**Hooks in ReactJS**

In ReactJS, hooks are functions that allow you to use state and other React features in functional components. They were introduced in React version 16.8 to enable functional components to have local component state, side effects, and other features that were previously only available in class components.

Some commonly used hooks in React are:

1. `useState`: Allows functional components to have local state.

2. `useEffect`: Enables performing side effects in functional components (e.g., data fetching, subscriptions, manual DOM manipulations).

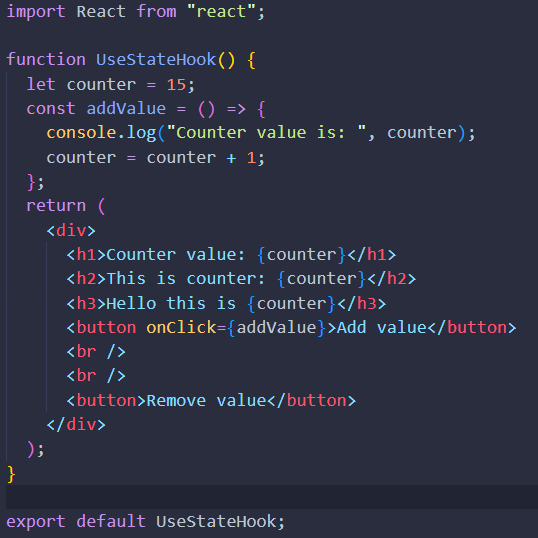
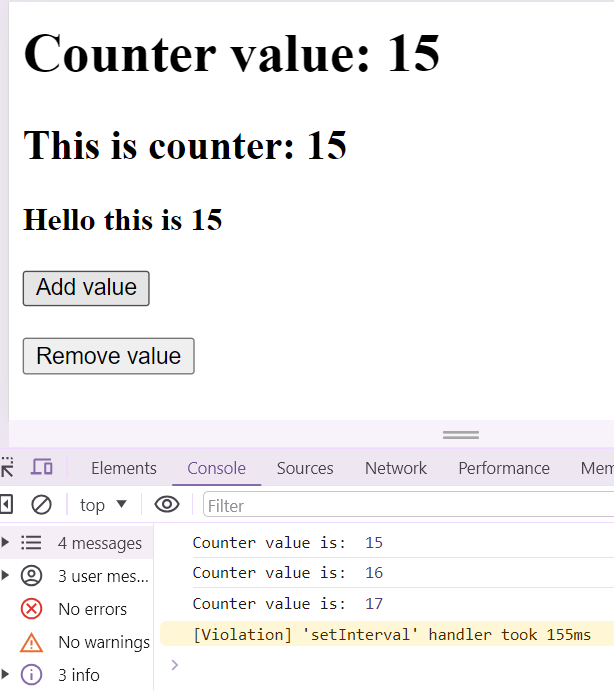
3. `useContext`: Provides a way to pass data through the component tree without having to pass props down manually at every level.

4. `useReducer`: An alternative to `useState` for more complex state logic.

5. `useMemo` and `useCallback`: Memoize values and functions to optimize performance.

6. `useRef`: Access DOM nodes or store mutable values that persist through renders.

Using hooks can make your code more concise, readable, and easier to maintain compared to class components.

**Problem Statement**

Here counter variable is at multiple places and if we want to update its value using javascript then we have to make a lot of variables and then take them using docummet.querySelector().

So, to make this complicated thing easy react give us a hook and say that UI updation ka kaam mera hoga.

Jaha Jaha phr ye counter hai mai har jagha usse update kr dunga.

UI Updation ko react control krta hai.

It can be done by react using useState Hook.

import React from "react";

iss line ko nhi bhi likhunga toh bhi chlega kyuki babel isko handle kr lega.

**useState Hook**

**useState**is React Hook that allows you to add state to a [functional component](https://blog.logrocket.com/fundamentals-functional-programming-react/). It returns an array with two values: the current state and a function to update it. The Hook takes an initial state value as an argument and returns an updated state value whenever the setter function is called. It can be used like this:

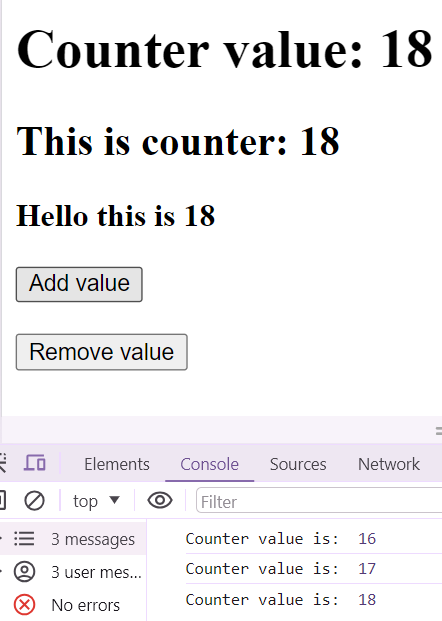
**const [state, setState] = useState(initialValue);**

Here, the **initialValue** is the value you want to start with, and state is the current state value that can be used in your component. The **setState** function can be used to update the state, triggering a re-render of your component.

**useState** can store any type of value, whereas the state in a class component is limited to being an object. This includes primitive data types like string, number, and Boolean, as well as complex data types such as array, object, and function. It can even cover custom data types like class instances.

We can import useState Hook like this

Import { useState } from “react”;

**Example of useState Hook**

Now we can clearly see that using this hook we have update the variable value in all the places where the variable is.

We can use useState with arrays and with objects also.

**Virtual DOM**

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

root.render(

  <React.StrictMode>

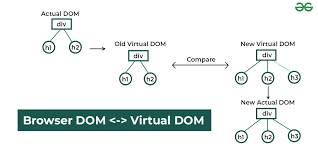
    <App />

  </React.StrictMode>

);

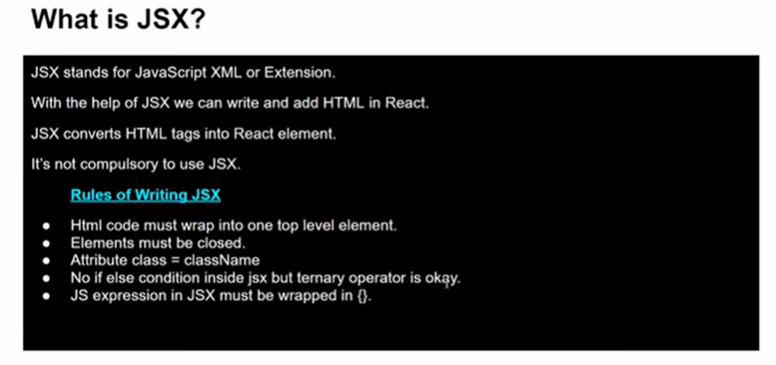
Here, the createRoot() function creates a DOM-like structure behind the scenes just like the browser has its DOM. This is done because it compares the original DOM with its own DOM and then updates the elements that have actually been updated in the UI. However, our browser removes the entire DOM and then repaints the entire DOM structure (the web structure is being prepared again), which is what we call a page reload.

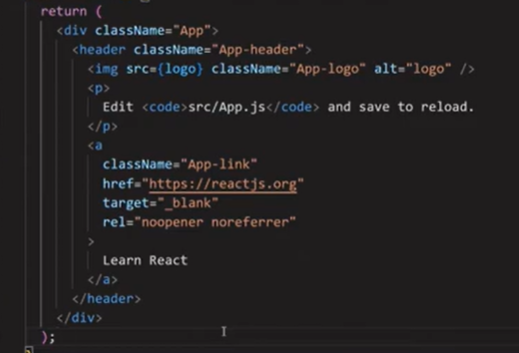
In contrast, with virtual DOM, you keep track of changes and simply remove the values that have changed from the DOM and reapply them.



1. **Real DOM vs. Virtual DOM:**
   * **Real DOM:** The DOM represents the structure of an HTML document and provides an interface for interacting with it. The Real DOM is the actual browser's representation of the DOM structure, and every time there is a change in the state of a React component, the Real DOM gets updated.
   * **Virtual DOM:** The Virtual DOM is a lightweight copy of the Real DOM. Instead of updating the Real DOM directly, React first updates the Virtual DOM. This allows React to perform operations on the Virtual DOM, which is faster and more efficient than directly manipulating the Real DOM.
2. **Reconciliation:**
   * When the state of a React component changes, a new Virtual DOM representation is created.
   * React then compares this new Virtual DOM with the previous one (diffing).
   * The difference between the new and old Virtual DOM is calculated efficiently to determine the minimal number of updates needed.
3. **Updating the Real DOM:**
   * Once the differences are identified, React updates only the necessary parts of the Real DOM.
   * This selective update is more efficient than updating the entire Real DOM, especially in large and complex applications.
4. **Benefits:**
   * **Performance:** Manipulating the Virtual DOM is faster than directly manipulating the Real DOM because the Virtual DOM is an in-memory representation and doesn't involve interacting with the actual browser's rendering engine until necessary.
   * **Efficiency:** By calculating and applying only the necessary changes to the Real DOM, React minimizes the performance impact of updates.
5. **React Reconciliation Algorithm:**
   * React uses a reconciliation algorithm to efficiently update the Virtual DOM and determine the minimal set of changes needed to update the Real DOM.
   * It employs a diffing strategy to identify what has changed, and the algorithm attempts to optimize for performance.

[Documentation](https://github.com/acdlite/react-fiber-architecture)

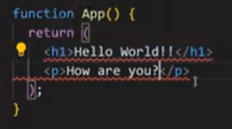
**JSX in react**

Example code of JSX

Yes we are writing HTML in a JS file and still not getting any error this is because Babel(A JS compiler) convert this jsx into pure javascript. So, when our web page see this code it see it as js not jsx. That’s why we don’t see any kind of compilation or run-time error.

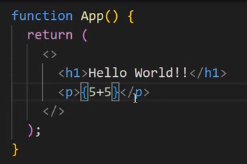
**Rules of writing JSX**

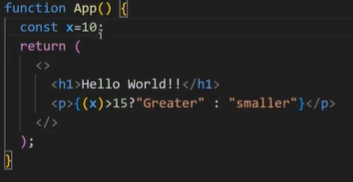
1. **HTML code must wrap into one top level element.**

**error:- JSX expression must have one parent element.**

**To solve this add all jsx into a div or we can use fragements <> </>**

1. **JS expression in JSX must be wrapped in {}.**

****

1. **No if else condition inside JSX but ternary operator is okay.**
2. **Attributes class = className**

****

**Props in React JS**

**In ReactJS, props (short for properties) are used to pass data from a parent component to a child component. Props are read-only and help make components reusable and modular. Here's an example to illustrate how props work in React:**

1. Props stands for properties.
2. Props are used to transfer data from one component to another.
3. Props are read only means you can change it’s value.
4. Props are just like function in JavaScript.



Props are read only means:- If in the Child Component before return If I try to change the props value like this props.message = “Hello” then this will give me error that Cannot assign to read only property ‘message’ of object.

**State in ReactJS**

1. The state is a built-in React object that is used to contain data or information about the component.
2. A State can be modified based on user action or network changes.
3. Every-time when the state of an object changes, React re-render the component to the browser.

Q1. What will be the output of this code?

Function App() {

const [counter, setCounter] = useState(15);

const addValue = () => {

setCounter(counter + 1);

setCounter(counter + 1);

setCounter(counter + 1);

setCounter(counter + 1);

}

return (

<button onClick={addValue}>Add Value</button>

);

}

Ans1. The output of this code appears to be 19 when clicking on the add button, but actually the answer is only 16. This discrepancy occurs due to the useState Hook. useState batches all the updates it needs to send to the UI or the variable. So, when we use setCounter(counter+1), it sees that these are the same operations and groups them together in a batch before executing them.

Now, the question is how to modify this code so that it works according to our logic and gives us 19 as output.

The setCounter function actually accepts a callback. It should be like this: setCounter(() => {})

Function App() {

const [counter, setCounter] = useState(15);

const addValue = () => {

setCounter(prevCounter => prevCounter + 1);

setCounter(prevCounter => prevCounter + 1);

setCounter(prevCounter => prevCounter + 1);

setCounter(prevCounter => prevCounter + 1);

}

return (

<button onClick={addValue}>Add Value</button>

);

}

Here, when using callbacks, the updates happen properly only after the completion of the previous one, resulting in the desired output.

**useEffect Hook**

The useEffect hook in React allows you to perform side effects in your functional components, such as data fetching or updating the DOM, after every render. React manages these side effects transparently to optimize performance. useEffect hook synchronizes your components with external systems or handles effects based on dependencies provided. It is a key tool in managing lifecycle events in functional components. When working with the useEffect hook, it's essential to understand when to use it, how to properly manage dependencies, and potential pitfalls to avoid for optimal performance.

The useEffect Hook allows you to perform side effects in your components. Some examples of side effects are: fetching data, directly updating the DOM, and timers. useEffect accepts two arguments. The second argument is optional.

useEffect(<function>, <dependency>)

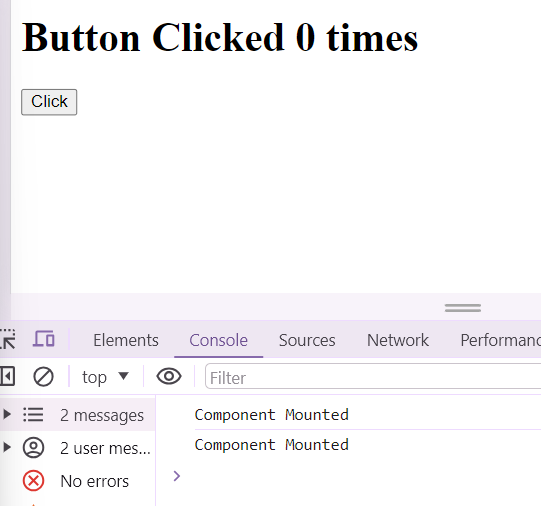
This hook runs on every render but there is also a way of using a dependency array using which we can control the effect of rendering

**How does it work?**

* You call useEffect with a callback function that contains the side effect logic.
* By default, this function runs after every render of the component.
* You can optionally provide a dependency array as the second argument.
* The effect will only run again if any of the values in the dependency array change.

useEffect is a function that accepts two arguments one is a callback function and second is the dependencies array.

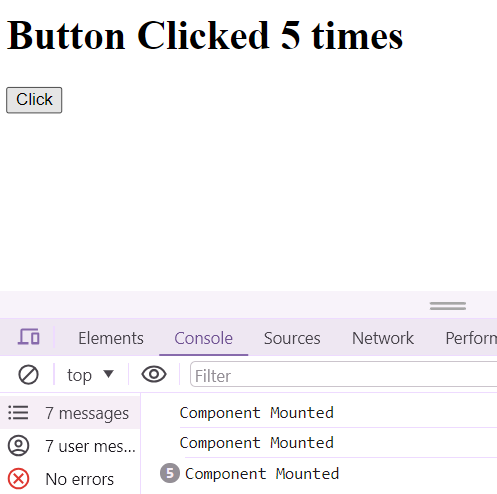
When the web page loads then our useEffect function get execute and on every render also.



Initially when the web page loads the useEffect hook is called. We can see two console message.

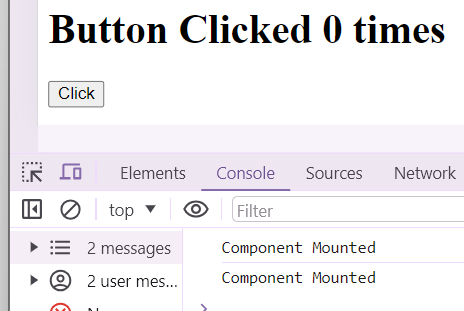
In this code, you are seeing "Component Mounted" being logged twice because the useEffect hook you are using runs after every render by default, including the initial render. Since you didn't specify any dependencies for the useEffect hook, it runs after the initial render and after every update.

To log "Component Mounted" only once, you can pass an empty dependency array as the second argument to useEffect, like this:



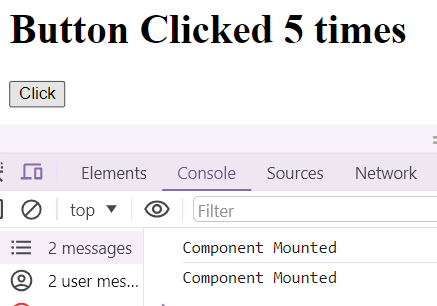
When our component is first rendered, the `useEffect` hook is called. However, whenever we update any state within our component, the `useEffect` hook is triggered again.

If I don't want my `useEffect` to run when I don't want any of my states to update, I will use the second parameter, which is the dependency array. To achieve this, I will pass an empty array as a dependency.

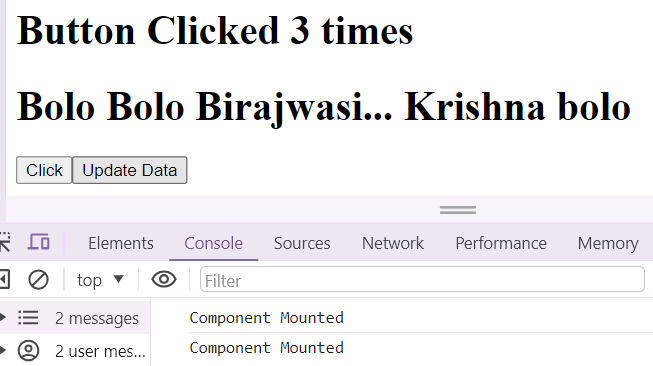


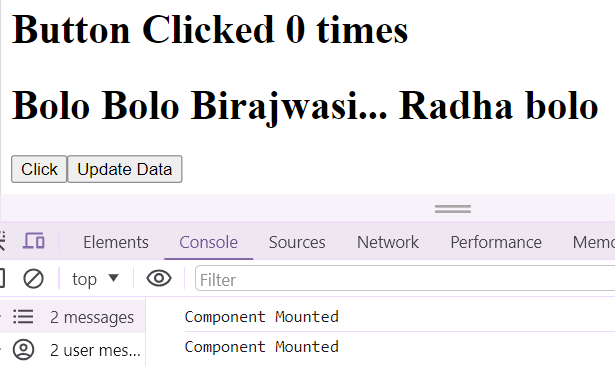
This is the output at first when button is not clicked and the component render at first time.

Now let’s see what happens when we try to update something some state in our component.

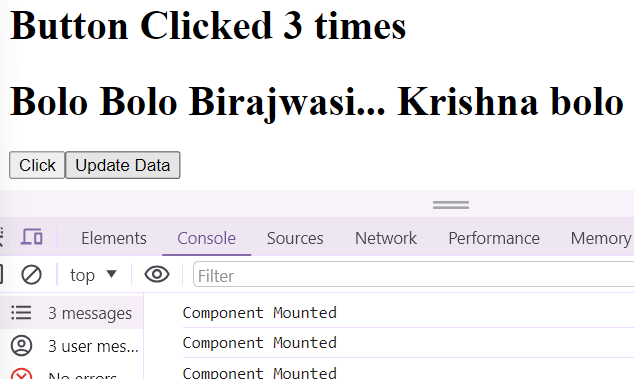
Here I clicked 5 times the button the state changed 5 time but still the useEffect hook has not been called this is because as I have passed an empty array as the dependency.

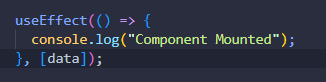
Now let’s make another state and in this case I want that useEffect hook should only call when I change a certain state and that state I can define as passing that state in useEffect hook as dependency array.

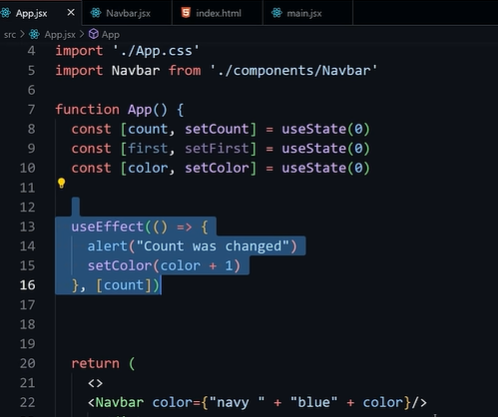


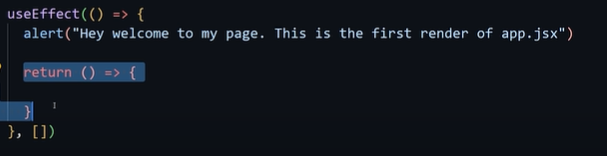
I

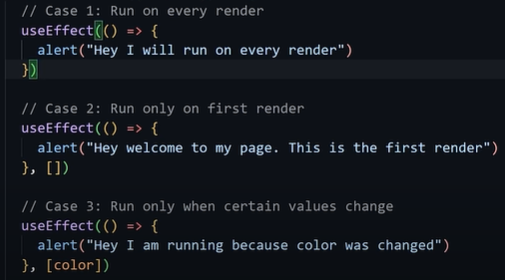
In this case I have passed empty array as dependency so useEffect is called only once not every time when the both two states changes.

Now I want that whenever my data get update then my useEffect should get run. To do this so we need to pass this state of data in the dependency array.



Whenever the button is clicked and count is changed then ye pura return wala re render hoga. Kyuki ek state change hui hai and jab bhi ek state change hoti hai toh pure component ko rerender krta hai react. So my navbar component will also get rerender.

When count will change color will get change. And useeffect for 3rd case will be called.



This return () function is a clean up function passed in useeffect it will run only when the component gets unmount. And it is optional. Through conditional rendering we can unmount our component.

**useRef Hook**

useRef is a React Hook that allows you to reference a value that does not trigger a re-render when updated. It is useful for storing mutable values between renders. When using useRef, you get a mutable ref object with a .current property that can be initialized with any value. This value persists between renders. [External Source](https://medium.com/@zahidbashirkhan/react-useref-use-cases-with-examples-d7680d48a6e1)

useRef is a React Hook that lets you reference a value that’s not needed for rendering.

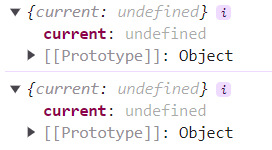
const ref = useRef(initialValue)

It’s particularly common to use a ref to manipulate the [DOM.](https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API) React has built-in support for this.

Mostly used to manipulate the DOM.

Call useRef like this :- const refElement = useRef();

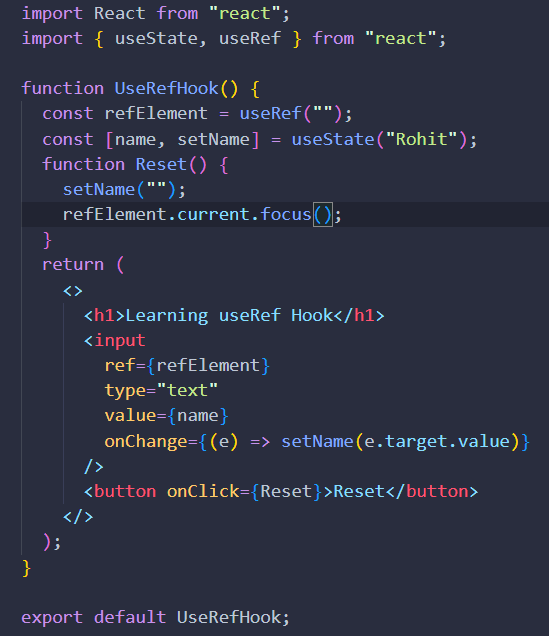
useRef hum ye return krta hai and iss current key ki help se hum apne DOM ko manipulate kr skte hai.



Jo bhi value hum apni input field mai dalenge wo yaha current.value mai save hogi.



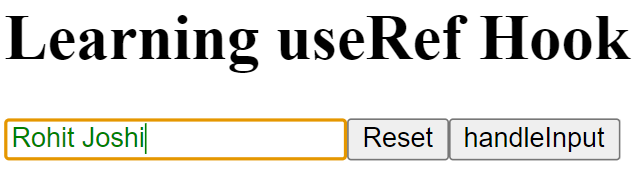
Here when I click on this input box to write I get focus on it but when I write something and then click on the reset button then my focus get away from the input box. So, I want to keep my focus on the input box. To do this I will use useRef Hook.



This is the code where we are manipulating the DOM with the help of useRef.





When I will click on HandleInput button the color of the text in the input button should change to green.

When component re-render then normal variables will also get re-render. Due to this they will not able to persist their value. To solve this issue we can use useRef hook.

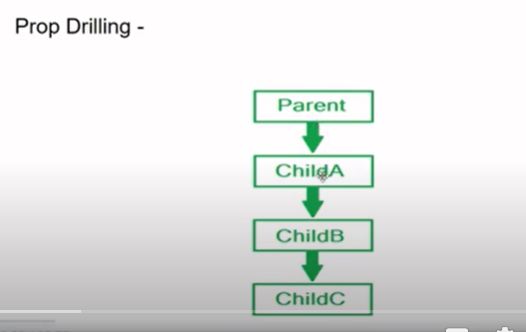
This a value will persist across every render.

We can also use this hook for DOM manipulation also.

React Strict mode k karan hume cheeze 2 baar render hote huye dikhti hai

NOTE: - Jab bhi hum iss ref ki value ko change krte hai toh useEffect trigger nhi hota hai. State ko change kia toh component re-render hota hai but ref ko change krne phr esa nhi hota.

**Prop Drilling**



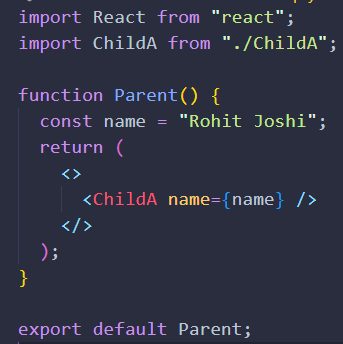
**Here I different components.**

**I have a Parent Component inside it I have ChildA component. Inside Chlid-A I have Chlid-B component. Inside Chlid-B I have Child-C Component.**

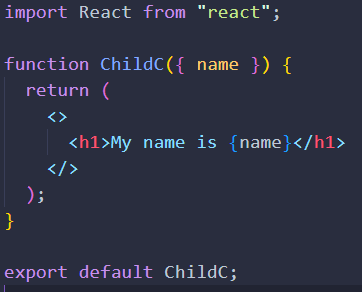
Let’s say I have a state/data/function in parent component and I want to send it to the Child-C component.

So, for doing this I need to pass the data through All the nested components then only it will reach to the Child-C.

The issue is We are sending the data in unnecessary components and to solve this issue we will use the concept of Context API. In Context API there we be no need of passing the data in Child-A, Child-B we can directly pass the data from Parent to Child-C.



Parent Chlid-A



Child-B Child-C

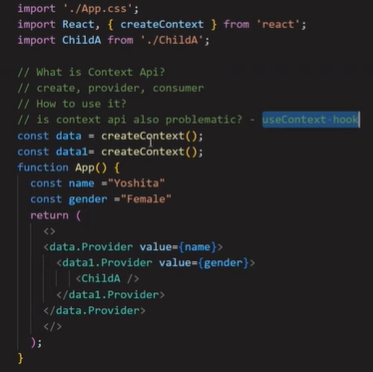


Output

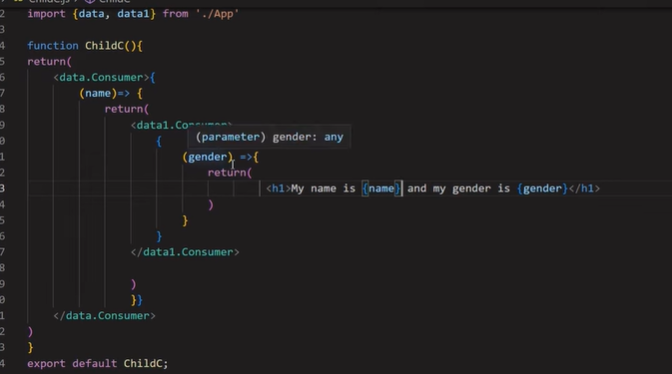
**Context-API**

The Context API allows you to create a global state that can be accessed by any component within the component tree without explicitly passing the props through each level. It consists of three main parts:

1. **Create Context**: This is where you define your context. It could be a simple JavaScript object or any other data type.
2. **Provide Context**: Wrap the part of your component tree that needs access to the context with a **Context.Provider** component. This provider accepts a **value** prop, which is the data you want to share.
3. **Consume Context**: Use the **Context.Consumer** component or the **useContext** hook within the child components to access the data provided by the context.

Here's an example of how you might use the Context API:





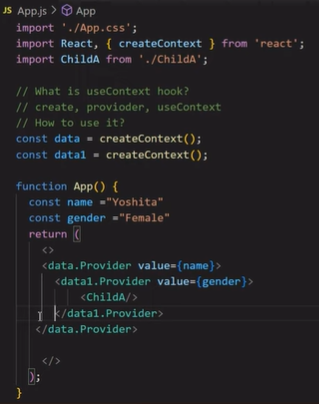
Context API itself become more problematic as we need to create nested function for each value we want to use and to solve this issue we will use useContext Hook.

**useContext Hook**

The Context API in React is a way to share data between components without having to pass props through every level of the component tree. It provides a way to pass data through the component tree without having to pass props down manually at every level.

`useContext` is a hook that allows you to subscribe to React context without introducing nesting. It lets you access the value of the context provided by the `Context.Provider` component closest to the current component in the tree. This hook is particularly useful for consuming context values in functional components.

The **useContext** hook is a React hook that allows you to consume context without using a **Context.Consumer** component. It takes a context object (the value returned from **React.createContext**) and returns the current context value for that context.







**useMemo Hook**

The useMemo hook in React is used to memoize expensive calculations so that they are only recomputed when the dependencies have changed. It helps optimize performance by caching the result of a function and only recomputing it when one of the dependencies has changed.

  useMemo(() => first, [second]);

first is my function/work which I want to perform and second is the dependency array and my useMemo will only get triggerd and something that is mentioned in the dependency array get changed.



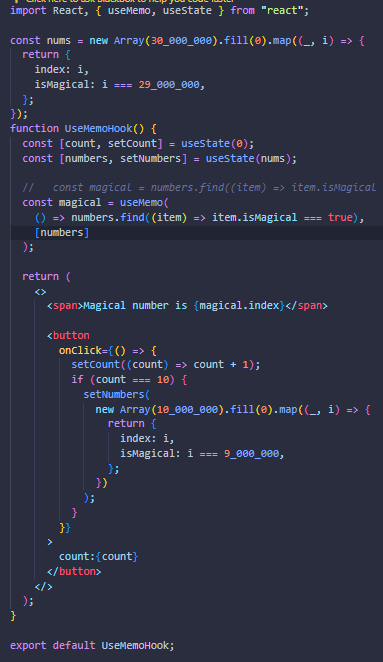
In this code I am searching magical number and I am doing 30 million time the same calculation which is quiet heavy task and specially when it is happening again and again on the change of the Button which is count. This 30M calculation is calculated again and again….which will hamper the website and it can crash also. So, to remove this issue we will use useMemo Hook.

useMemo will save the value of magical number and whenever it require it will use it. That means b/w re-renders the 30M computation will not be done. It will memorize it. And will make your website responsive.

It can happen that our magical number get changed so then again we have to calculated that magical number and need to store so that it can use to memorize the code. And that is where my dependency array role comes in play.

We will pass numbers in my dependency, so whenever my magical number will get changed the useMemo hook will compute it value again.



This will change my magical number from 29M to 9M when count becomes 10



