**REACT INTERVIEW QUESTIONS**

**🔍 What is the Difference Between a Library and a Framework?**

| **Feature** | | **Library** | | **Framework** | |
| --- | --- | --- | --- | --- | --- |
| **Definition** | | Collection of functions/methods you can call | | A complete structure that controls your app | |
| **Usage** | Pick and use functions as needed | | Must follow the framework’s rules and patterns | |

|  |  |  |
| --- | --- | --- |
| **Examples** | React (UI library), Lodash, Axios | Angular, Django, Spring, Ruby on Rails |
| 🧠 Real-Life Analogy  Imagine you're building a house.   * Library = Set of tools (hammer, saw, drill). You decide how and when to use them. * Framework = A blueprint + contractor who tells you where to place rooms, pipes, and windows. You fill in the pieces.   **REACT** |  |  |

**📌 Definition:**

React is a **JavaScript library** (not a full framework) used for building **user interfaces**, especially **Single Page Applications (SPAs)** using a **component-based architecture** and a fast **Virtual DOM**.

**🔍 Why React?**

| **Feature** | **What It Means** | **Real-Life Analogy** |
| --- | --- | --- |
| Component-Based | Break UI into reusable pieces | Like reusable LEGO blocks for building many structures |
| Virtual DOM | Efficient updates to web pages | Like editing a digital photo preview instead of reprinting |
| SPA (Single Page App) | Loads one HTML page, updates dynamically | Like using a remote control instead of changing the TV set |

| **Traditional JS/HTML** | **React** |
| --- | --- |
| Multiple HTML pages | One index.html |

**What is JSX?**

JSX combines **HTML** and**JavaScript** in a single syntax, allowing you to create **UI components** in React

**How JSX Works**

When[React](https://www.geeksforgeeks.org/react/) processes this JSX code, it converts it into JavaScript using **Babel**. This JavaScript code then creates real HTML elements in the browser’s DOM . which is how your web page gets displayed.

**JSX Transformation Process**

* **Writing JSX:**Write JSX just like HTML inside JavaScript files (React components).

const element = <h1>Hello, World!</h1>;

* **JSX Gets Transformed:** JSX is not directly understood by browsers. So, it gets converted into JavaScript by a tool called **Babel**. After conversion, the JSX becomes equivalent to[React.createElement()](https://www.geeksforgeeks.org/what-is-the-use-of-react-createelement/)calls. After transformation JSX becomes.

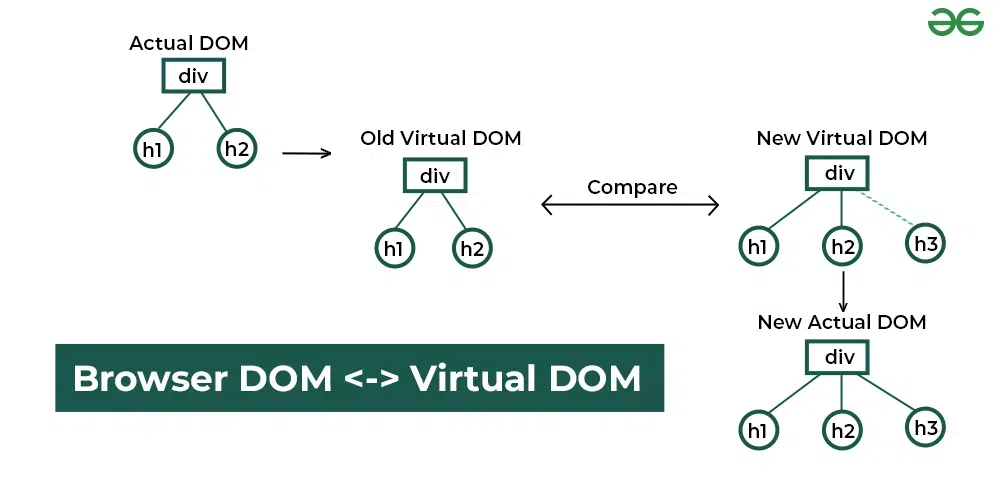
const element = React.createElement('h1', null, 'Hello, World!');

* **React Creates Elements:** React takes the JavaScript code generated from JSX and uses it to create real DOM elements that the browser can render on the screen.

**📌 What is the Virtual DOM?**

The **Virtual DOM (VDOM)** is a **lightweight JavaScript copy** of the **Real DOM** (Document Object Model).  
Instead of updating the browser's DOM directly, React updates the VDOM first, **calculates the difference** (diffing), and then updates the **minimum number of real DOM nodes**.

React updates only the necessary parts of the real DOM, instead of re-rendering everything.



**✅ 2. ReactDOM**

**📌 What is ReactDOM?**

**ReactDOM** is the **package** that connects React with the **actual DOM in the browser**. It handles **rendering** your React components to HTML and updating the DOM based on the virtual DOM diffing.

function App() {

const items = ['Apple', 'Banana', 'Cherry'];

return (

<div>

<h1>My Fruit List</h1>

<ul>

{items.map((item, index) => (

<li key={index}>{item}</li>

))}

</ul>

</div>

);

}

export default App;

**5. {items.map((item, index) => ( ... ))}**

* map() is a JavaScript function used to **loop over an array**.
* Here, it loops through items:
  + On the first iteration, item = 'Apple'
  + On the second, item = 'Banana'

**6. key={index}**

* key is a **special prop in React** used to uniquely identify each list item.
* It helps React optimize rendering by tracking which items change, are added, or removed.
* Here, we're using the index as a key (not ideal in real apps where items can change order, but fine for static lists).
* **❓ So what happens with key={index}?**
* When you write:
* jsx
* CopyEdit
* {items.map((item, index) => (
* <li key={index}>{item}</li>
* ))}
* You're saying:
* “Hey React, use the **position in the array** (index) as the unique identity.”
* This **works** for **static lists** that **never change**.
* **🔹 What is “Rendering” in React?**
* **Rendering** means **displaying elements (UI components) on the screen**.
* In React, you “render” elements or components into the **Real DOM** using the ReactDOM package.

**🔧 Example:**

**✅ index.js**

import React from 'react';

import ReactDOM from 'react-dom/client';

import App from './App';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<App />);

**✅ App.js**

function App() {

return <h1>Hello, React World!</h1>;

}

export default App;

**✅ index.html (in public folder)**

<body>

<div id="root"></div> <!-- This is where React will render -->

</body>

**CONDITIONAL RENDERING**

function Greeting({ isLoggedIn }) {  
 return <h1>{isLoggedIn ? "Welcome Back!" : "Please Sign In"}</h1>;  
}

**Types of React Components**

There are two primary types of React components:

**1. Functional Components**

[Functional components](https://www.geeksforgeeks.org/reactjs-functional-components/) are simpler and preferred for most use cases. They are[JavaScript functions](https://www.geeksforgeeks.org/functions-in-javascript/) that return [React](https://www.geeksforgeeks.org/react/) elements. With the introduction of [React Hooks](https://www.geeksforgeeks.org/reactjs-hooks/), functional components can also manage state and lifecycle events.

* **Stateless or Stateful:**Can manage state using React Hooks.
* **Simpler Syntax:**Ideal for small and reusable components.
* **Performance:** Generally faster since they don’t require a 'this' keyword.

**function** Greet(props) {

**return** <h1>Hello, {props.name}!</h1>;

}

**2. Class Components**

[Class components](https://www.geeksforgeeks.org/reactjs-class-components/) are [ES6 classes](https://www.geeksforgeeks.org/es6-classes/) that extend React.Component. They include additional features like state management and lifecycle methods.

* **State Management:** State is managed using the this.state property.
* **Lifecycle Methods:** Includes methods like [componentDidMount,](https://www.geeksforgeeks.org/reactjs-componentdidmount-method/) [componentDidUpdate](https://www.geeksforgeeks.org/reactjs-componentdidupdate-method/), etc.

**class** Greet **extends** React.Component {

render() {

**return** <h1>Hello, {**this**.props.name}!</h1>;

}

}

**PROPS**

* **Props** (short for “properties”) are like **arguments** you pass to a component.
* Just like you pass parameters to a function, you pass props to a component when using it.

**✅ Example:**

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

<Welcome name="Dinesh" />;

Here, "Dinesh" is passed **from parent to child** as props.name.

**STATE**

**State** is a built-in React object used to **store and manage data** that can **change over time** inside a component.  
When the state changes, React **automatically re-renders** the component to reflect the updated data.

**✅ Simple Example:**

import React, { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0); // 'count' is stat

return (

<div>

<p>You clicked {count} times</p>

<button onClick={() => setCount(count + 1)}>Click</button>

</div>

);

}

🟢 count is **state**  
🔄 setCount() updates it  
⚡ UI changes whenever count changes

**REACT LIFECYCLE**

**🌀 Three Main Phases of Lifecycle:**

| **Phase** | **What It Means** |
| --- | --- |
| **Mounting** | Component is **created** and inserted into DOM |
| **Updating** | Component is **re-rendered** due to state/props change |
| **Unmounting** | Component is **removed** from the DOM |

**🧠 Real-Life Analogy:**

Think of a React component like a **person’s life**:

* **Mounting**: They are **born** (component is created)
* **Updating**: They **grow** and **change** (state/props change)
* **Unmounting**: They **retire** or **die** (component is removed)
* **🔍 Lifecycle Methods (Class Components)**

| **Phase** | **Lifecycle Method** | **Purpose** |
| --- | --- | --- |
| Mounting | constructor() | Set initial state or bind methods |
|  | componentDidMount() | API calls, subscriptions (runs once after first render) |
| Updating | componentDidUpdate() | Runs after state or props change |
| Unmounting | componentWillUnmount() | Clean up (like timers, subscriptions) |

“The React lifecycle describes how a component is created, updated, and destroyed. Class components use methods like componentDidMount() and componentWillUnmount(), while functional components use the useEffect hook to achieve the same effects.”

**✅ What is Event Handling in React?**

**Event Handling** is how we make our React app respond to **user actions** like clicks, typing, hovering, etc.Just like in plain HTML/JavaScript, but in React we use **camelCase event names** and pass a **function** as the event handler.

**🧠 Real-Life Analogy**

Think of a **doorbell**: when someone presses it (event), a function is triggered (ring the bell).

In React:  
<button onClick={ringBell}>Ring</button>

**📌 Common React Events**

| **Event** | **Description** |
| --- | --- |
| onClick | Button click |
| onChange | Input change |
| onSubmit | Form submission |
| onMouseOver | Mouse hover |
| onKeyDown | Key pressed on keyboard |

**✅ What is Prop Drilling in React?**

**Prop Drilling** is when you **pass data (props) through multiple levels of components**, even if **only the last component** needs the data.

**🧠 Real-Life Analogy**

You want to give a message to your **friend in a classroom**, but you’re outside. So, you pass the message through **5 classmates** sitting in a line, even though **only the last person** reads it.

The message = prop  
The classmates = intermediate components

**🔧 Simple Example**

**🧩 Component Structure**

kotlin

CopyEdit

App

└── Parent

└── Child

└── GrandChild (needs the data)

**❌ Problems with Prop Drilling**

* Adds unnecessary **complexity**
* Intermediate components become **bloated**
* Difficult to **maintain** or **refactor**
* **Not scalable** for large applications
* **🧠 Interview Answer (Say This):**
* “Prop drilling occurs when we pass props through multiple levels of components just to reach a deeply nested child. While it works, it’s not scalable. That’s why React provides solutions like the Context API or state management libraries (Redux, Zustand) to share data globally without drilling.”

**🔷 1. useState – To store data (memory)**

**✅ Simple Definition:**

useState lets your component **remember and update data** (like a variable that changes when a user clicks a button).

**🧠 Real-life Analogy:**

Imagine a **whiteboard** on your desk. You write something (like a number). When it changes, you erase and write again.

**💻 Example:**

import React, { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0); // 0 is initial value

return (

<div>

<p>You clicked {count} times</p>

<button onClick={() => setCount(count + 1)}>Click</button>

</div>

);

}

**🗣️ Interview:**

"useState is used to store changing data in a functional component. When state changes, the component re-renders."

**🔷 2. useEffect – To do something after render (side effects)**

**✅ Simple Definition:**

useEffect runs code **after the component is displayed**, like fetching data or setting a timer.

**🧠 Real-life Analogy:**

You wake up (render), then you brush your teeth (effect after render). Brushing is a **side effect**.

**💻 Example:**

import React, { useEffect } from 'react';

function Hello() {

useEffect(() => {

alert('Component displayed!');

}, []);

return <h1>Hello!</h1>;

}

**🗣️ Interview:**

"useEffect is used for side effects like fetching API data, running timers, or updating the DOM after rendering."

**🔷 3. useContext – To share data globally**

**✅ Simple Definition:**

useContext lets you **use data from a global store** instead of passing props manually to every component.

**🧠 Real-life Analogy:**

Think of a **bulletin board** in a school. Everyone can read it without being told directly.

**💻 Example:**

import React, { useContext } from 'react';

const UserContext = React.createContext();

function Child() {

const user = useContext(UserContext);

return <h1>Hello, {user}!</h1>;

}

function App() {

return (

<UserContext.Provider value="Dinesh">

<Child />

</UserContext.Provider>

);

}

**🗣️ Interview:**

"useContext is used to access global values without passing props through each level of components."

**🔷 4. useRef – To reference a DOM element or keep a value**

**✅ Simple Definition:**

useRef is used to **get direct access to HTML elements**, or store a value that doesn’t cause re-render.

**🧠 Real-life Analogy:**

It’s like writing a note on a sticky pad that stays with you but doesn’t affect your surroundings.

**💻 Example:**

import React, { useRef } from 'react';

function InputFocus() {

const inputRef = useRef();

function focusInput() {

inputRef.current.focus(); }

return (

<>

<input ref={inputRef} />

<button onClick={focusInput}>Focus</button>

</>

);

}

**🗣️ Interview:**

"useRef lets us access or modify a DOM element directly or store a mutable value without re-rendering."

**5. useMemo – To remember expensive calculations**

**✅ Simple Definition:**

useMemo saves a **calculated result** so that it doesn’t have to re-calculate every time unless needed.

**🧠 Real-life Analogy:**

If you calculate your marks and save them on a sheet, you don't re-calculate every time someone asks.

**💻 Example:**

jsx

CopyEdit

import React, { useMemo, useState } from 'react';

function ExpensiveCalculator({ number }) {

const result = useMemo(() => {

console.log("Calculating...");

return number \* 2;

}, [number]);

return <p>Result: {result}</p>;

}

**🗣️ Interview:**

"useMemo helps improve performance by storing (memoizing) a computed value and reusing it unless its dependencies change."

**🔷 6. useCallback – To remember a function**

**✅ Simple Definition:**

useCallback saves a function so it’s not re-created on every render (useful in optimization).

**🧠 Real-life Analogy:**

If you write a function on a sticky note, you just reuse the note every time, not rewrite the function.

**💻 Example:**

jsx

CopyEdit

import React, { useCallback } from 'react';

function Button({ onClick }) {

return <button onClick={onClick}>Click</button>;

}

function App() {

const handleClick = useCallback(() => {

alert("Clicked");

}, []);

return <Button onClick={handleClick} />;

}

**🗣️ Interview:**

"useCallback memoizes a function so that it doesn't get recreated unless dependencies change — helpful in performance optimization."

**🔷 7. useReducer – For complex state**

**✅ Simple Definition:**

useReducer is like useState, but better for handling **complex state logic** or multiple actions.

**🧠 Real-life Analogy:**

Think of it like a traffic controller — it listens to actions and tells the system how to respond.

**💻 Example:**

import React, { useReducer } from 'react';

function reducer(state, action) {

switch (action.type) {

case 'INCREMENT': return { count: state.count + 1 };

case 'DECREMENT': return { count: state.count - 1 };

default: return state;

}

}

function Counter() {

const [state, dispatch] = useReducer(reducer, { count: 0 });

return (

<>

<p>Count: {state.count}</p>

<button onClick={() => dispatch({ type: 'INCREMENT' })}>+</button>

<button onClick={() => dispatch({ type: 'DECREMENT' })}>-</button>

</>

);

}

**🗣️ Interview:**

"useReducer is useful when state logic is complex or when managing multiple values or actions like in Redux."

**✅ What is a Custom Hook?**

A **Custom Hook** is a **JavaScript function that starts with use** and allows you to **reuse logic between React components**.

**Example 1: useCounter Hook**

Let’s create a custom hook to manage **counter logic**.

**🔸 useCounter.js**

import { useState } from 'react';

function useCounter(initialValue = 0) {

const [count, setCount] = useState(initialValue);

const increment = () => setCount(prev => prev + 1);

const decrement = () => setCount(prev => prev - 1);

const reset = () => setCount(initialValue);

return { count, increment, decrement, reset };

}

export default useCounter;

**🔹 How to use it:**

import React from 'react';

import useCounter from './useCounter';

function CounterComponent() {

const { count, increment, decrement, reset } = useCounter(5);

return (

<>

<p>Count: {count}</p>

<button onClick={increment}>+</button>

<button onClick={decrement}>-</button>

<button onClick={reset}>Reset</button>

</>

);