**Render**

* **Render** means displaying or updating the **UI (User Interface)** of a web page or component.
* In frameworks like React, rendering happens **when the component's state or props change**, and React updates the DOM accordingly.
* **Rendering only updates parts of the page**, without refreshing the whole page.
* It’s usually fast and smooth, because only necessary parts are updated.

**Example:**

* React calls the render method to update what’s shown on the screen after a state change.
* When you type in a form input, React might re-render just that component.

**Reload**

* **Reload (or refresh)** means **reloading the entire web page** from the server (or cache).
* This is like pressing the browser's refresh button.
* It **restarts the whole page load process**: fetching HTML, CSS, JS, and rendering everything again from scratch.
* Usually slower and interrupts any ongoing interactions.

**Example:**

* Pressing F5 or clicking the refresh icon on your browser reloads the entire page.
* You lose unsaved state or form inputs unless explicitly handled.

**Memoized function :**

memoized funtion like that if the parent component pass the data into child component and the parent component changdes but passing data is not changed then child component will not render is it right?

**ChatGPT said:**

**Yes, you're almost right, and you're thinking in the right direction! Let's clarify it fully:**

**🧠 Goal:**

**You want the child component to not re-render when the parent re-renders, as long as the props passed to the child haven't changed.**

**✅ To achieve that:**

**You use React.memo() to memoize the child component.**

**const Child = React.memo(function Child({ data }) {**

**console.log("Child rendered");**

**return <div>{data}</div>;**

**});**

* **Now, React will skip re-rendering <Child /> if the data prop hasn’t changed by reference.**
* **If the parent re-renders but passes the same data, the child won’t re-render. ✔️**

**Mounting**

In **React**, **"mounting"** refers to the **phase when a React component is created and inserted into the DOM (Document Object Model)** for the first time.

-Mounting is the **first time your component appears on the screen**.

When a component is **mounted**:

1. **Constructor (optional)** is called → for setting up state and binding methods.
2. **render()** method is called → returns the JSX to be shown.
3. **React updates the DOM** with the JSX returned.
4. **componentDidMount()** (for class components) or useEffect(() => {}, []) (for functional components) is called → this is where you do things like API calls, setting timers, etc.

**Lifecycle :**

constructor() ➝ render() ➝ componentDidMount()

**Example:**

import React, { useEffect } from 'react';

function MyComponent() {

useEffect(() => {

console.log('Component mounted!');

}, []);

return <div>Hello!</div>;

}

**## React allows unidirectional data flow but we can send data from child to parent using function.**

**Example: Passing Data from Child to Parent**

**✅ Parent Component**

function Parent() {

const handleDataFromChild = (data) => {

console.log("Received from child:", data);

};

return <Child sendData={handleDataFromChild} />;

}

**✅ Child Component**

function Child({ sendData }) {

const handleClick = () => {

const message = "Hello Parent!";

sendData(message); // sending data to parent

};

return (

<button onClick={handleClick}>

Send Data to Parent

</button>

);

}

**🔍 Here's how it works conceptually:**

1. Parent defines a function → handleData(data)
2. Parent passes it to child → <Child sendData={handleData} />
3. Child calls sendData('some data') → this **invokes the parent’s function**
4. The parent **receives the data** and updates its own state if needed

🧠 React still considers this **unidirectional**, because:

* The parent **controls** what happens to the data
* The child is only **notifying**, not changing parent state directly

**State**: In React, state is an object that stores dynamic data for a component — it determines how that component behaves and what it renders.

**State is like a memory for your component.**  
It stores values that can **change over time**, like input fields, counters, or fetched data.

**HOOK :**

Hooks are special functions in React that allow functional components to use features like state, lifecycle methods, context, and more — without needing to use class components.

**Simple Version:**

Hooks let you **“hook into”** React features (like state or side effects) **inside functional components**.

**📝 Summary Table**

| **Hook** | **What it does** | **Use Case Example** |
| --- | --- | --- |
| useState | Add dynamic state | Counter, form input |
| useEffect | Handle side effects (like API calls) | Fetch data, timers, lifecycle |
| useRef | Access DOM or store stable values | Focus input, track value without re-render |
| useContext | Use global/shared state | Theme, Auth data |
| useCallback | Memoize functions | Prevent child re-renders |
| useMemo | Cache expensive calculations | Avoid recalculating on every render |
| Custom Hook | Reuse logic across components | useCounter, useAuth, useForm |

**1. useState – Add state to functional components**

**📘 Purpose:**

Track and update values that change (like counters, inputs).

**🧪 Example:**

import React, { useState } from 'react';

function Counter() {

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

return (

<div>

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

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

</div>

);

}

**2. useEffect – Run side effects**

**📘 Purpose:**

Perform tasks like fetching data, timers, logging, etc.

**🧪 Example:**

import React, { useEffect } from 'react';

function Timer() {

useEffect(() => {

const interval = setInterval(() => {

console.log('Running...');

}, 1000);

return () => clearInterval(interval); // Cleanup

}, []);

return <p>Check your console!</p>;

}

**useRef – Reference DOM or store persistent values**

**📘 Purpose:**

* Control DOM elements
* Store values without causing re-renders

**🧪 Example:**

import React, { useRef } from 'react';

function InputFocus() {

const inputRef = useRef(null);

const handleFocus = () => {

inputRef.current.focus(); // Focuses the input

};

return (

<>

<input ref={inputRef} type="text" />

<button onClick={handleFocus}>Focus Input</button>

</>

);

}

**useContext – Consume global data from React Context**

**📘 Purpose:**

Avoid prop-drilling by using global state/data

**🧪 Example:**

import React, { createContext, useContext } from 'react';

const ThemeContext = createContext("light");

function Child() {

const theme = useContext(ThemeContext);

return <p>Current Theme: {theme}</p>;

}

function App() {

return (

<ThemeContext.Provider value="dark">

<Child />

</ThemeContext.Provider>

);

}

**✅ 5. useMemo – Memoize computed values**

**📘 Purpose:**

Optimize performance for **expensive calculations**

useMemo is used to **memoize a calculated value**, so that the calculation is **only done when its dependencies change**.

**🧪 Example:**

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

function ExpensiveCalculation({ number }) {

const result = useMemo(() => {

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

let total = 0;

for (let i = 0; i < 1e7; i++) total += number;

return total;

}, [number]);

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

}

function App() {

const [num, setNum] = useState(1);

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

return (

<>

<input

type="number"

value={num}

onChange={(e) => setNum(parseInt(e.target.value))}

/>

<ExpensiveCalculation number={num} />

<button onClick={() => setCount(count + 1)}>Re-render</button>

</>

);

}

**Q:** when useMemo runs only when dependency changes then why we not use the useEffect?

**Ans:** because useEffect does not return value it is only for fetch the api and for timers.

**✅ 6. useCallback – Memoize functions**

**📘 Purpose:**

Prevent unnecessary function recreation on re-renders.

**🧪 Example:**

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

function Button({ handleClick }) {

console.log("Button rendered");

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

}

function App() {

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

const memoizedClick = useCallback(() => {

console.log("Button clicked");

}, []); // will not change unless dependencies change

return (

<>

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

<button onClick={() => setCount(c => c + 1)}>Update Count</button>

<Button handleClick={memoizedClick} />

</>

);

}

**✅ 7. Custom Hooks – Reuse logic across components**

**📘 Purpose:**

Extract reusable logic from components.

**🧪 Example: useCounter Hook**

import { useState } from 'react';

function useCounter(initialValue = 0) {

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

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

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

return { count, increment, decrement };

}

**🔁 Using useCounter in a component:**

import React from 'react';

function CounterApp() {

const { count, increment, decrement } = useCounter(10);

return (

<>

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

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

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

</>

);

}

**🔹 What is Optional Chaining?**

Optional chaining (?.) allows you to safely access deeply nested object properties without having to check if each level exists.

Instead of writing:

user && user.profile && user.profile.name

You can simply write:

user?.profile?.name

If user or profile is null/undefined, the expression short-circuits and returns undefined instead of throwing an error.

**🔹 1. Nullish Coalescing (??)**

Works well with optional chaining to set a **default value** only if the left side is null or undefined (not 0 or false).

<h2>{user?.name ?? "Guest User"}</h2>

👉 Difference from ||:

* || treats falsy values ("", 0, false) as empty.
* ?? only checks null and undefined.

**NOTE:** ?? → nullish coalescing (default values).