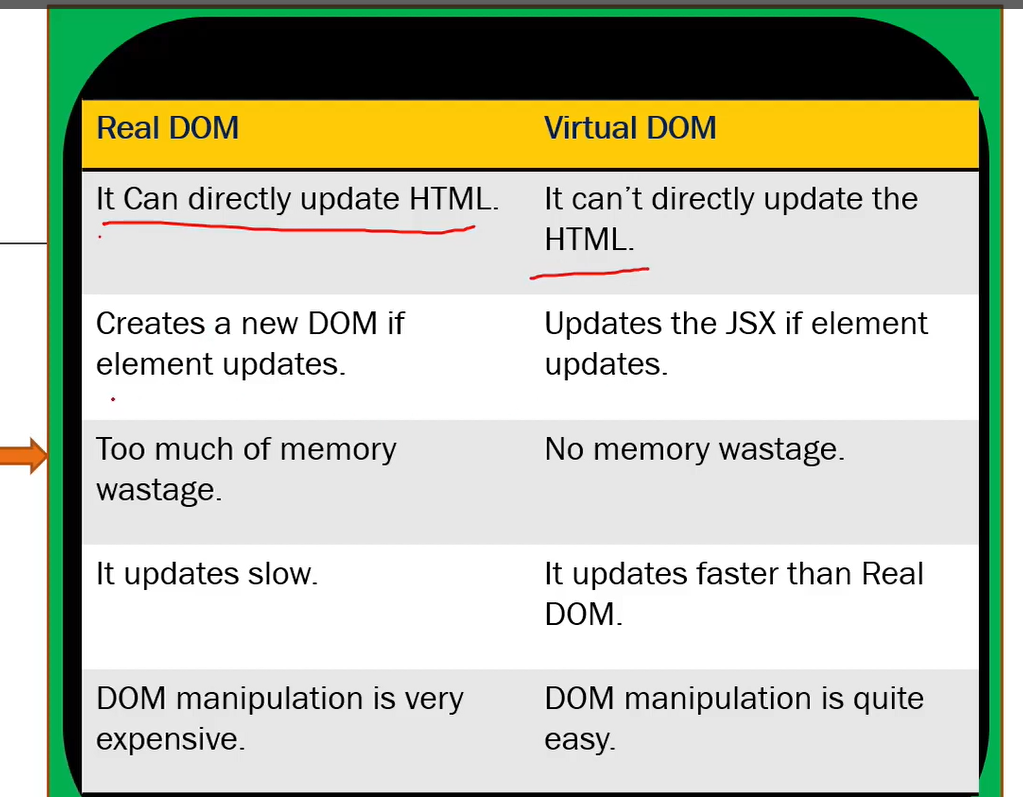
Based on this, React **updates only the changed parts** in the real DOM.

**Diffing Algorithm:** Compares the old and new Virtual DOM trees, the **minimum number of operations** needed to update the real DOM.

**React Fiber architecture**: In large UI apps, React Fiber ensures smoother UI by **pausing, resuming, and reusing rendering work** efficiently.

* Supports **server-side rendering**.
* Follows **Unidirectional** data flow or data binding.
* Uses **reusable/composable** UI components to develop the view(maintaining and scalable approach)

**Difference between DOM and virtual DOM?**



**Can you explain how React’s rendering process works and how the Virtual DOM helps with performance?**

Step-by-Step: React Rendering Process

1. State/Props Change

When component state or props change, React re-renders that component (and potentially its children).

1. Reconciliation (Virtual DOM diffing)

React builds a new Virtual DOM — a lightweight JavaScript representation of the UI.

It compares the new Virtual DOM with the previous Virtual DOM using the Diffing Algorithm.

Only the elements that changed are identified.

1. Batching Updates

React batches multiple state updates together for efficiency instead of re-rendering after each small change.

1. Commit Phase (Real DOM update)

React updates only the changed parts in the real DOM (not the whole page) using minimal operations.

**How the Virtual DOM Improves Performance**

* Direct DOM manipulation is slow because the browser recalculates layout, styles, and paints.
* The **Virtual DOM** keeps these calculations in memory, reducing costly direct DOM writes.
* By updating only the changed parts, React **avoids full page reloads** and makes updates faster.

**Advantages of React.?**

* Used to build sing page Application (SPA) by using components. (React follows component-based architecture which allows developer to use reusable components which can be used throughout the application and that makes its heavy speed.
* React.js is cross flatform and open-source JavaScript framework library, free to use. React is used to develop mobile and applications.
* Light weight and very fast virtual DOM. (Very fast due to Virtual dom)
* Large Community and Ecosystem.
* Testing is easy.

**What is SINGLE PAGE APPLICATION.?**

The web page which consists many components, if there is any change/ update in any component, only that content in the component is dynamically updated instead of refreshing the whole page.

**What is JSX?**

**In how many ways you can create Component ?**

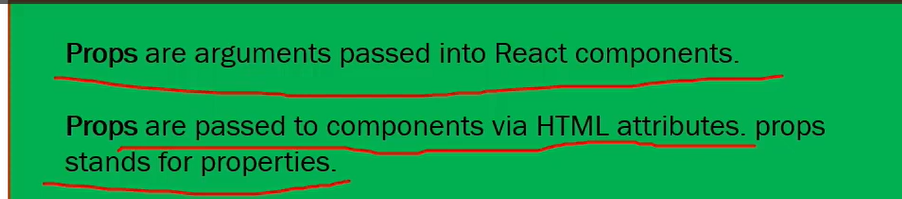
Components come in two types, Class components and Function components

**What are the main differences between class components and functional components?**

Class components use lifecycle methods (ComponentDidMount,componentDidUpdate etc.) and manage state with this.state, whereas functional components are simpler, using Hooks like useState, useEffect to manage state and side effects.

Real-time Note:  
Since React 16.8, Hooks made functional components more powerful, and now most codebases—including ours—prefer function components for cleaner, testable code.

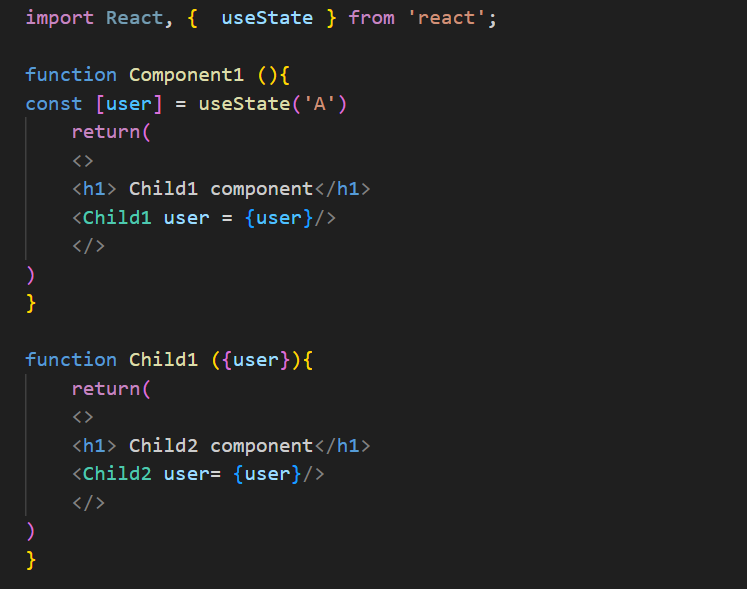
**What is Props ?**



**Why we use Props ?**

Props allow you to pass data and behavior (via functions) from a parent component to a child component.

**Hint:** If you want to access a property in a child object even when the parent doesn't have that property, it will not throw an error.

****

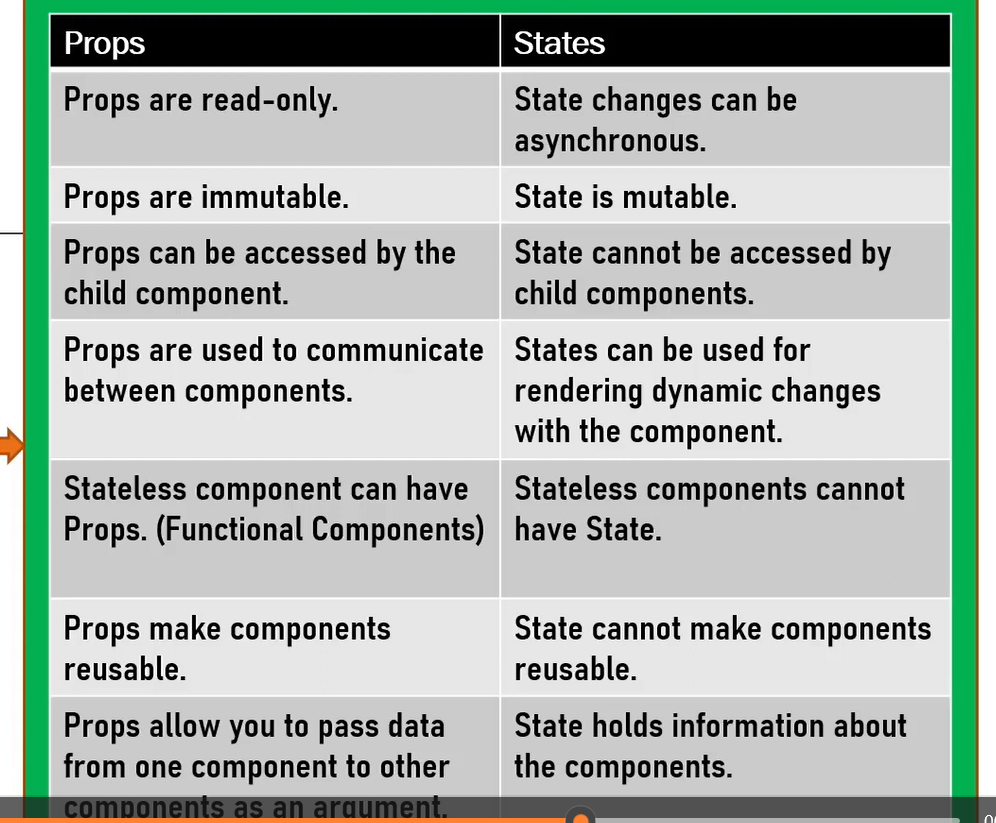
**Can you explain the difference between state and props? OR How is state different from props in React?**

**Props** and **state** are both used to manage data in React, but they serve different purposes.

🔹 **Props** (short for "properties") are used to pass data **from a parent component to a child**. They are **read-only** from the child’s perspective, ensuring unidirectional data flow. This promotes component reusability and predictable behaviour.

🔹 **State**, on the other hand, is a **local, mutable data container** managed within the component itself using the useState (or useReducer) hook. When state updates, it triggers a **re-render**, allowing dynamic UI updates based on user interactions or side effects.

In short, **props are for configuration**, while **state is for internal data that may change** over the component’s lifecycle.



**How do you pass data from parent to child and vice versa?**

**What is key in React?**

The key is a special prop used by React to identify which items in a list have changed, been added, or removed.

It helps React optimize rendering by reusing elements instead of re-creating them from scratch.

**What is data binding in React? Explain the types with examples.**

**Data binding** in React refers to the connection between **UI elements (like inputs, text)** and the component’s **data/state**.  
It determines how changes in the **state** reflect in the UI and how **user input** updates the state.

React supports mainly **one-way data binding from parent to child using props**, and it allows you to **simulate two-way binding** in controlled components.

You manually **bind the input value to state**, and **update the state on input change**.

**how can you create reference for element in functional Component ?**

**What is the difference between an Element and a Component?**

**What is the diff between functional and Class components.**

**What are React hooks? Name a few.**

React Hooks are special functions introduced in React 16.8 that let you use state and other React features in functional components without writing class components.

**They allow you to:**

* Manage state (useState)
* Handle side effects (useEffect)
* Access context (useContext)
* Work with refs (useRef)
* Optimize performance (useMemo, useCallback)

**Common Built-in Hooks**

1. useState – For managing local component state.
2. useEffect – For handling side effects (API calls, subscriptions, DOM updates).
3. useContext – For accessing React Context values.
4. useRef – For persisting values across renders without triggering re-renders.
5. useMemo – For memoizing values to avoid expensive recalculations.
6. useCallback – For memoizing functions to prevent unnecessary re-creations.

React uses a Virtual DOM — a lightweight copy of the real DOM.

When state/props change, React:

1. Re-renders the component in Virtual DOM.
2. Diffs the new Virtual DOM with the previous one using a process called Reconciliation.
3. It then batches minimal updates to the real DOM, which improves performance.

React also uses Fiber architecture (since v16) to break rendering into units of work, allowing interruptible rendering for smooth UX.

**What is state and How to Update State ?**

**state** is an object that holds or allows you to store property values that belong to the component that may change over the lifetime of a component. It is essentially the data that drives the behaviour and appearance of your component. When the state of a component changes, React re-renders the component to reflect the updated state.

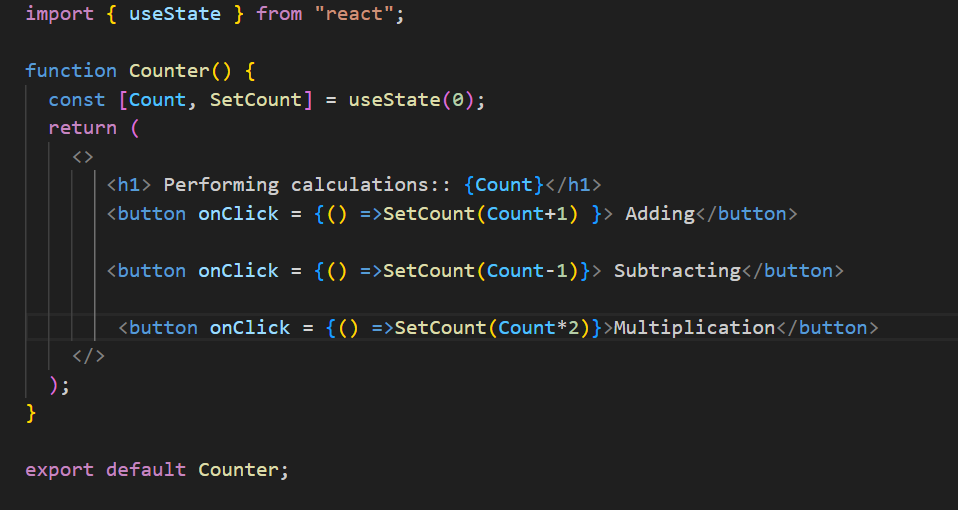
- Each component can have its own state.

- The state is **mutable**, meaning it can change over time, usually in response to user actions or network responses.

- State is used in **class components as this.state** and in **functional components using useState** hook.

- The **useState** hook returns an array with two elements: the current state value and a function to update it.

**- Example: Build a simple counter app using React hooks.**

**- Here, `count` is the current state, and `setCount` is the function to update the state.**

State management is a core concept in React, enabling you to create interactive and dynamic user interfaces.

**What happens when setState() is called ?**

React initiates an efficient, batched update process that eventually leads to the component being re-rendered with the updated state, ensuring that the UI stays in sync with the underlying data.

**What happens in React when you click a button that updates state?**

When I click a button that updates the state using useState, react re-renders the component with the new state value. Under the hood, React uses the Virtual DOM to compare the previous state and the new one. It then updates only the changed parts in the real DOM using a diffing algorithm, improving performance.

**What are React Hooks? Which ones do you use most often?**

There are two main rules of using Hooks in React:

1. Only call Hooks at the top level.  
   Don’t call them inside loops, conditions, or nested functions. This ensures Hooks are called in the same order every time a component renders.
2. Only call Hooks from React functions.  
   You can use Hooks inside:
   * Functional components
   * Custom Hooks  
     But not in regular JS functions or class components.

Hooks are functions introduced in React 16.8 that let you use state and lifecycle methods in functional components.

* Simplify state management in functional components with useState.
* Perform side effects with useEffect.
* Access context values with useContext.
* Manage complex state logic with useReducer.
* Optimize performance with useMemo and useCallback.
* Access and persist values or DOM elements with useRef.

I frequently use:

* useState for managing local state
* useEffect for side effects (like fetching data)
* useRef for storing mutable values or DOM refs
* useContext for global/shared state

**What is useState?**

useState is a **React Hook** that allows functional components to have **local state** — enabling them to hold and update values that trigger re-renders.

**What triggers a re-render?**

Calling setState (e.g., setCount) triggers a re-render with updated UI.

**Purpose of useState?**

To manage dynamic data inside functional components.

Trigger re-renders when the state changes.

Encapsulate UI behavior like toggling, counters, inputs, modals, etc.

**When to Use useState?**

You need to track a value over time (e.g., input field, counter, toggle).

1. A value needs to re-render the UI when changed.
2. You want local component state (not global or shared state).
3. The state doesn't need persistence across routes/pages or deep component trees.

**Why is key concept in React important?**

Keys help React identify which items have changed in a list, making updates more efficient.

**What is useEffect or how useEffect will maintain 3 lifecycle method ? OR explain useEffect — how does dependency array work?**

useEffect is a React Hook used to handle side effects in functional components — like fetching data, setting up subscriptions, or manually manipulating the DOM.  
It replaces the old lifecycle methods from class components:

* Acts like componentDidMount when the dependency array is empty.

useEffect(() => {}, [])

Like componentDidMount in class components.

Used for **initial data fetch**, API calls, subscriptions, or timers, etc.

**Trigger count:** 1 (after first render)

* Acts like componentDidUpdate when specific dependencies are listed.

useEffect(() => {}, [someValue])

**Runs after first render**, and then **whenever someValue changes**

Like componentDidUpdate (for that value).

Acts like componentWillUnmount

useEffect(() => {})

**Runs after every render** (initial + all updates)

Like combining componentDidMount and componentDidUpdate

**What is useEffect used for?**

useEffect runs side effects like API calls, subscriptions, or timers.  
You can control when it runs using the **dependency array**:

[] → runs only once (like componentDidMount)

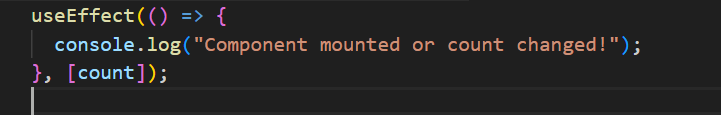
[dep] → runs when dep changes(like ComponentDidUpdate)

no array → runs on every render( like componentWillUnmount)

**return () => clearTimeout(timer);**

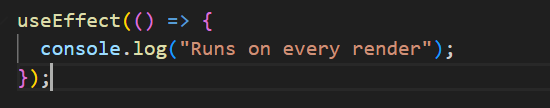
This is the **cleanup function** of useEffect. It runs before the next effect is applied.

The cleanup function in useEffect is used to remove things like event listeners, timers, or subscriptions to prevent memory leaks. It runs when the component unmounts or before the effect re-runsz



**What hook runs on every render by default?**

The useEffect() hook runs on every render **by default**, unless we provide a dependency array.  
If no second parameter is given, React assumes it should run the effect **after every re-render**.



**What’s the second parameter in useEffect?**

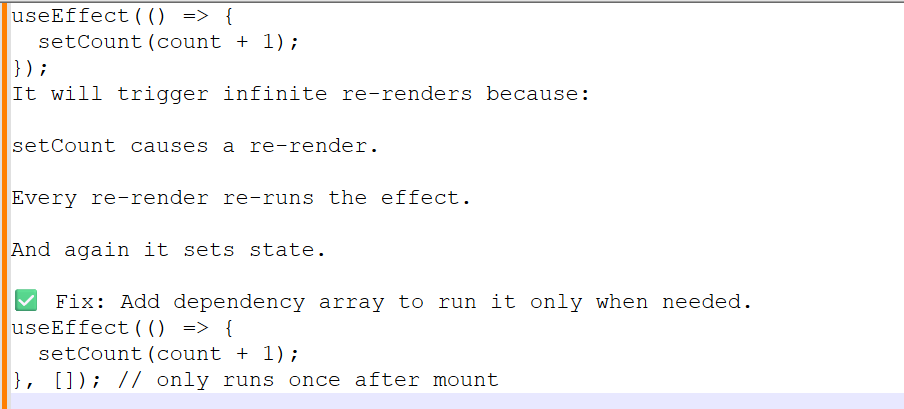
The second parameter is the **dependency array**. It tells React when to re-run the effect:

* [] → run once on mount (like componentDidMount)
* [value] → run only when value changes
* No array → run on every render

**What happens when you call setState inside useEffect without a dependency array?**

If you call setState inside a useEffect without passing a dependency array.

stale



**How do you pass data from parent to child and vice versa?**

**Have you used Redux or Context API? Explain with example OR What is the useContext hook and when would you use it?**

The useContext hook is used to access the value of a React Context directly inside a functional component, without the need for prop drilling.

It’s ideal for sharing global state like:

* Themes (e.g., dark/light mode)
* Authentication status
* User preferences or language settings

I’ve used useContext in large UI apps to manage layout preferences like dark mode or sidebar visibility — avoiding unnecessary Redux setup or passing props deeply through multiple components.

For more complex shared state, I’ve combined useContext with useReducer to manage state transitions cleanly while still avoiding external state libraries.

**What is Context Api and Why do we use Context API ?**

* Context API is a way to pass data through the component tree without having to pass props down manually at every level.
* Use Case: It's primarily used to avoid prop drilling and to manage global state that needs to be accessed by multiple components.
* Key Components: createContext, Provider, Consumer (or useContext hook).
* Advantages: Simplifies state management for certain use cases, reduces dependency on parent components, and provides a more maintainable and scalable solution for passing data across components.

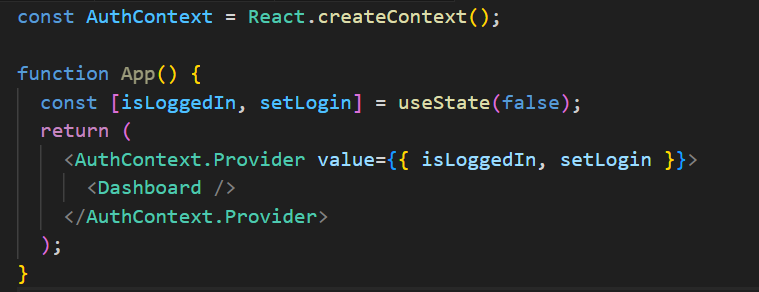
**Which kind problem will be solved using Context API ?**

The Context API is a powerful tool for solving various problems related to state management and data sharing in React applications, including:

* Avoiding prop drilling.
* Managing global state.
* Decoupling components.
* Sharing configuration or constants.
* Handling user preferences or localization.

**How do you manage global state in React OR How do you manage global state in your application?**

For simple shared state, I use the **Context API**. If the app grows complex, I use **Redux Toolkit** or **Zustand**.  
Context is great for theme, auth, or language settings.



**What is use Reducer and for what purpose it is used for?**

**What is useMemo and for what purpose it is used for and when should you use it?**

used to memoize the result of a computation, so it only re-computes when its dependencies change.  
It helps improve performance by avoiding expensive calculations on every render.

Prevents unnecessary recalculation (e.g., doubleCount)

**What is useCallback?**

Prevents function prop from being recreated (e.g., onGreet)

useCallback is used to **memoize functions**, ensuring they're only recreated when their **dependencies change**.

This is particularly useful when passing callbacks to **memoized child components** (like those wrapped with React.memo) to **prevent unnecessary re-renders** caused by changing function references.

It helps improve performance in component trees where rendering logic is heavy or deeply nested.

In a large-scale UI, I’ve used useCallback to stabilize handler functions that were passed into child dropdowns or reusable lists, avoiding performance issues from repeated renders during user interactions.

**What is React.memo and useCallback?**

* useCallback(fn, deps) returns a **memoized function avoid rendering a function.**
* useMemo(() => compute, deps) returns a **memoized value—avoid rerunning expensive calculations.**

Both are used to avoid unnecessary recalculations or re-creations.

**What is HOC (Higher-Order Component) in React?**

A **Higher-Order Component (HOC)** in React is an advanced pattern used to **enhance the behaviour of a component** by wrapping it with additional logic. Technically, an HOC is a **function that takes a component as input and returns a new enhanced component** with additional features or props.

**When to Use HOC?**

Use HOCs when you want to reuse logic across multiple components without duplicating code.

Examples:

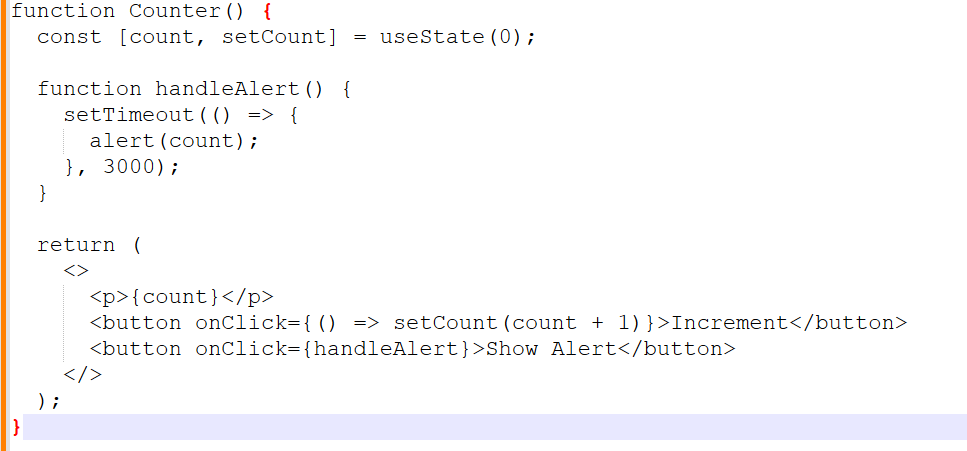
* Authentication/Authorization
* Logging
* Role-based access
* Conditional rendering
* Injecting common props or behavior

**Where to Use HOC?**

* Wrap UI components that need common logic
* Place them in a separate file (withAuth.js, withLogger.js, etc.)
* Use them to wrap components in your routing or layout structure

**What are common issues with stale closures in React? Explain with an example?**

A **stale closure** happens when a function holds on to an outdated value of a variable from an earlier render.



If you click **Increment** once (count becomes 1), then click **Show Alert**, after 3 seconds you'll still see 0.

Because the closure **captures old value of count**

**What is useRef?**

Using useRef (to avoid stale closures in setTimeout, setInterval, or event handlers)

It doesn’t trigger re-render.

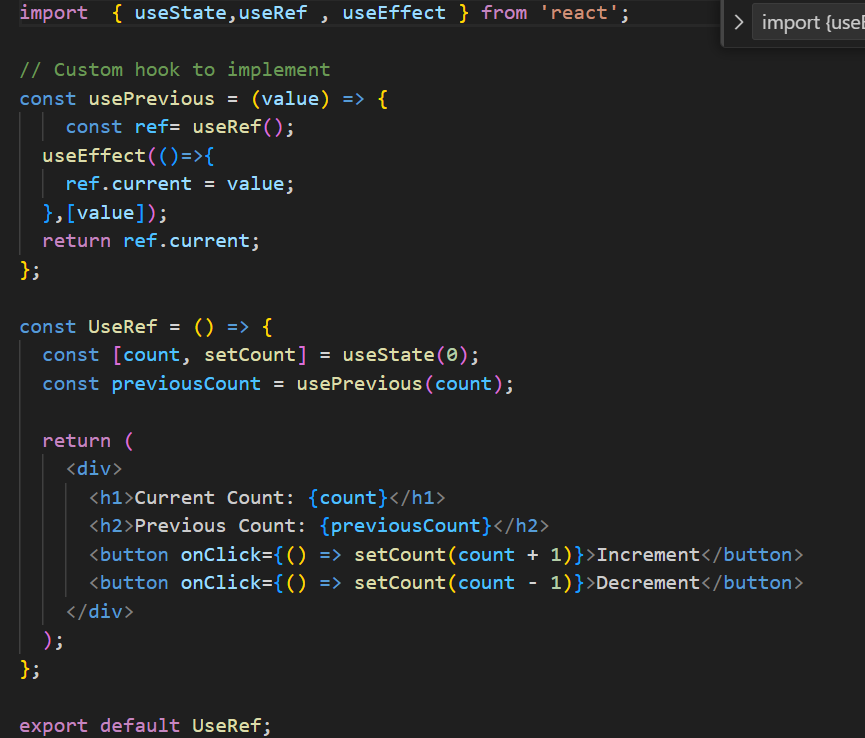
**OR**

useRef is used to create a mutable reference that persists across renders without causing re-renders.

Common use cases:

* Accessing DOM elements (e.g., focusing an input)
* Storing previous values
* Holding timers or interval IDs

It’s like a container that doesn’t trigger a re-render when updated.

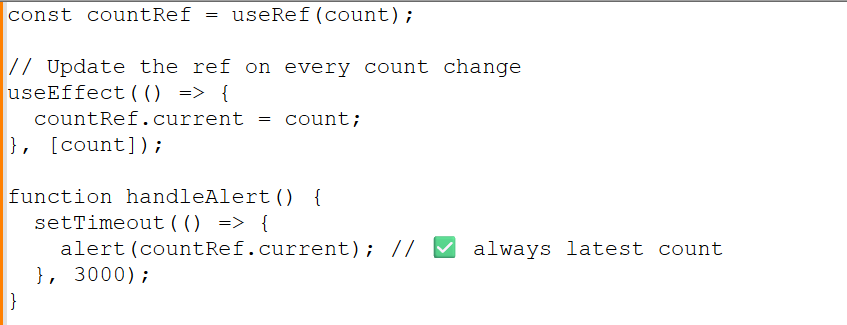
****

**Compare useRef vs prevCount?**

**useRef**

To access the latest state/value inside:

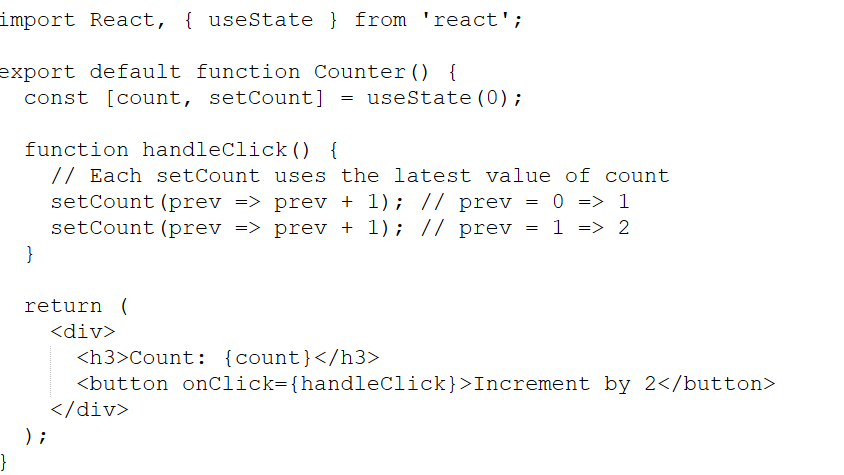
* setTimeout, setInterval, or event listeners
* Without causing re-renders

****

**Prev**

When updating state based on previous value, especially in:

* Loops
* Multiple state updates in a row

****

HINT:

You want to do setState twice based on old state **setCount(prev => prev + 1)**

You want to read latest state in setTimeout **useRef + useEffect**

You want to update some state but not re-render **useRef**

You want to persist value between renders **useRef**

**How do you handle API calls in React?**

* I use useEffect() to fetch data from APIs. I also handle loading and error states for a better user experience.

**Difference between React Router v5 and v6**

**How do you handle error boundaries in React?**

Only class components can be used as error boundaries. You wrap components inside an Error Boundary to catch rendering errors and prevent breaking the entire app.

Error boundaries catch JavaScript errors in child components during rendering, in lifecycle methods, and in constructors. They are implemented using class components with componentDidCatch and getDerivedStateFromError.  
  
As a lead, I recommend wrapping high-risk areas like remote data UIs and using logging services like Sentry.

**Code — Build a To-Do App using Hooks**

**What is server-side rendering and Client-side rendering?**

**What are controlled and uncontrolled components?**

**Controlled**

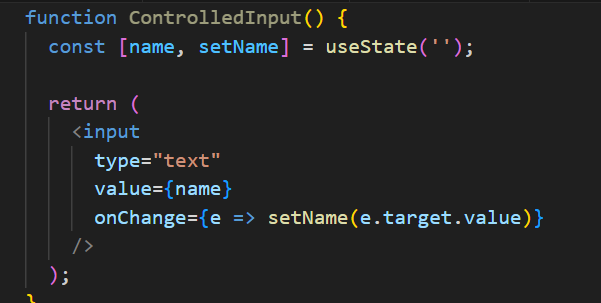
React **fully controls** the form input using useState.

The value of the input is **bound to a state variable**.

Input changes are **tracked with onChange()** and update the state.

The input elements receive their current value from the state and have their value updated through a callbackfunction.

Controlled components are preferred in React because they make the data flow predictable and easy to validate



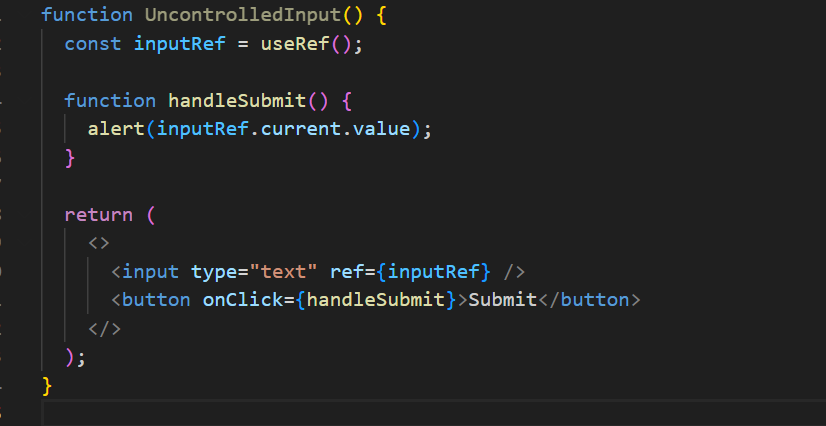
**In uncontrolled components**

React **does not control** the input state.

Uses a **ref** to access the DOM element and get its value.

Input state is **maintained by the DOM**, not React.

Uncontrolled components may be used for quick forms, performance-sensitive parts, or third-party libraries.



**How do you optimize performance in React?**

Use React.memo to prevent unnecessary renders

Split code using **lazy loading** and Suspense

Use proper keys in lists

Avoid anonymous functions in render

**How does React handle reconciliation?**  
React uses the reconciliation algorithm to compare the current Virtual DOM with the previous one and applies minimal changes to the real DOM using keys and component types.

**How to prevent re-render in a child component?**

Use React.memo() to memoize the child component. It prevents re-render unless props actually change.

Also Avoid

Passing **new inline functions or objects** as props (unless wrapped in useCallback or useMemo)

Changing parent state unnecessarily

**How to persist state between refreshes?**

Use **localStorage**, **sessionStorage**, or even **IndexedDB** to persist state across refreshes.

**How to create Parent Child Relation ?**

Creating a parent-child relationship in React involves nesting components, where one component (the parent) renders another component (the child) and often passes data or functions as props.

**What is Application Flow ?**

App Initialization: The React app starts, and the root component (<App />) is rendered.

User Logs In: The user inputs credentials into a form and submits it.

State Update: The login form triggers an API call to validate the user. Upon success, the user’s data is stored in the global state.

Navigation: The user is redirected to the dashboard, where the relevant data is displayed.

Data Fetching: The dashboard component fetches additional data (e.g., user’s recent activity) and updates its state accordingly.

User Interaction: The user interacts with various elements on the dashboard, triggering updates in the UI and possibly making further API calls.

Logout: The user logs out, triggering a cleanup of user data from the state and redirecting back to the login page.

**What is diff between render() in React and render() in ReactDOM ?**

**render() in React**

* The render() method is a lifecycle method in class components that defines what the component's UI should look like.
* Purpose: It returns the JSX that represents the component's structure. This JSX is then converted to actual DOM elements by React.
* render() function from the ReactDOM library is used to render a React element (or component) into a specified DOM container on the page.

**render() in ReactDOM ?**

* Purpose: It serves as the entry point to rendering a React application or component tree into the DOM.

**In how many ways we can pass the data from parent to child ?**

* Props: The most straightforward and common way to pass data.
* Callback Functions: Allows the child to communicate back to the parent.
* Context API: Useful for avoiding prop drilling in deeply nested components.
* State Management Libraries: Ideal for complex, large-scale applications.
* Render Props: A pattern for passing data in more dynamic scenarios.
* Higher-Order Components (HOCs): A pattern for sharing logic between components.
* Composition: Passing JSX or components through the children prop for more flexible component design.

**In how many ways you can manage state in React Applicatiion ?**

* Local State (useState, setState): Best for managing state within individual components.
* Derived State (useMemo, useCallback): Useful for optimizing performance by memoizing expensive calculations or functions.
* Lifting State Up: Essential for sharing state between sibling components.
* Context API: Ideal for sharing state across a component tree without prop drilling.
* Redux: A powerful, widely-used global state management solution, especially in large applications.

**How do you manage state in a complex React application?**

For local state, I use useState or useReducer.  
If multiple components share state (e.g., filters or user session), I prefer Context API for light use-cases.  
For large-scale or asynchronous flows, I use Redux Toolkit — it's clean, maintainable, and handles middleware like createAsyncThunk.  
As a lead, I ensure we don't overuse global state and keep components pure and reusable.

**What are controlled vs uncontrolled components in React?.**

A controlled component has its input value bound to React state via useState.  
An uncontrolled component uses ref to access the DOM value directly.  
Controlled inputs give you **more control over validation and user feedback**, but for file uploads or 3rd-party plugins, uncontrolled may be better.  
I guide my team to **stick to controlled inputs** for consistency in enterprise apps.

**Explain a challenging UI you built and how you approached it.**

**How do you ensure performance optimization in your React app?**

**How do you handle errors and loading states in your app?**

**Have you worked with any UI libraries like Material UI or Ant Design?**

**How do you collaborate with backend developers in your project?**

**What tools do you use for testing your React app?**

**Have you done deployment of your React app? How?**

**What is axios ?**

Axios is a powerful and flexible library for handling HTTP requests in JavaScript applications. Its features, such as promise-based handling, support for various HTTP methods, interceptors, and built-in error handling, make it a popular choice for making network requests and integrating with APIs in both browser and Node.js environments

**Anything alternate to axios ?**

Fetch API: Native, modern, and flexible; built into browsers.

jQuery AJAX: Part of jQuery; suitable if you're already using jQuery.

Superagent: Lightweight and flexible with a simple API.

Got: Advanced features for Node.js environments.

Ky: A small and elegant wrapper around Fetch.

node-fetch: Provides Fetch functionality for Node.js.

Apollo Client, Relay: Specialized for GraphQL queries and state management.

**How to implement the routing the react application ?**

React Router is the most commonly used library for routing in React applications.

Basic Setup: Install React Router, create components, and define routes using BrowserRouter, Routes, and Route.

Navigation: Use Link instead of traditional <a> tags for navigation.

Nested Routes: Define nested routes using Outlet.

404 Handling: Add a fallback route for handling non-existent paths.

**What is State Mangement?**

In React, state management is about handling UI, shared, and server data efficiently.  
For local state, I use useState or useReducer.  
For shared/global state, I use useContext, or Redux if the app is large.  
For API data, I prefer using React Query to manage caching, loading, and error states easily.  
In real projects, I’ve used useContext with useReducer for lightweight state management in layout controls, and Redux for complex apps like e-commerce dashboards.

**How do you optimize performance in a large-scale React app?**

Performance optimization involves:

* Code-splitting using dynamic import() and React.lazy
* Memoization with React.memo, useMemo, and useCallback
* Avoiding unnecessary re-renders using pure components
* Virtualizing long lists using react-window
* Debouncing and throttling user inputs
* Implementing lazy loading of routes and images
* Using Profiler API to analyze performance

**Why hooks were introduced?**

**Hooks were introduced to:**

* Simplify state management and side effects in functional components.
* Facilitate the reuse of stateful logic through custom hooks.
* Improve the readability and maintainability of components.
* Align with functional programming principles.
* Avoid issues associated with complex component hierarchies and wrapper hell.

**What is the use of useContext() ?**

The useContext() hook in React is used to access the value of a context in functional components. Context is a way to pass data through the component tree without having to pass props down manually at every level.

**What is lazy loading and How do you implement lazy loading in React?**

**Lazyloading?**

Lazy loading in React helps optimize performance by loading components or resources only when they are needed. You can use React.lazy and Suspense for components, dynamic imports for assets, and libraries like react-lazy-load-image-component for images. This technique reduces initial load time and improves user experience.

**What is the diff btw useState and useReducer**

Encourage useReducer for:

* Complex forms
* Nested state objects
* Undo/redo features

Use useState for:

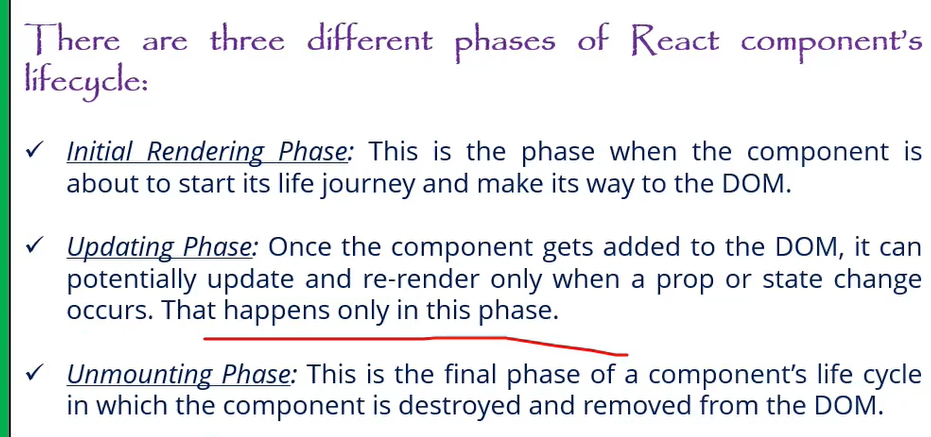
Simple toggles

Primitive values

Quick UI flags

**What is the use of custom Hooks ?**

|  |  |
| --- | --- |
| **Functional component** | **Class component** |
| No need to import react. | Import of (react & component) must include |
| Normal function syntax with return () is must. | **Class App extends Component** is must, in order to act as react component |
| light weight | Extra structure is needed. |
| Destructing will be done by passing within args | Extra line needed for destructuring, Method and variables |
| Here we cannot use life cycle methods in functional components we can handle it using HOOKS except **ComponentDidCatch** error boundaries. (higher level error catch). | Render () is also react lifecycle method. |
|  |
| Mounting- **ComponentDidMount** |
| Updating- **ComponentDidUpdate** |
| Unmounting- **ComponentWillMount** |
| function Heva (){  return (  <>  <p>She is playing</p>  <p>Here hobby is playing</p>  </>  ) } | Class component: Methods and   import React, { Component } from 'react';  class classComponent extends Component {  render() {  return (  <div>    </div>  );  } }  export default class; |



**Can you explain the main differences between React class components and functional components with hooks? Why might you choose one over the other?**

**Lifecycle Methods in Class Components:**

- \*\*Mounting:\*\* - `componentDidMount`: Called once the component is mounted.

- \*\*Updating:\*\* - `componentDidUpdate`: Called after the component updates due to changes in props or state.

- \*\*Unmounting:\*\* - `componentWillUnmount`: Called right before the component is unmounted and destroyed.

**Note**: `componentWillMount` is not recommended for use anymore due to potential issues and is considered legacy. For mounting, use `componentDidMount` instead.

2. \*\*Functional Components with Hooks:\*\*

- Functional components do not have lifecycle methods directly. Instead, you use hooks to achieve similar functionality:

- \*\*`useEffect` Hook\*\*: This hook can replicate `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` behavior. By specifying dependencies, you can control when effects run.

- \*\*`useState` Hook\*\*: Used for managing state in functional components, replacing the need for `this.state` and `this.setState`.

\*\*Why Choose One Over the Other?\*\*

- \*\*Functional Components\*\*: Generally preferred for their simplicity and ease of use. Hooks make it easier to share logic and manage state without worrying about the complexity of class components.

- \*\*Class Components\*\*: Might be used for legacy codebases or when you need specific lifecycle methods not yet covered by hooks (although hooks generally cover all use cases).

Would you like to elaborate more on this or move to the next question?

**what are the main files in React.js**

Minimum 6 filers are.

* Index.html file-🡪 Though single page application which is under public folder.
* Components🡪 The components which should be in application are used to place under src folder only.
* App.js🡪 Main component. or root components
* App.test.js-🡪 unit test
* Index.css-🡪 glogal css file.
* Index.js file🡪 entry point of javascript file

**Why to use React over other framework?**

• Performance: React excels at handling dynamic UIs with

minimal re-renders, keeping your app fast and snappy.

• Composability: React components promote code reusability,

making development faster and easier to maintain.

• JSX: JSX syntax blends JavaScript with HTML, making UI

creation intuitive for developers.

Large Community: A vast community and ecosystem provide support,

libraries, and tools for rapid development.

**What’s your approach to debugging in React?**

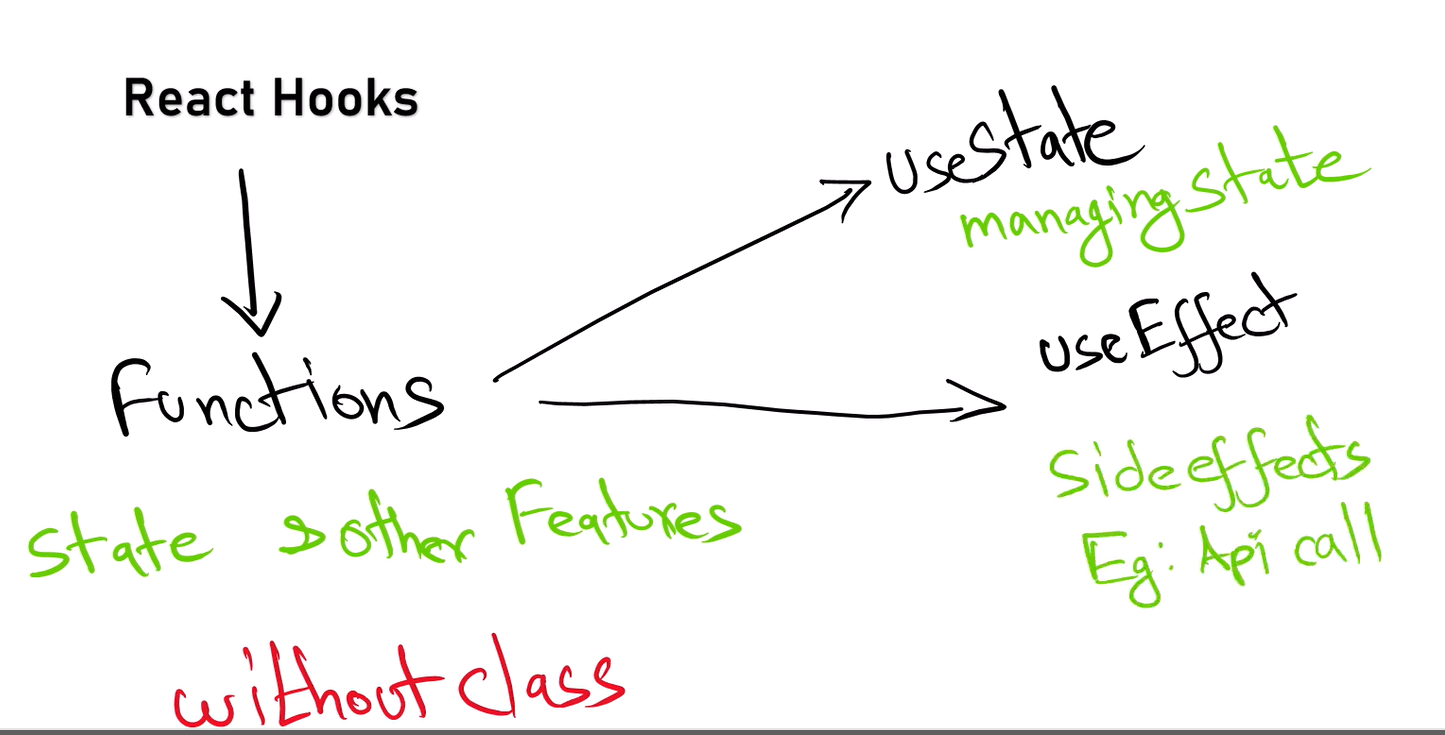
**Arrow function expression in React.js?**

Arrow function is a way of declaring the function using the arrow operator where first pass the parameters in the bracket then put the whole logic.

Difference we don’t need to add function and function name.

Assigning function itself to a variable.

**React hooks**



**Parent to child pros?**

Passing data from parent to child components in React provides benefits such as centralized data management, controlled data flow, component reusability, simplified state management, enhanced readability, encapsulation, and flexibility. This approach aligns with React’s design principles and helps build maintainable and scalable applications

**what is the disadvantages of using JavaScript in reactjs?**

If you developed react js using Javascript and we declare an object variable with string data type this will throw error at run time in production browser so will face type safety with javascript.

**Authentication flow in React.js**

Authentication Context: Use the React Context API to manage authentication state and provide it throughout your application.

Authentication Components: Create login and logout components and handle user authentication.

Protect Routes: Use a ProtectedRoute component to restrict access to certain routes based on authentication state.

Persistence: Persist authentication state using localStorage or sessionStorage.

API Integration: Implement API calls for login and manage authentication tokens as needed.

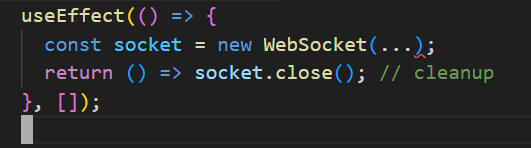
**Side effects?**

While the component is re-rendering the logic inside the component will execute again and again, in this case we have to isolate that particular logic using use Effect.

**Can you explain useEffect with a real-time use case?**

I use useEffect(() => {...}, []) to fetch data on mount.

For cleanup (like WebSockets or event listeners), I return a cleanup function:



**What’s the difference between useEffect, useLayoutEffect, and useInsertionEffect?**

* useEffect Runs **after the DOM is painted** (asynchronous). **Non-blocking** – good for API calls, timers, subscriptions, The user **sees UI updates** immediately before effect runs.
* useLayoutEffect Runs **synchronously after DOM mutations but before paint**.

**Blocking** – used when you need to **measure DOM or update layout** before the browser paints

Can cause performance issues if overused.

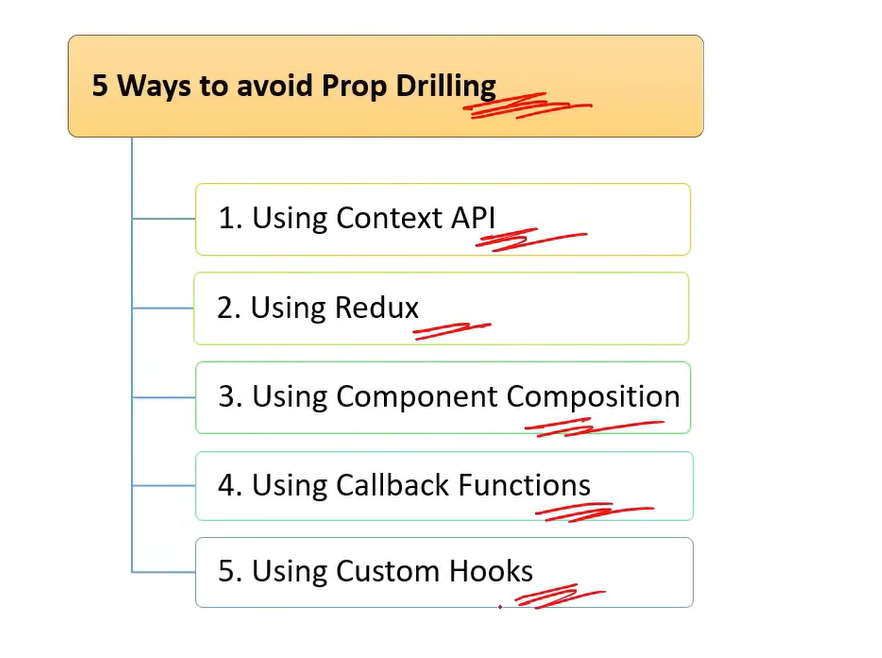
* useInsertionEffect (introduced in React 18) runs **before DOM mutations**, mainly used for libraries injecting styles (e.g., Emotion).

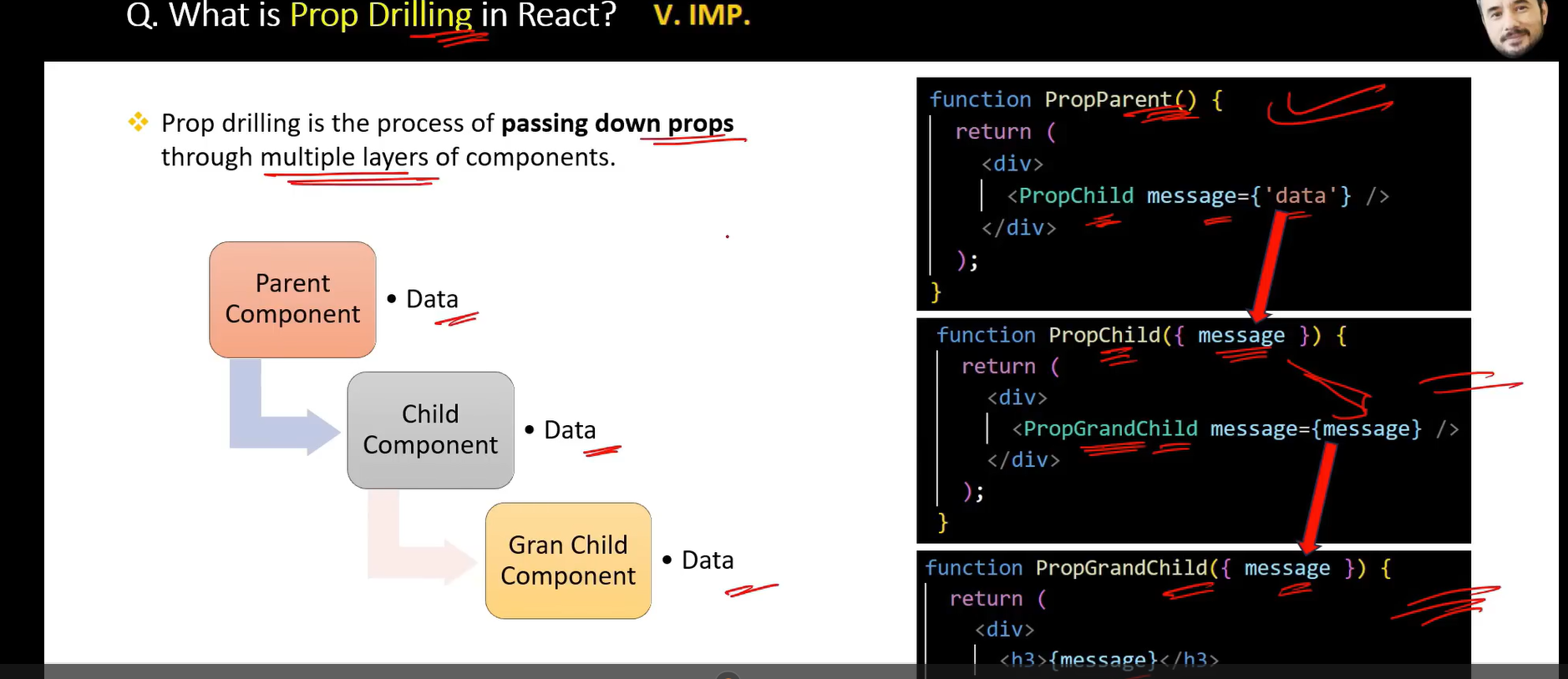
**What is Props drilling in React and how to overcome.**

Passing of props to the multi-layer of components (Nested components) or is called props drilling but the properties passing may not useful for all components that is wasting of memory

Complexity increases code reusability.

Debugging becomes challenging in in order to overcome this we have use Context. Or using redux library we can avoid props drilling.





**How to protect a Route.?**

Protected route is used to handle the security which is avail from React router dom.

**Stateless and stateful components?**

**Stateless Components:**

* Do not manage internal state.
* Render UI based on props.
* Simpler and more focused on presentation.

**Stateful Components:**

* Manage internal state and lifecycle methods (class components) or side effects (functional components with hooks).
* Handle user interactions and complex operations.
* More complex and capable of managing state and side effects.

**Purpose of React render Method?**

The render method is crucial in React for defining the component’s UI based on its state and props. It:

* Describes the structure of the UI.
* Updates the UI when state or props change.
* Composes complex UIs using other components.
* Operates as a pure function, ensuring predictable UI updates.

**Controlled and uncontrolled components?**

Controlled Components:

* State Management: Form data is managed via React state.
* Predictable: The component’s state is the single source of truth.
* Event Handling: Input changes update React state.
* Advantages: Easier state management, validation, and debugging.

Uncontrolled Components:

* Ref Management: Form data is managed through the DOM.
* Less Control: React does not directly manage form data.
* Event Handling: Values are accessed directly from the DOM.
* Advantages: Less boilerplate, direct access to DOM values.

**What is the difference between HTML and React event handling?**

**What is the purpose of keys in React lists?**

Keys help React identify which items changed, are added, or removed. They optimize re-renders by giving each list item a unique identity.

Wrong Practice: Using index as a key can lead to bugs during reordering.

**What is prop drilling and how can you avoid it?**

Prop drilling is passing props through multiple levels of components unnecessarily. It can be avoided using Context API or state management tools like Redux or Zustand.

Example:  
Instead of passing theme from App → Layout → Header → Button, we used React.createContext and useContext to directly access it inside Button.

**What are Pure Components?**

\*`React.PureComponent`\* is exactly the same as \*`React.Component`\* except that it handles the `shouldComponentUpdate()`, the component will re-render by default whenever `shouldComponentUpdate` is called.

a **Pure Component** is a component that only re-renders when its props or state change.

**What is lifting stateup?**

Lifting state up involves moving state management to the closest common ancestor of components that need to share the state. This pattern helps in maintaining a single source of truth, synchronizing data between components, and simplifying state management in React applications.

**What is reconciliation?**

**What are Fragments?**

**What are stateless components?**

**What are stateful components?**

**How do you apply validation to props in React?**

**What are the limitations of React?**

**How do you apply styles in React?**

**How do you conditionally render components?**

Conditional rendering in React lets us control what appears in the UI based on state, props, or any logic.  
It uses standard JavaScript conditions like if, ? :, or && inside JSX.

In my projects, I’ve used it for showing loading states, error handling, and dynamic layouts like login/logout views or role-based dashboards.

**How do you conditionally apply class attributes?**

**Dynamic Imports with React.Lazy?**

* React.lazy: Allows you to load components dynamically, enabling code splitting.
* Dynamic import(): The mechanism behind React.lazy for asynchronous module loading.
* React.Suspense: Manages the loading state for components loaded lazily.
* Error Boundaries: Handle potential errors in loading components to improve user experience.

## **How do you structure a scalable React application?**

I use a modular folder structure like:  
src/  
 components/  
 pages/  
 hooks/  
 utils/  
 services/  
 store/  
  
I encourage:  
- Atomic design principles for components  
- Centralized API service layer  
- Environment-based config management  
- Lazy-loaded routing setup  
- Separation of concerns  
  
As a lead, I emphasize code readability, maintainability, and testability.

**What is Concurrent Rendering in React 18?**

Concurrent rendering allows React to prepare multiple versions of the UI at the same time. It improves responsiveness and avoids UI blocking during heavy rendering. Features like automatic batching, transitions, and startTransition() help React prioritize urgent updates.  
  
As a Tech Lead, I focus on using these APIs to ensure better user experience without performance drops.

## **How do you ensure code quality in a React project?**

I enforce:  
- Strict ESLint + Prettier rules  
- Unit and integration tests using Jest, React Testing Library  
- Code reviews with PR templates  
- CI/CD integration for testing and linting  
- Component documentation using Storybook  
  
As a Tech Lead, I balance quality and velocity by automating checks and mentoring the team on best practices.

**How do you handle SEO in a React application?**

Since React is client-side rendered, SEO can be a challenge. For SEO-critical apps, I recommend:  
- Server-side rendering (SSR) with Next.js  
- Using React Helmet for dynamic meta tags  
- Pre-rendering or static site generation where possible  
- Ensuring crawlable, semantic HTML

## **What’s your approach to mentoring junior React developers?**

I pair them with senior devs for code reviews, introduce component reuse and state management basics, and gradually give them ownership of modules. I also encourage best practices early—like meaningful naming, code comments, and testing. Regular 1:1s and shared learning sessions keep them engaged and growing.

**How do you ensure code quality in your team as a Tech Lead?**

- Code reviews with focus on clarity, performance, and security.

- Setup ESLint + Prettier for consistent code style.

- Maintain unit and integration test coverage.

- Automate builds, tests, and deploys using CI/CD pipelines (Jenkins/GitHub Actions).

- Encourage knowledge sharing, mentoring, and pair programming

Using **React.lazy and dynamic imports effectively can help improve the performance of your React application by reducing the initial bundle size and loading components only when they are needed.**

**Sure! Let's dive into one of the techniques.**

**### Technique: Code Splitting**

**\*\*Code Splitting\*\* helps improve the performance of your React application by loading only the necessary code when it’s needed. This can reduce the initial load time of your application and improve user experience.**

**#### \*\*How to Implement Code Splitting:\*\***

**1. \*\*Dynamic Imports with React.lazy:\*\***

**- `React.lazy` allows you to load a component lazily when it’s needed.**

**- You wrap your lazy-loaded component in a `React.Suspense` component, which allows you to specify a fallback UI (e.g., a loading spinner) while the component is being loaded.**

**\*\*Example:\*\***

**```jsx**

**import React, { Suspense } from 'react';**

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

**function App() {**

**return (**

**<div>**

**<Suspense fallback={<div>Loading...</div>}>**

**<LazyComponent />**

**</Suspense>**

**</div>**

**);**

**}**

**```**

**2. \*\*Dynamic Import with React Router:\*\***

**- If you’re using React Router for navigation, you can also dynamically import route components.**

**\*\*Example:\*\***

**```jsx**

**import React, { Suspense } from 'react';**

**import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';**

**const HomePage = React.lazy(() => import('./HomePage'));**

**const AboutPage = React.lazy(() => import('./AboutPage'));**

**function App() {**

**return (**

**<Router>**

**<Suspense fallback={<div>Loading...</div>}>**

**<Switch>**

**<Route path="/home" component={HomePage} />**

**<Route path="/about" component={AboutPage} />**

**</Switch>**

**</Suspense>**

**</Router>**

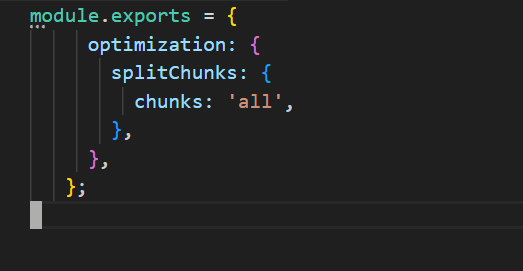
**);**

**}**

**3. \*\*Webpack Code Splitting:\*\***

**- For more advanced scenarios, you can configure Webpack to split code into separate bundles based on your application's needs. This usually involves setting up Webpack’s optimization options, like `splitChunks`.**

**Example Configuration (Webpack):\*\***

****

**\*\*Benefits of Code Splitting:\*\***

**- \*\*Improved Performance:\*\* Reduces the amount of JavaScript needed on the initial page load.**

**- \*\*Faster Load Times:\*\* Users only load the components they need, which can lead to quicker interactions.**

**- \*\*Enhanced User Experience:\*\* By deferring the loading of non-critical components, users see meaningful content faster.**

**FORMs?**

React Forms are the components used to collect and manage the user inputs.

**Events?**

React Events are user interactions with the web application

**Difference in ES5 ES6?**

* ES5: The version before ES6, with more basic features and less syntactic sugar.
* ES6: Introduced many new features, including let, const, arrow functions, classes, modules, template literals, destructuring, default parameters, rest and spread operators, and promises.

**Bundle of react?**

Bundling in React involves combining JavaScript, CSS, and other assets into optimized files to enhance performance and reduce the number of HTTP requests. Tools like Webpack, Vite, and Parcel are commonly used to handle this process. Bundling not only helps in optimizing load times but also in managing and organizing code effectively.

**Redux**

**What is Redux?**

Redux is a state management library used to manage global state across a React application in a predictable, centralized, and debuggable way.

**What is the purpose of Redux in a React application? When should you use Redux instead of useState or useContext?**

At when we can Use Redux:

* The app has complex global state (e.g., user auth, theme, cart, dashboard data)
* Many components need shared data
* You want predictability and debugging
* The app requires middleware for async logic (e.g., API calls using Redux Thunk/Saga)

**How to use Redux in Functional Component ?**

**What is Redux ?**

Redux is a predictable state management library for JavaScript apps, commonly used with React.

It helps manage global state — the data that needs to be shared across many components — in a centralized and consistent way.

**Why we have to use redux ?**

* Helps when many components need access to the same state (e.g., authentication, theme, language, cart).
* Improves testability and debuggability.
* Useful for handling complex async flows (e.g., API calls) using Redux Thunk or Redux Saga.

**What is Reducer ?**

A reducer is a pure function that takes the current state and an action, and returns a new updated state — without mutating the original state.

**How redux store is available to react app ?**

We use the <Provider> component from the react-redux library to inject the Redux store into the React component tree.

**What is redux Logger ?**

Redux Logger is a middleware that logs every action dispatched and the resulting state in the console.

It is mainly used for development to help visualize:

* The action type dispatched
* The previous state
* The next state

**How to call the reducer ?**

You don’t call a reducer directly — instead, you dispatch an action, and Redux internally calls the corresponding reducer to update the state.

**Can you explain Redux flow ? ()**

Redux is a state management library commonly used with React to manage the state of applications in a predictable way. The Redux flow refers to the sequence of steps and interactions involved in managing state changes within a Redux-based application.

**Action Creation**: An action is created (either directly or using an action creator).

**Dispatching Action**: The action is dispatched to the store.

**Reducer Handling**: The store sends the action to the appropriate reducer.

**State Update**: The reducer processes the action and returns a new state.

**State Change Notification**: The store updates its state and notifies subscribed components or listeners.

**Component Rendering**: Components access the updated state (usually via mapStateToProps or hooks like useSelector) and re-render based on the new state.

**Core Concepts: Store, Action, Reducer, Dispatch**

**react-redux hooks: useSelector, useDispatch, Provider**

**Redux Toolkit (modern approach)**

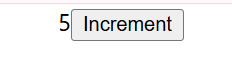
**Smart example: Counter or Todo using Redux**

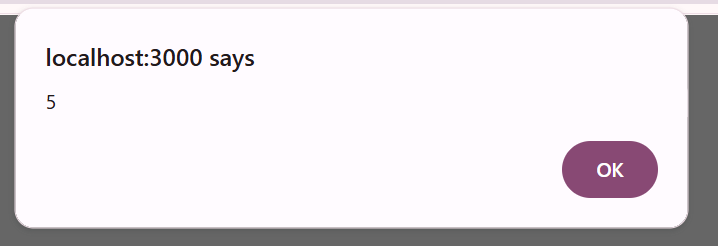
**Real-time scenario Q&A**

**Coding Exercise**

**What is the output of below code**

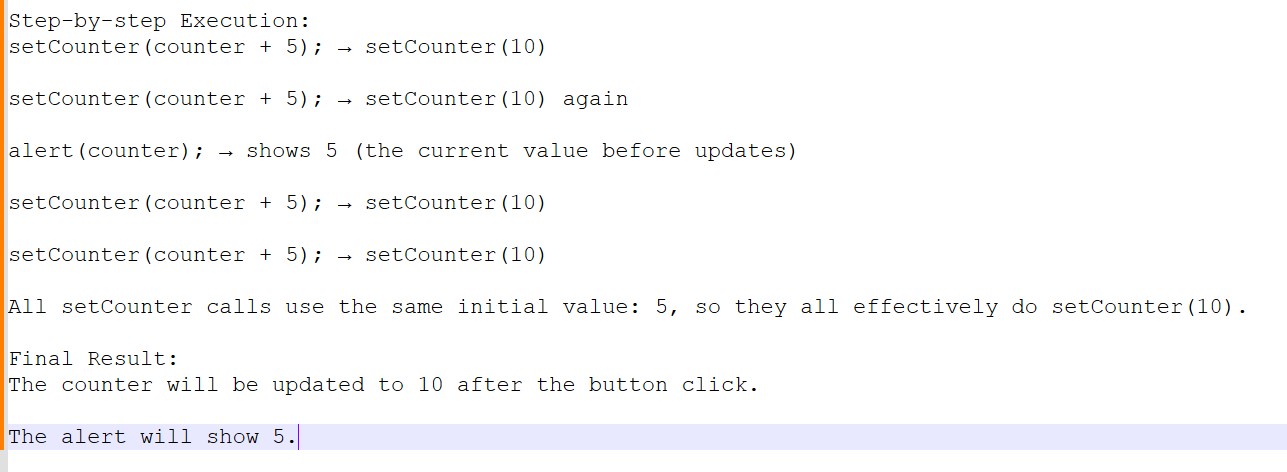
****

On screen:  


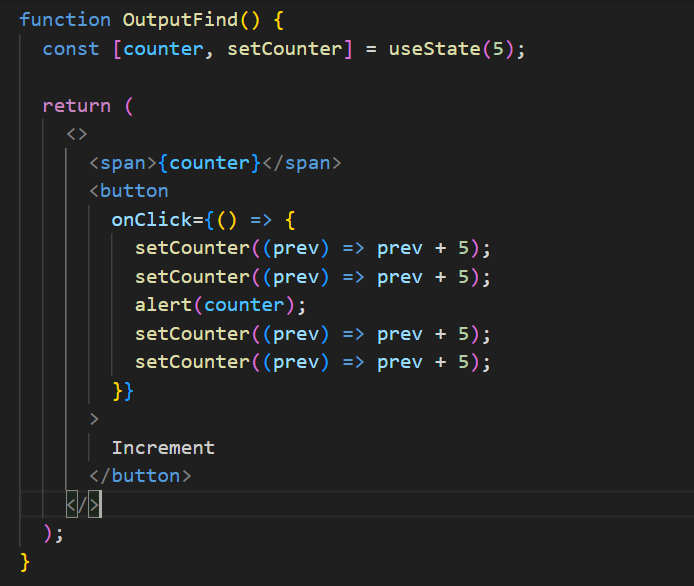
In alert popup:  


**alert(counter) shows the value before any updates take effect**:

* + Since state updates are asynchronous, alert(counter) will show the value before any of the setCounter calls are applied.

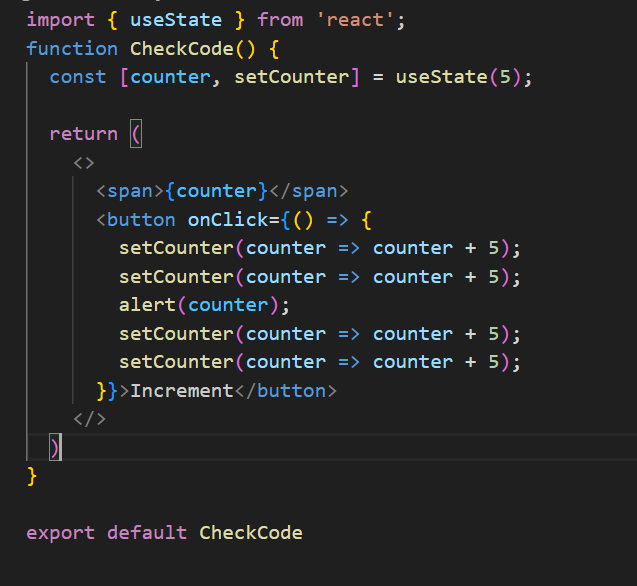


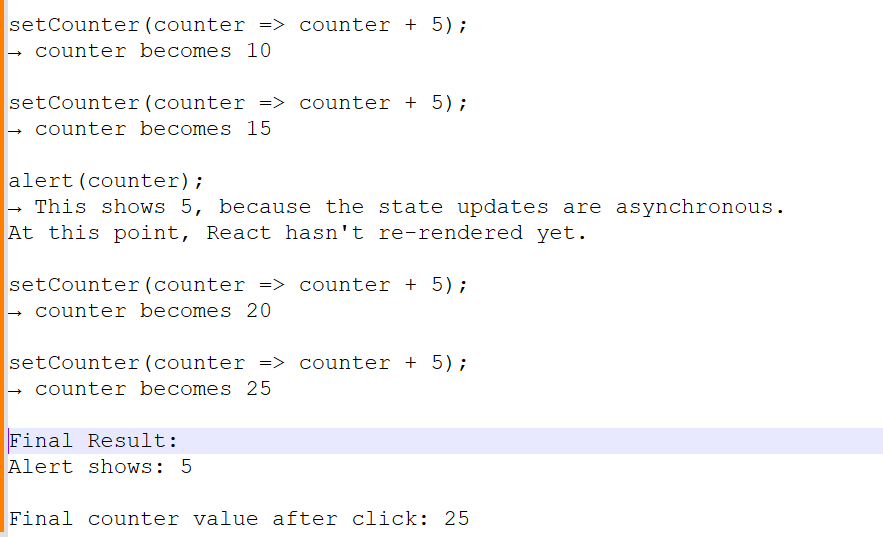
**In order to make work do below changes.**



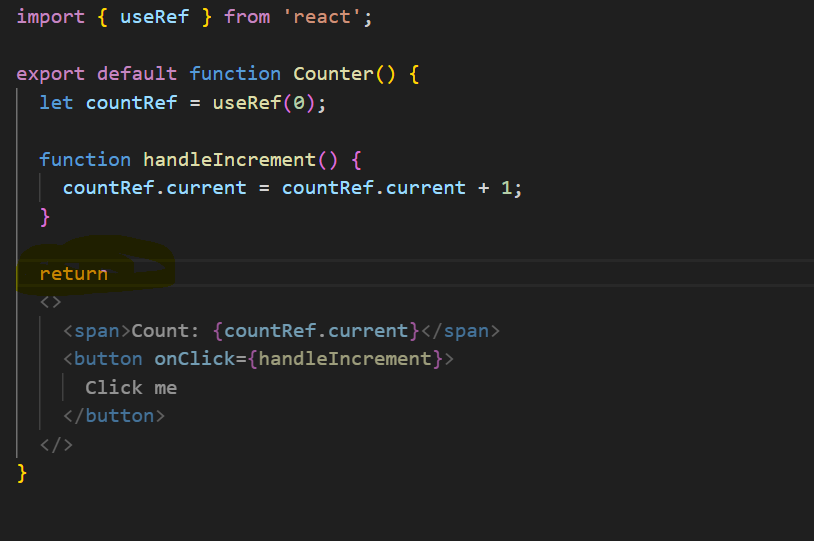
* **Alert shows**: 5
* **Final counter value**: 10 (with original code)
* **Final counter value with functional updates**: 25 (if using prev => prev + 5 four times)

**What is the output of below code**

****

****

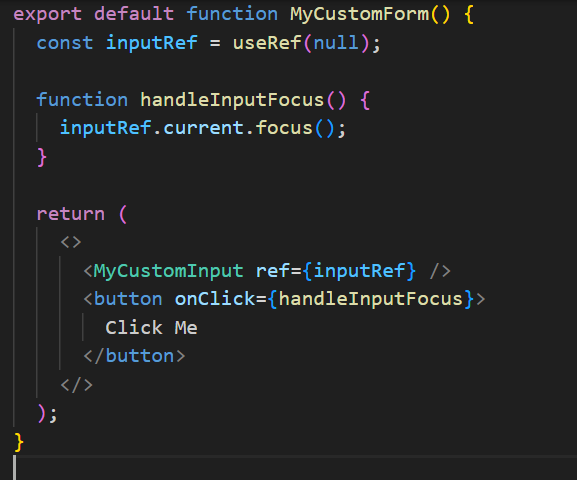
**What is the output of span after one click?**

****

**The return statement ends right after return, so the JSX is not returned at all — JavaScript automatically inserts a semicolon after return!**

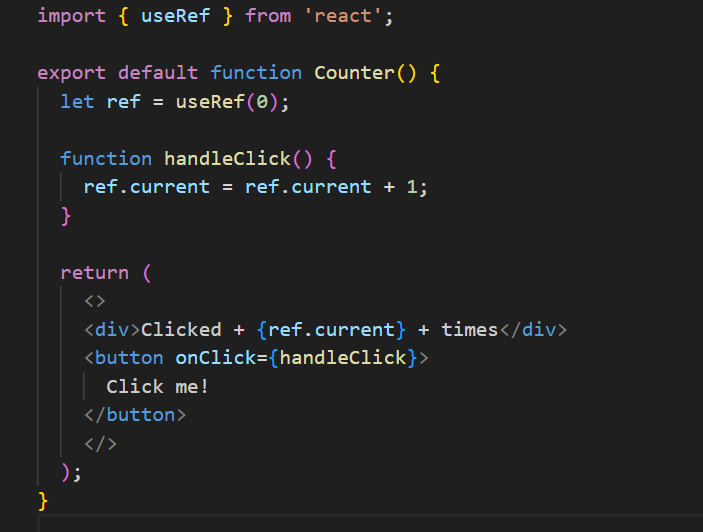
So the JSX inside is **never returned**, meaning the component returns **nothing** — React renders **nothing**.

**What is the outcome of below code after button click?**

****

You will get a **runtime warning or error** in the console:  
You **can’t directly attach a ref to a functional component** like <MyCustomInput />. Because ref is meant to attach to a **DOM node**, not to a function.

**What is the outcome of number of clicks after 3 button clicks?**

****

useRef updates do not trigger re-render in React.

**What does this code do?**  
  
const [items, setItems] = useState([1, 2, 3]);

const newItems = items; //**This doesn't clone the array; it's a reference.**

newItems.push(4);// **Mutates the original array in-place.**

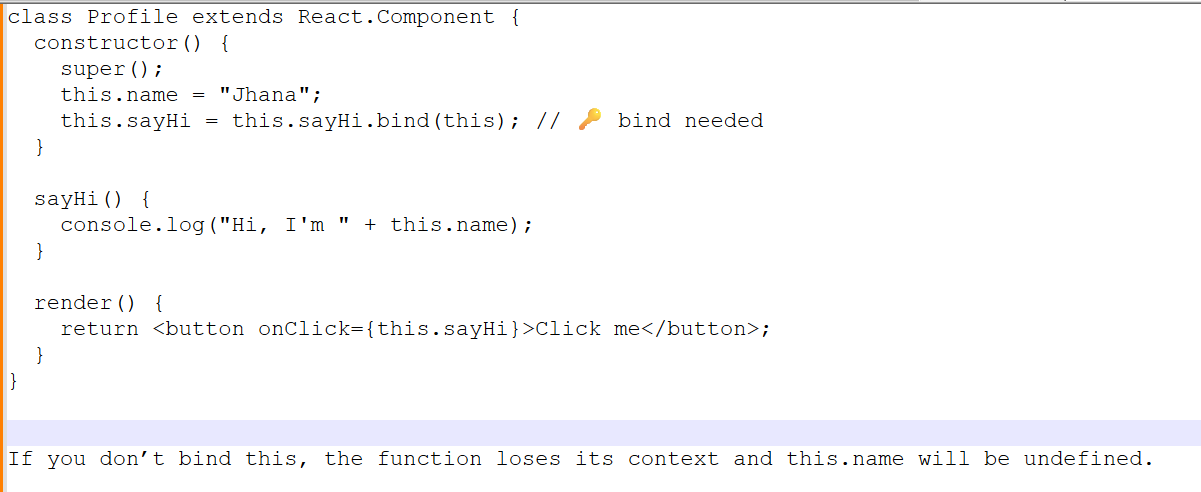
setItems(newItems); //tries to update state with **the same array (by reference)**.

The array becomes [1, 2, 3, 4] but React **may not re-render**, because the **reference to the array hasn't changed**.

React re-renders only when state **identity (reference)** changes. In this case, items and newItems are the **same reference**, so React might skip rendering.

Correct Approach : setItems([...items, 4]);

**NEXT**



**NEXT**

**NEXT**

**NEXT**