React JS
React is an efficient, flexible, and open-source **JavaScript library** that allows developers to create simple, fast, and scalable web applications

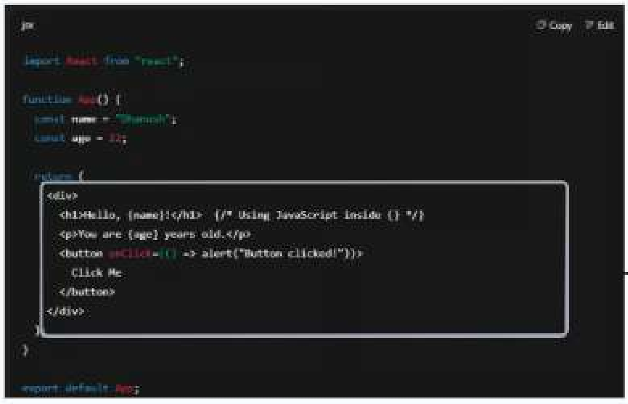
DIff bwtween framework and library
Library:
A collection of functions or reusable code that you can call whenever you want — **you control when and how to use them.**

Framework:
Also provides reusable code, but it comes with a predefined **structure and flow**, and **it calls your code at specific points**
(inversion of control).

Main features of Reactjs
1)**Component-Based Architecture** – Build UIs as reusable, independent components.
2)**Virtual DOM** – Efficiently updates only the changed parts of the UI for better performance.

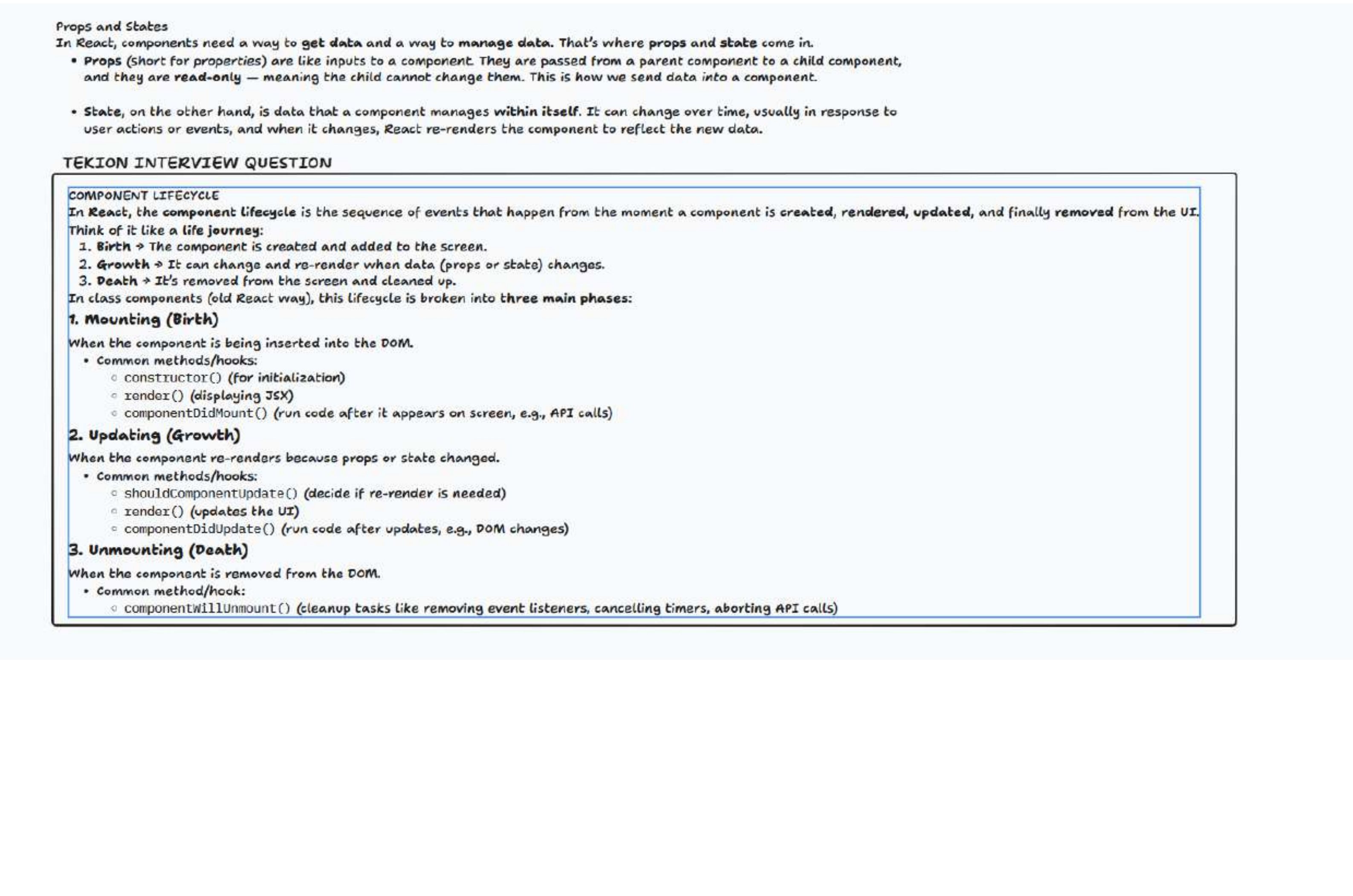
What is the Virtual DOM?(IMP)

The Virtual DOM is like a **copy of the real web page stored in memory**.
When something changes in your app (like a counter increasing), React updates the virtual copy first, **figures out what exactly changed**, and then updates only that part in the real page — instead of reloading the whole page.(uses diffing process to see changes between virtaul copy and og copy).
3)**Unidirectional Data Flow** – Data flows from parent to child, making it easier to debug and predict behavior.
4)**JSX (JavaScript XML)** – Lets you write HTML-like syntax inside JavaScript, making UI code easier to read.
-->JSX (**JavaScript XML**) is a special syntax that lets you write **HTML-like code inside JavaScript**.
-->It's like mixing HTML and JavaScript together so you can create UI elements in a more readable way — and React will turn that JSX into normal JavaScript code that the browser understands.



```jsx
import React from "react";

function App() {
  const name = "Dhanush";
  const age = 22;

  return (
    <div>
      <h1>Hello, {name}!</h1>  {/* Using JavaScript inside {} */}
      <p>You are {age} years old.</p>
      <button onClick={() => alert("Button clicked!")}>
        Click Me
      </button>
    </div>
  );
}

export default App;
```

Rendering HTML content inside a js syntax this is jsx



Props and States

In React, components need a way to **get data** and a way to **manage data.** That's where **props** and **state** come in.

- **Props** (short for *properties*) are like inputs to a component. They are passed from a parent component to a child component, and they are **read-only** — meaning the child cannot change them. This is how we send data into a component.

- **State**, on the other hand, is data that a component manages **within itself**. It can change over time, usually in response to user actions or events, and when it changes, React re-renders the component to reflect the new data.

TEKION INTERVIEW QUESTION

COMPONENT LIFECYCLE

In **React**, the **component lifecycle** is the sequence of events that happen from the moment a component is **created**, **rendered**, **updated**, and finally **removed** from the UI.

Think of it like a **life journey**:

1. **Birth** → The component is created and added to the screen.
2. **Growth** → It can change and re-render when data (props or state) changes.
3. **Death** → It's removed from the screen and cleaned up.

In class components (old React way), this lifecycle is broken into **three main phases**:

1. Mounting (Birth)

When the component is being inserted into the DOM.

- Common methods/hooks:
  - `constructor()` *(for initialization)*
  - `render()` *(displaying JSX)*
  - `componentDidMount()` *(run code after it appears on screen, e.g., API calls)*

2. Updating (Growth)

When the component re-renders because props or state changed.

- Common methods/hooks:
  - `shouldComponentUpdate()` *(decide if re-render is needed)*
  - `render()` *(updates the UI)*
  - `componentDidUpdate()` *(run code after updates, e.g., DOM changes)*

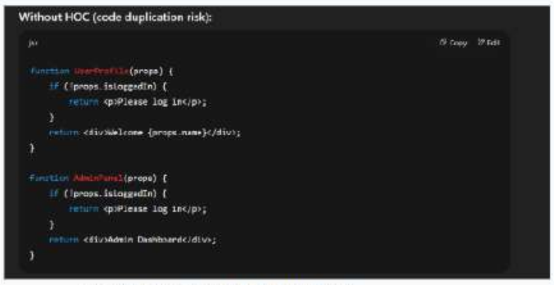
3. Unmounting (Death)

When the component is removed from the DOM.

- Common method/hook:
  - `componentWillUnmount()` *(cleanup tasks like removing event listeners, cancelling timers, aborting API calls)*

HIGHER ORDER COMPONENT --> it is actually a function not a component

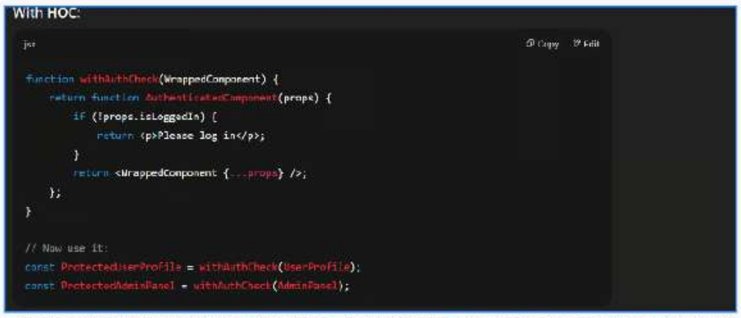
It's actually a **function** that takes a component as an argument and **returns a new component** with some added features or logic.



Without HOC (code duplication risk):

```jsx
function UserProfile(props) {
    if (!props.isLoggedIn) {
        return <p>Please log in</p>;
    }
    return <div>Welcome {props.name}</div>;
}

function AdminPanel(props) {
    if (!props.isLoggedIn) {
        return <p>Please log in</p>;
    }
    return <div>Admin Dashboard</div>;
}
```

without HOC code duplication



With HOC:

```jsx
function withAuthCheck(WrappedComponent) {
    return function AuthenticatedComponent(props) {
        if (!props.isLoggedIn) {
            return <p>Please log in</p>;
        }
        return <WrappedComponent {...props} />;
    };
}

// Now use it:
const ProtectedUserProfile = withAuthCheck(UserProfile);
const ProtectedAdminPanel = withAuthCheck(AdminPanel);
```

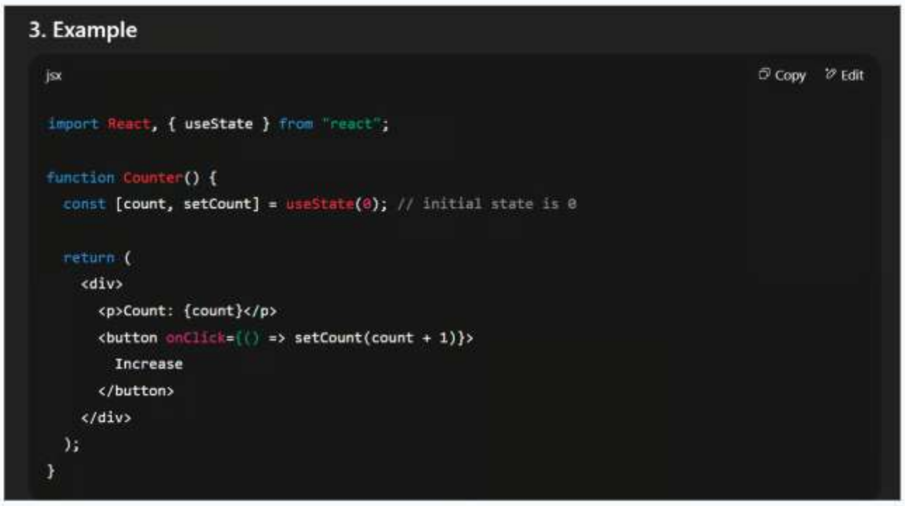
here withAuthCheck is a function , for that function we are passing different components as arguments that is what is higher order component

HOOKS

-->Hooks are **special functions** in React that let you **"hook into"** (access) React's built-in features — like **state, lifecycle methods,** or **context** — **inside functional components.**
-->Before Hooks, these features were only available in **class components.**
-->Hooks gave functional components the same powers.

1) USE STATE HOOK
`useState` is a **React Hook** that lets you add **state** to functional components.
It returns two things:

1. **The current state value.**
2. **A function to update that state.**



3. Example

```jsx
import React, { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0); // initial state is 0

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

NOTE : "Only call Hooks at the top level of your React function (not inside loops, conditions, or nested functions)."

2) USE EFFECT HOOK

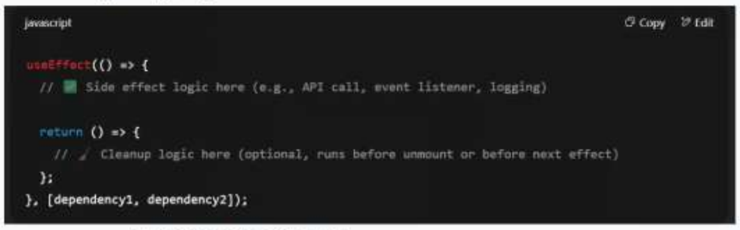
-->Before hooks, we did side effects inside **lifecycle methods** in class components:

- `componentDidMount` → *run effect after first render*
- `componentDidUpdate` → *run effect after updates*
- `componentWillUnmount` → *cleanup before removal*

`useEffect` is the **functional component way** of doing the same

-->`useEffect` lets you **run side effects** in a React component after it renders.
Side effects = any code that affects something outside the function's scope, like:

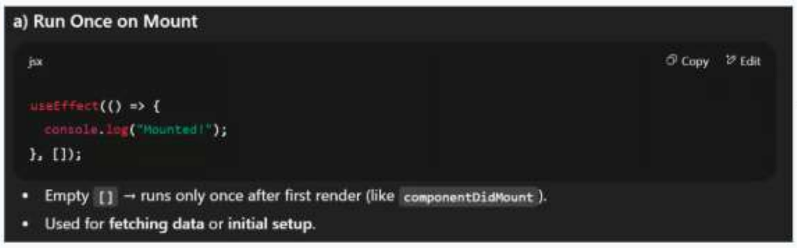
- Fetching data from an API
- Setting up a subscription
- Changing the DOM manually
- Starting/stopping timers



```javascript
useEffect(() => {
  // ✅ Side effect logic here (e.g., API call, event listener, logging)

  return () => {
    // 🧹 Cleanup logic here (optional, runs before unmount or before next effect)
  };
}, [dependency1, dependency2]);
```

USE EFFECT SYNTAX

3 main usage patterns for use effect hook

1) 

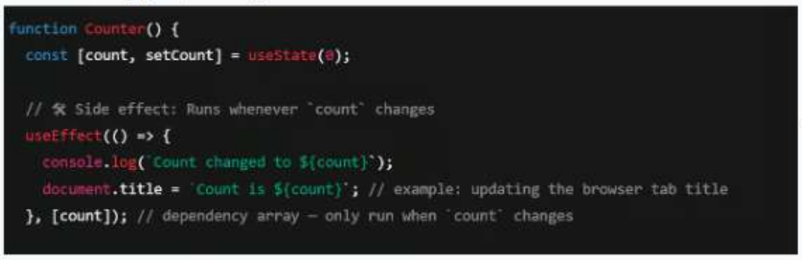
a) Run Once on Mount

```jsx
useEffect(() => {
  console.log("Mounted!");
}, []);
```

- Empty `[]` → runs only once after first render (like `componentDidMount`).
- Used for **fetching data** or **initial setup**.

→ enables componentdidmount()
funtionality as we use empty dependency array []

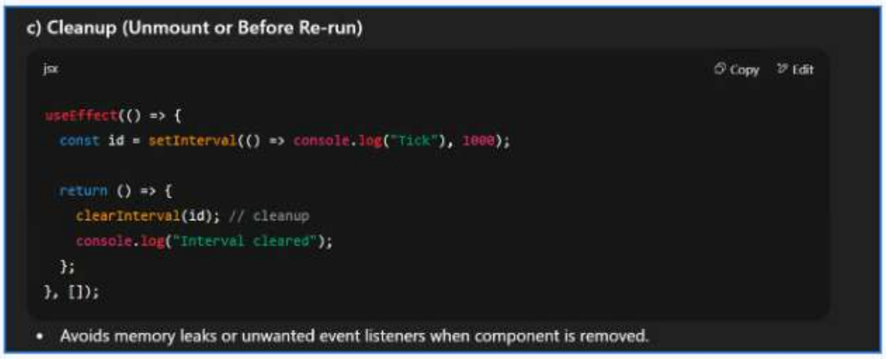
2) Run on state / prop change



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

  // 🛠 Side effect: Runs whenever `count` changes
  useEffect(() => {
    console.log(`Count changed to ${count}`);
    document.title = `Count is ${count}`; // example: updating the browser tab title
  }, [count]); // dependency array — only run when `count` changes
```

→ we are truing to update the title of document(side effect) on every count state change

3)



c) Cleanup (Unmount or Before Re-run)

```jsx
useEffect(() => {
  const id = setInterval(() => console.log("Tick"), 1000);

  return () => {
    clearInterval(id); // cleanup
    console.log("Interval cleared");
  };
}, []);
```

- Avoids memory leaks or unwanted event listeners when component is removed.

3) Use Context hook

useContext why..? — Prop Drilling Solution

What it is

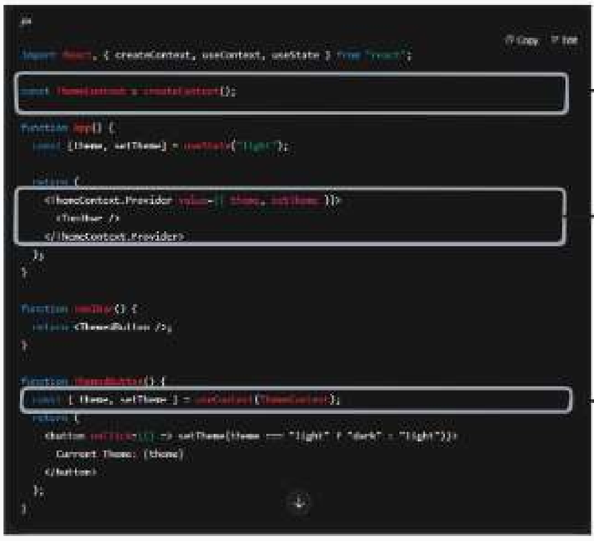
A React hook that lets you consume data from a Context object without passing props manually through multiple component levels.

Why it's used

To avoid **prop drilling** — when you pass the same prop through many layers just to reach a deeply nested component.

When to use

- Theme toggling (light/dark mode)
- User authentication state
- Global language settings



```jsx
import React, { createContext, useContext, useState } from "react";

const ThemeContext = createContext();

function App() {
  const [theme, setTheme] = useState("light");

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar() {
  return <ThemedButton />;
}

function ThemedButton() {
  const { theme, setTheme } = useContext(ThemeContext);
  return (
    <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
      Current Theme: {theme}
    </button>
  );
}
```

--> uses a publisher subscriber model to work

$\longrightarrow$ 1) createing a context

$\longrightarrow$ 2) publishing the context to a component called tool bar

$\longrightarrow$ 3) subscribing the context in toolbar component

4) useRef — DOM Access & Persistent Values Text

What it is

A hook to store a **mutable reference** that doesn't cause re-renders when updated.

Why it's used

- To directly access/manipulate a DOM element.
- To store values across renders without triggering re-render.

When to use

- Focusing an input field automatically
- Storing previous props/state
- Storing timers or IDs

4) useRef — DOM Access & Persistent Values Text

What it is

A hook to store a **mutable reference** that doesn't cause re-renders when updated.

Why it's used

- To directly access/manipulate a DOM element.
- To store values across renders without triggering re-render.

When to use

- Focusing an input field automatically
- Storing previous props/state
- Storing timers or IDs

5) useCallback — Function Memoization

What it is

A hook that memoizes a function definition so it's not recreated on every render.

Why it's used

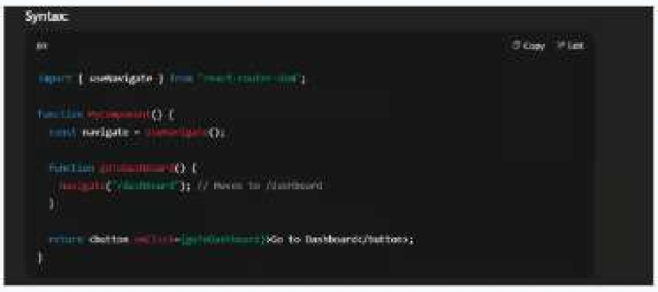
When passing functions to child components (especially `React.memo` ones), recreating the function causes unnecessary re-renders.

When to use

- Passing event handlers to memoized children
- Stable function references in dependency arrays

6) Use navigate hook

--> for programmatic navigation

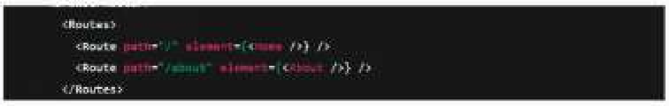

Syntax:

```jsx
import { useNavigate } from "react-router-dom";

function MyComponent() {
  const navigate = useNavigate();

  function goToDashboard() {
    navigate("/dashboard"); // Moves to /dashboard
  }

  return <button onClick={goToDashboard}>Go to Dashboard</button>;
}
```

1)import
2) create a navigator

3) use it to route programatically

Note



```jsx
<Routes>
  <Route path="/" element={<Home />} />
  <Route path="/about" element={<About />} />
</Routes>
```

-->defining route paths for components