WEEK 6

As we know we can't return multiple siblings in react component . We need to return the top level div or any component. inside which we have multiple siblings .

```
src > App.jsx > App
function App() {

  retu
    <Header title="Yash1"></Header>
    <Header title="Yash2"></Header>

  }

function Header(props){
  return <div>
    {props.title}
  </div>

}

export default App
```

JSX expressions must have one parent element. ts(2657)
View Problem (⌥F8) Quick Fix... (⌘.)

Why is that ?

Because it makes it easy to do reconciliation .

Reconciliation

In the context of React, reconciliation refers to the process of comparing the current state of the React virtual DOM with the previous state and determining the most efficient way to update the actual DOM to reflect the changes. It's a key optimization technique used by React to minimize the number of DOM manipulations and improve performance.

When React components update their state or receive new props, React constructs a new virtual DOM tree representing the updated UI. React then compares this new virtual DOM tree with the previous one using a diffing algorithm. The goal of this comparison is to identify the minimum set of changes required to update the actual DOM.

Now to understand these lets create a react app that has -
1. a header component that takes a title As a prop and renders it inside a div .
2. The top level app component renders 2 headers .

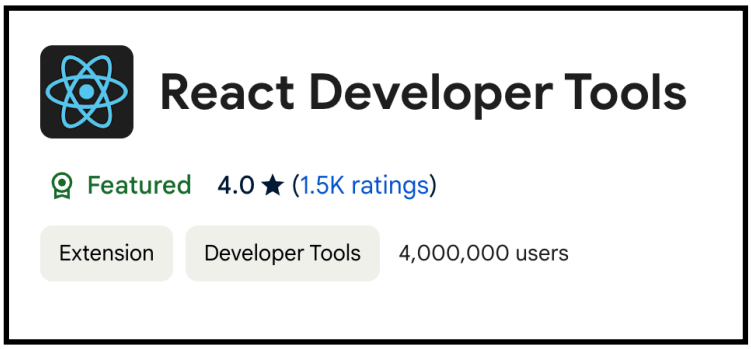
```
src > App.jsx > App
function App() {
  return (
    <div>
      <Header title="Yash1"></Header>
      <Header title="Yash2"></Header>
    </div>
  )
}
function Header(props){
  return <div>
    {props.title}
  </div>

}
export default App
```

This one uses div as parent .

```
src > App.jsx > App
function App() {
  return (
    <>
      <Header title="Yash1"></Header>
      <Header title="Yash2"></Header>
    </>
  )
}
function Header(props){
  return <div>
    {props.title}
  </div>

}
export default App
```

This is slightly better as it don't uses extra component .

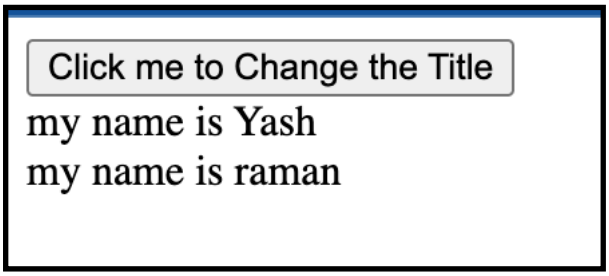
Now lets study about re - rendering . So in order to learn it first we have to download the react extension - know as react developers tools .



As we know react is used to create dynamic web contents(a website whose content changes very quickly) .

Any time such dynamic features happen such as -
1. Append something to DOM.
2. Remove Something from the DOM .
3. Or you update something inside the DOM.

The goal is to minimise the number of re renders .



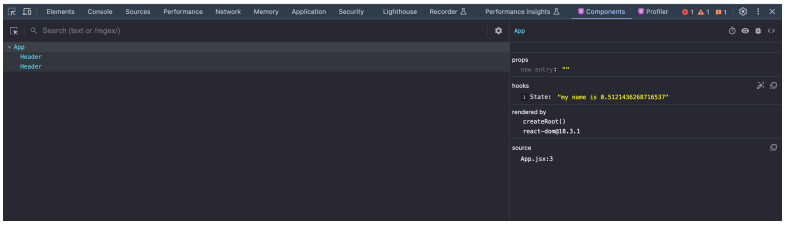
In order to understand change the previous code a little bit . Add button to it . And when we click on that button Yash changes to some random number between 0 to 1 .

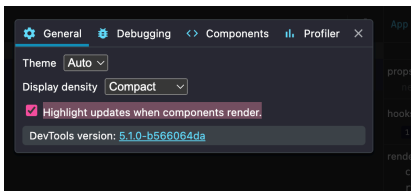
How to achieve it ?

```
src > App.jsx > ...
import { useState } from "react"
function App() {
  const [count,setCount]=useState("my name is Yash");
function updateTitle(){
  setCount("my name is " + Math.random())
}
  return (
   <>
      <button onClick={updateTitle}>Click me to Change the Title</button>
     <Header title={count}></Header>
      <Header title="my name is raman"></Header>
   </>
  )
}
function Header(props){
  return <div>
    {props.title}
  </div>

}
export default App
```

Now to see what is re rendered on the screen we have to enable react developers tools .

Components -> App -> Setting icon ->





Enable it .

Now when you interact with screen you will see boxes. Which means these things are re -rendered.

When you see it you will see instead of re rendering only one title . It re renders all the title .(all the component which are on the screen or all the children's of the parent in which only one children state is changed.)

Re-Render

A re render means that

1. React did some work to calculate what all should update in this component .
2. The component actually got called (you can put a log to confirm this)
3. The inspector shows you a bounding box around the component .

It happens when

1. A state variable that is being used inside a component changes .
2. A parent component re -render triggers all children re - rendering .

You want to minimise the number of re-renders to make a highly optimal react app.
The more the components that are getting re-rendered , the worse .

Q. How can you minimise the number of re-renders in this app?

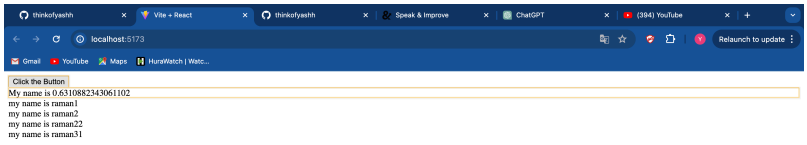
1. Pushing the state down

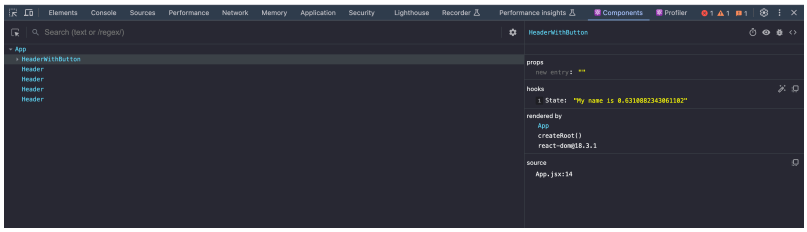
```
src > App.jsx > App
import { useState } from "react"
function App() {
  const [count,setCount]=useState("my name is Yash");
function updateTitle(){
  setCount("my name is " + Math.random())
}
  return (
   <div>
      <button onClick={updateTitle}>Click me to Change the Title</button>
     <Header title={count}></Header>
      <Header title="my name is raman"></Header>
   </div>
  )
}
function Header(props){
  return <div>
    {props.title}
  </div>

}
export default App
```

—->

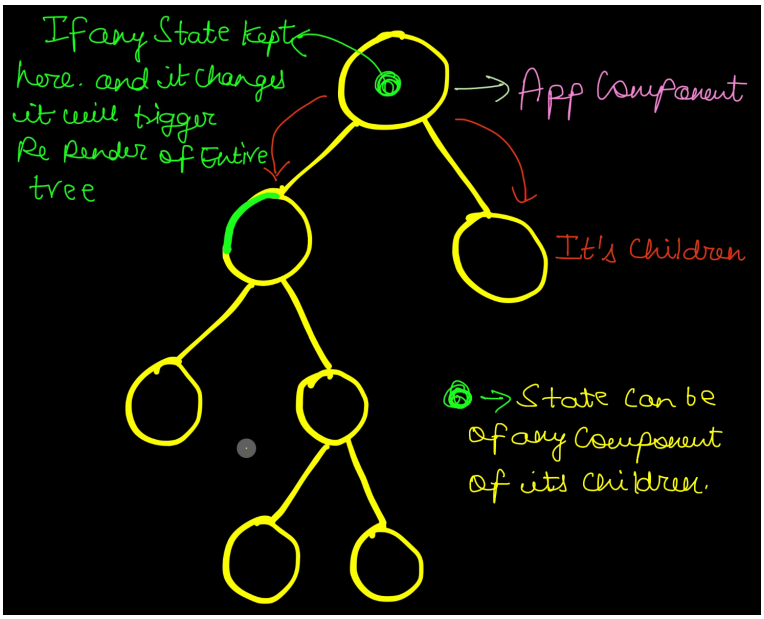
```
src > App.jsx > HeaderWithButton
import { useState } from "react"
function App() {
  return (
   <div>
      <HeaderWithButton></HeaderWithButton>
      <Header title="my name is raman1"></Header>
      <Header title="my name is raman2"></Header>
      <Header title="my name is raman22"></Header>
      <Header title="my name is raman31"></Header>
   </div>
  )
}
function HeaderWithButton(props){
  const [title,setTitle]=useState("my name is yash")

  function onClicked(){
    setTitle("My name is "+ Math.random());
  }
  return <>
  <button onClick={onClicked}>Click the Button</button>
  <Header title={title}></Header>

  </>
}
function Header(props){
  return <div>
    {props.title}
  </div>

}
export default App
```

Output - slightly optimal than before .





NOTE -



It is a thumb rule to keep the state. At the lowest common ancestors . To help in avoiding the re renders of static components or the components that doesn't change if this state variable is changes.

Alternative way to avoid builds is we can use reactmemo hook .

React memo

memo lets you skip re-rendering a component when its props are unchanged.

How to use it ?

1.Simply first Copy the function (components).

```
function Header(props){
  return <div>
    {props.title}
  </div>

}
```

2. Create a variable and use React.memo() like -

```
const Header=React.memo()
```

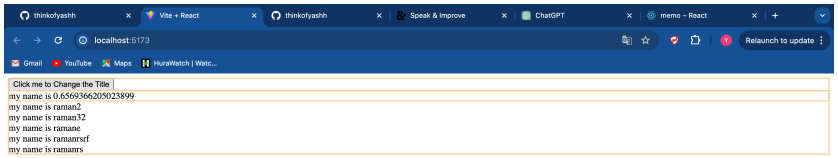
3. And paste the entire function inside this .

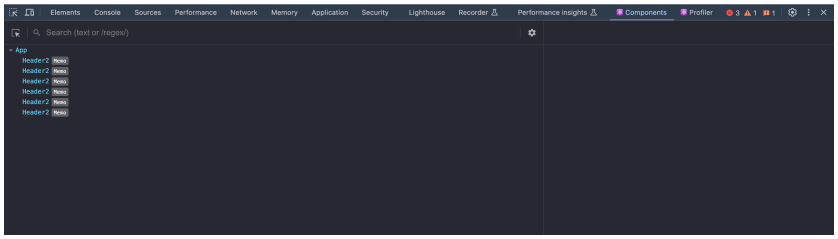
```
const Header=React.memo(function Header(props){
  return <div>
    {props.title}
  </div>

})
```

4. Also import react memo first

```
import {useMemo} from "react";
```

Out put -





Keys in React

Create a simple todo App .

```
src > App.jsx > App
2   import {useMemo} from "react";
3   function App() {
4     const [todos,setTodos]=useState([{
5       id:1,
6       title:"go to gym",
7       description:"from 6-9 pm"
8     },
9     {
10      id:2,
11      title:"DSA",
12      description:"from 9-12 pm"
13    },
14    {
15      id:3,
16      title:"Dev",
17      description:"from 2-5 pm"
18    }])
19    function updateTodo(){
20      setTodos([...todos,{
21        id:4,
22        title:Math.random(),
23        description:Math.random()
24      }])
25    }
26    return (
27     <div>
28      <button onClick={updateTodo}>Add Todo</button>
29
30     {todos.map(function(todo){
31      return <Todo title={todo.title} description={todo.description}></Todo>
32     }){}
33     </div>
34    )
35  }
36  function Todo(props){
37    return <div>
38      <div>{props.title}</div>
39      <div>{props.description}</div>
40    </div>
```

Now if run it you will see an error in the chrome console which says -

```
▶Warning: Each child in a list should have a unique "key" prop.

Check the render method of `App`. See https://reactjs.org/link/warning-keys for more information.
    at Todo (http://localhost:5173/src/App.jsx?t=1715080284952:69:53)
    at App (http://localhost:5173/src/App.jsx?t=1715080284952:22:29)
```

Q. Why does React need Keys ?

Imagine that files on your desktop didn't have names. Instead, you'd refer to them by their order — the first file, the second file, and so on. You could get used to it, but once you delete a file, it would get confusing. The second file would become the first file, the third file would be the second file, and so on.

File names in a folder and JSX keys in an array serve a similar purpose. They let us uniquely identify an item between its siblings. A well-chosen key provides more information than the position within the array. Even if the position changes due to reordering, the key lets React identify the item throughout its lifetime.

Keeping list Items in order with the help of the key .

You need to give each array item a key — a string or a number that uniquely identifies it among other items in that array:

Keys tell React which array item each component corresponds to, so that it can match them up later. This becomes important if your array items can move (e.g. due to sorting), get inserted, or get deleted. A well-chosen key helps React infer what exactly has happened, and make the correct updates to the DOM tree.

Q. What we want to do to correct it ?

Note - whenever we are dealing with array data always remember to assign unique id to each item in an array . To identify that element .

Now to solve this we simply need to add one more attribute to the components name **key** and provide our unique identifier to it such as -

```
return <Todo key={todos.id} title={todo.title} description={todo.description}></Todo>
```

Wrapper Components
Wrapper components, also known as higher-order components (HOCs), are components in React that wrap around other components, enhancing their functionality or behavior. They are a pattern used to compose and reuse functionality across multiple components without repeating code. Here's how wrapper components work:

1. **Accepting Components as Props**: Wrapper components accept other components as props. These components are often referred to as "child components" or "wrapped components."

2. **Enhancing Functionality**: Wrapper components can enhance the functionality of the child components by adding additional props, state, or behavior. They can also intercept props and modify them before passing them down to the child components.

3. **Composition**: Since wrapper components accept components as props, they enable composition, allowing you to combine multiple pieces of functionality from different wrapper components into a single component.

4. **Reusability**: Wrapper components promote reusability by encapsulating common functionality that can be applied to multiple components throughout the application.

For example you can create a wrapper Card Component that takes the inner React components as an input .

Firstly created a Card Wrapper function in which props takes the another components as input .

```
function CardWrapper(props){
return <div style={{border:"2px solid black"}}>
    {props.innerComponent}
  </div>

}
```

```
function App() {

  return <div>
    <CardWrapper innerComponent={<TextComponent></TextComponent>}></CardWrapper>
  </div>
}
```

In innerComponent I passed the Component name TextComponent for inner content of my CardWrapper components .

Here what my TextComponent looks like -

```
function TextComponent(){
  return <div>
    hit here
  </div>
}
```

Output -

hit here

But 99.9 percent you wouldn't use this syntax what will you use is -

Step 1 - You will simply create the wrapper function forExample CardWrapper .

Step 2 - call the CardWrapper in the App function .

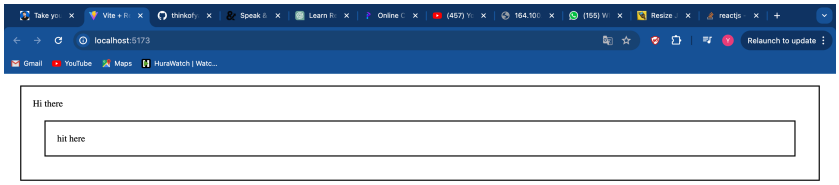
Step 3 - write all your inner logic inside the CardWrapper component call in the App function.

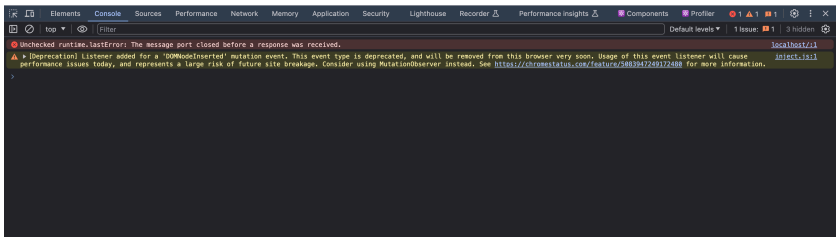
```
function App() {

 return <div>
  <CardWrapper>

    <div>Hi there </div>

    <CardWrapper>
      <TextComponent></TextComponent>
      </CardWrapper>

  </CardWrapper>

 </div>
}
```

Step 4 - Use the children props to access all the logic you have written inside the CardWrapper and rendering it in the Wrapper function .

```
function CardWrapper(props){
return <div style={{border:"2px solid black", padding:20, margin:20}}>
  {props.children}
  </div>

}
```

Output -





Class 6.2

Common Hooks in React

UseEffect, useCallBack, useMemo, custom hooks ,prop Drilling

Some Jargons We need to Understand Before We start this session -

1. Side Effects

In react , we have functional components(we create a function inside in which we write some logic). Anything which is not related to rendering or anything which is not related to putting things on the DOM taking things out of the DOM are called Side Effects .

Specifically things like Whenever you are trying to fetch data from the backend or you are trying to create a setTimeOut or a setInterval or if you're directly trying to manipulate the DOM(which is something you don't do in react). All of this can be considered as something called side effects because these aren't part of the main react rendering cycle.

We need to separate this side effects from the react rendering cycle .There are specific things that are happening when you are rendering the components(functions get called (components get called)).

And then there are other set of things that need to happen on the side which is why they are called side effects .

2. Hooks

Hooks in React are special functions that allow you to use state and other React features in functional components. Before hooks were introduced, state and lifecycle methods were only available in class components. However, with hooks, you can use state, effects, context, and more in functional components, which simplifies component logic and encourages code reuse.

LifeCycle Methods

Lifecycle methods in React are special methods that are called automatically at various points in the lifecycle of a component. These methods allow you to perform different tasks at different stages of a component's existence, such as when it is first created, updated, or removed from the DOM.

Before the introduction of React hooks, lifecycle methods were primarily used in class components. With the introduction of functional components and hooks, many lifecycle methods have been replaced by equivalent functionalities provided by hooks like **useEffect**.

Mounting:

- **constructor()**: This method is called when a component is first created. It's used for initializing state and binding event handlers.
- **componentDidMount()**: This method is called after the component has been mounted (inserted into the DOM). It's commonly used for tasks like fetching data from a server or initializing third-party libraries.

Updating:

- **componentDidUpdate(prevProps, prevState)**: This method is called after the component's state or props have changed. It's used for responding to changes and performing side effects.
- **static getDerivedStateFromProps(props, state)**: This method is called every time the component is rendered, before **render()**. It allows the component to update its internal state based on changes in props. Note: This method is rarely used in practice due to its complexity and potential for bugs.

Unmounting:

- **componentWillUnmount()**: This method is called just before the component is removed from the DOM. It's used for cleanup tasks like unsubscribing from event listeners or cancelling network requests.

Error Handling:

- **componentDidCatch(error, info)**: This method is called when an error occurs during rendering, in a lifecycle method, or in the constructor of any child component. It's used for displaying fallback UI and logging error information.

Some common hooks are
1. useState
2. useEffect
3. useCallback
4. useMemo
5. useRef
6. useContext

These are bunch of popular hooks .

useState Hook

Lets you describe the state of your app whenever state updates , it triggers re - render which finally result in the Dom update .

Basic Usage

To use the `useState` hook, you need to import it from React:

javascript Copy code

```javascript
import React, { useState } from 'react';
```

A SIMPLE COUNTER APPLICATION

javascript Copy code

```javascript
import React, { useState } from 'react';

function Counter() {
  // Declare a state variable named "count" and a function to update it
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}

export default Counter;
```

Destructuring

The `useState` hook returns an array with two elements:

1. The current state value.
2. A function that lets you update the state.

We use array destructuring to assign these values to variables.

The **useState** hook is a powerful and essential tool in React for adding state to functional components.

useEffect Hook

The useEffect hook is a feature in react ,a popular JavaScript library for building user interfaces . It allows you to perform side effects in function components .

Side effects are operations that can affect other components or can't be done during rendering , such as data fetching ,subscriptions or manually changing the DOM in react Components .

The 'useEffect' hook serves the same purpose as 'compoundDidMount', componentDidUpdate, and componentWillUnmount in react class components ,but unified into a single API.

Q. WHY WE CAN'T DO BOTH RENDERING AND FETCHING DATA FROM API PARALLEL?

The primary reason you can't directly fetch data from an API during the rendering phase in React is because rendering should be a pure and synchronous process.

Rendering as a Pure Function-

Purity of Rendering:

- The render method of a React component should be a pure function. This means that given the same inputs (props and state), it should always produce the same output (React elements).
- Pure functions do not cause side effects. They do not alter state or interact with external systems (like APIs) directly.

- Synchronous Execution:
  - Rendering is a synchronous process. React needs to calculate the next state of the UI based on the current state and props without waiting for asynchronous operations to complete.
  - If rendering involved asynchronous operations, it would complicate the reconciliation process and make it difficult for React to maintain a consistent UI state.

Asynchronous Nature of Data Fetching

Uncertainty and Delay:

- Data fetching is inherently asynchronous and involves network requests, which are subject to delays, failures, and varying response times.
- Including asynchronous operations within the render method would make the rendering process uncertain and potentially block it indefinitely while waiting for the data to be fetched.

State Management:

- State updates in React are managed in a predictable manner. When data is fetched, it triggers a state change, which in turn triggers a re-render.
- This ensures that the component renders its initial state immediately and updates itself once the asynchronous data fetching is complete, providing a responsive user experience.

React's Approach to Side Effects

React provides hooks like **useEffect** to handle side effects, including data fetching, in a way that maintains the purity of the rendering process.

Lets Start with Example -

You are a car racer that has to do a 100 laps across a stadium . You are allowed to take a pit stop from time to time .do you take the stop in between every lap ? Or do you take a stop after every 10 laps lets say ?

You will only take pitstop when you need it for example when you want to change your tier or any other reason.

Even tho you pass pit stop in every lap .

Making a Pit stop is a Side effect .

So imagine Taking a laps is rendering . And pit stop a side effect . So you don't want to do it in every render it is something that should independently run and when I say independently I don't mean on different threads I mean independly irrespective of how many renders are happening if a certain condition is met pit stop should be done else pit stop should not be done .this is the same fetching data from the backend .

Suppose when you want to fetch data in the todo application .when the components renders for the first time . After that it doesn't matter how many re - renders happen .you will not send backend request unless you have a certain condition under which you have to do it but you shouldn't do it on every renders .

Pit stop is a side effect . And making a pit stop is something similar to useEffect .

AXIOS

For sending Http Request . Under the hood still uses Fetch APIs. But it is the cleaner way of writing the Syntax .

Step 1 - npm install axios
Step 2 - import it

Step 3 -

```
const res= await axios.get("https://sum-server.100xdevs.com/todos")
setTodos(res.data.todos)
```

Now you might be wondering we don't tell the axios that it is the json data . Because axios is smart enough to understand it .

Note - React Re- renders the screens multiple times but we want some code such as side effects to run and re render only when certain condition mets.thats why we wrap this code under useEffect hook .

```
useEffect(
  ()=>{
   async  function fetchdata(){
    try{
      const res= await axios.get("https://sum-server.100xdevs.com/todos")
      setTodos(res.data.todos)
    }catch(err){
      console.log(err)
    }
    }

    fetchdata();
  }
  ,[])
```

Declare anonymous function

Inside of which you have to create another function if you want to use async/await.

As the dependency array is empty that mean it only run when the object is initially Mounted.

Dependency array

DEPENDENCY ARRAY -

In React, the `useEffect` hook is used to perform side effects in functional components, such as data fetching, subscriptions, or manual DOM manipulations. The dependency array is a crucial part of this hook as it determines when the effect should be re-run.

Empty Array `[]`: The effect runs only once after the initial render, similar to `componentDidMount` in class components.

```
useEffect(() => {
  // Side effect
}, []); // Run once
```

No Array: The effect runs after every render, similar to `componentDidMount` and `componentDidUpdate` in class components.

```
useEffect(() => {
  // Side effect
}); // Run after every render
```

Array with Dependencies `[dep1, dep2, ...]`: The effect runs only when one of the dependencies has changed since the last render.

```
useEffect(() => {
  // Side effect
}, [dep1, dep2]); // Run when dep1 or dep2 changes
```

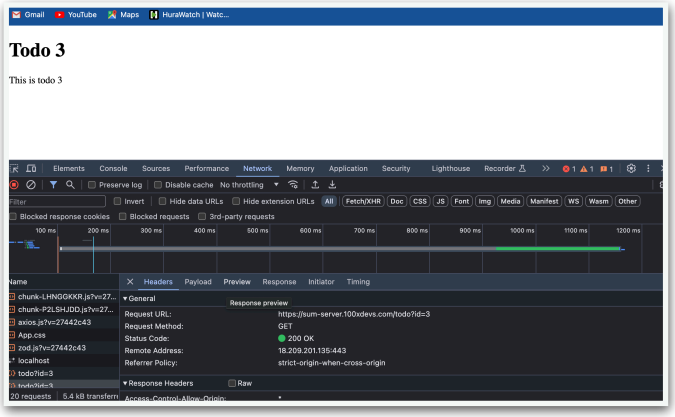
Assignment Question -

Write a component that takes a todo id as an input and fetches the data for that todo from the given endpoint and then renders it .

How would the dependency array change ?

```
function Todo({id}){
// use state
  const [Todos,setTodos]=useState({});
// use effect
  useEffect(()=>{
    async function fetchData(){
    try{
        const res =await axios.get("https://sum-server.100xdevs.com/todo?id="+ id)
        setTodos(res.data.todo)

    }catch(err){
      console.log("error")
    }
    }
    // aysnc-await function
    fetchData();
  },[])
```

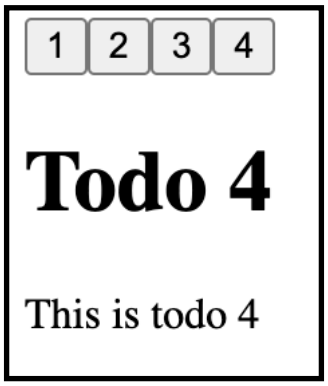
```
function App() {
  return (
    <>
    <Todo id ={3}></Todo>
    </>
  )
}
```



Out put -

You will the request will go out .

Assignment - create a button such that when we click in it it changes number to the button that is clicked .



Answer-

Step 1 - first create a new state.

```
const [count,setCount]=useState(1);
```

Step 2 -

```
return (
  <>
  <button onClick={()=>{setCount(1)}}>1</button>
  <button onClick={()=>{setCount(2)}}>2</button>
  <button onClick={()=>{setCount(3)}}>3 </button>
  <button onClick={()=>{setCount(4)}}>4</button>
  <Todo id ={count}></Todo>
  </>
)
```

Note - we can't direct call the function in onClick attribute instead we have to create an anonymous function . And in that anonymous function we need call the set state function .

Step 3 -

```
function Todo({id}){
// use state
  const [Todos,setTodos]=useState({});
// use effect
  useEffect(()=>{
    async function fetchData(){
    try{
        const res =await axios.get("https://sum-server.100xdevs.com/todo?id="+ id)
        setTodos(res.data.todo)

    }catch(err){
      console.log("error")
    }
    }
    // aysnc-await function
    fetchData();
    // dependency array .
  },[id])
```

Remember we have to add the id in the dependency array because we need to watch out id. And if the id changes we have to trigger the useEffect hook and ultimately it triggers re rendering .

useMemo Hook

Memoization - it means remembering some output given an input and not computing it again .

For example -
lets say you are the driver and you want to check how much petrol is left .

Would you do it in every lap ?
Or would u do that every 10 laps ? Or every 20 minutes .

Assignment -

We ask you to create an app that does two things -
1. Increases a counter by 1 .
2. Lets user put a value in an input box (n) and you need to show sum from 1 to n .

One restriction - everything needs to be inside App.

Step 1- firstly we declare the state of the counter button count and state for the input value we will insert in the input box.

```
let [count,setCount]=useState(1);
let [Inputvalue,setInputvalue]=useState(0);
```

Step 2 -

```
<input placeholder='Enter the Number ' onChange={function(e){
 setInputvalue(e.target.value)
}}></input><br></br>
```

e.target.value reads the input every time it changes and triggers the state . Which finally update the InputValue variable .

```
let counting =0;
for(let i=0;i<=Inputvalue;i++){
  counting=counting+i;

}
```

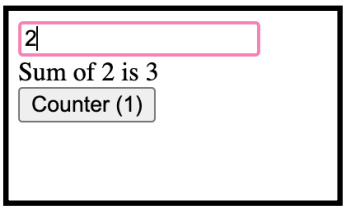
Run a for loop based on Input value .and store answer in the counting variable .

```
Sum of {Inputvalue} is {counting}
```

For button logic it simple -

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

Result -



Now the problem with this code base is every time I click on Counter button it re renders entire screen . Means its fun all the expensive for loop again and again .

Decent approach (but still not the optimal)

Use of useEffect hook .

Step 1 -

```
let [finalvalue,setfinalvalue]=useState(0);
```

Step 2 -

```
useEffect(()=>{

  let counting =0;
for(let i=0;i<=Inputvalue;i++){
  counting=counting+i;

}
setfinalvalue(counting)

},[Inputvalue])
```

Step 3 -

```
Sum of {Inputvalue} is {finalvalue}
```

The problem with this approach that it cause unnessarry re render and it also uses extra state variable . Every time the input value change it re renders and due to change in input value it changes final value which again causes re - render .

Optimal Approach -

It is the use of useMemo hook .

Step 1 -

```
const counted =useMemo(()=>{
  console.log("memo is being called")
  let counting =0;
for(let i=0;i<=Inputvalue;i++){
  counting=counting+i;

}
return counting;
},[Inputvalue])
```

Step 2 -

```
Sum of {Inputvalue} is {counted}
```

useCallback hook

Q. What is referential equality ? And why it is so important in react?

Referential equality is an important concept in React primarily because it affects how React detects changes and determines when to re-render components.

Referential equality means that two references point to the same memory address. If the reference to a state object or array changes, React detects this as a change and triggers a re-render.

For example -
Imagine you have two friends, Alice and Bob. They each have their own toy box.

- **Same Toy Box:** If Alice and Bob are pointing to the exact same toy box, it means they both have the same toy box in their hands. This is like referential equality – two people (or two references) pointing to the same thing (the same memory address).

- **Different Toy Boxes:** If Alice and Bob each have their own toy box, even if the toy boxes look exactly the same and have the same toys inside, they are still different toy boxes. This means the references are different because they point to different toy boxes (different memory addresses).

How Does This Work in React?

React uses a virtual DOM to manage and update the actual DOM efficiently. When state or props change, React compares the new virtual DOM with the previous one to determine the minimum number of changes required to update the real DOM. This process is called “reconciliation."

- **Shallow Comparison**: React performs a shallow comparison of state and props to decide whether a component should re-render. A shallow comparison checks if the references of the objects have changed. If the references are the same, React assumes that there are no changes and skips the re-render.

- **Referential Equality**: Referential equality means that two references point to the same memory address. If the reference to a state object or array changes, React detects this as a change and triggers a re-render.

State updates
When updating state in React, especially for objects and arrays, it's crucial to create new references rather than mutating the existing state directly. This ensures that React detects the change and re-renders the component.

Props and Child Components:

When passing objects or arrays as props to child components, ensuring that these props are updated with new references when their values change is essential. This helps React determine whether the child components need to be re-rendered.

useEffect and Dependency Arrays :
When using hooks like `useEffect`, the dependency array determines when the effect should run. React uses referential equality to check if the dependencies have changed.

Primitive types -

- Primitive types (such as numbers, strings, booleans) are compared by value.

```
var a=10;
var b=10;
console.log(a==b);

function sum(a,b){
    return a+b;
}

function sum1(a,b){
    return a+b;
}

console.log(sum==sum1)
```

```
node /tmp/nOL1OgfmqS.js
true
false

=== Session Ended. Please Run the code again ===
```

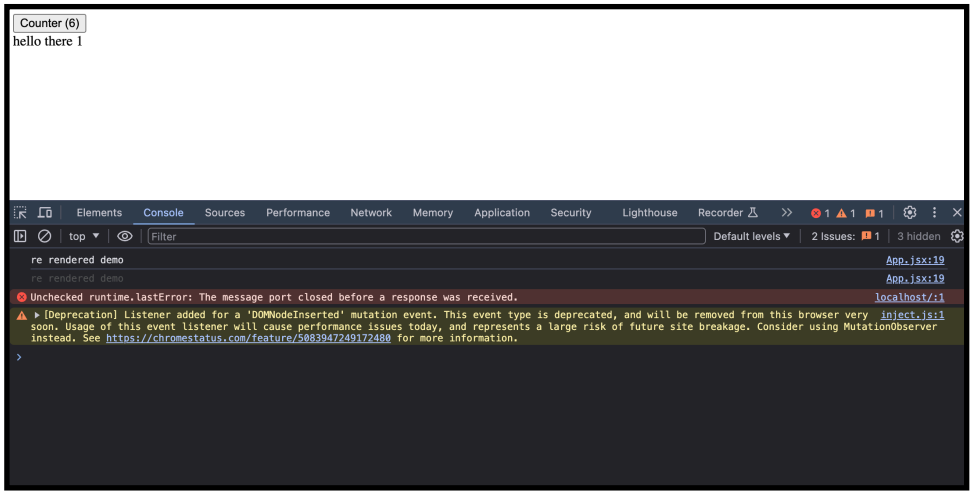
Even tho sum and sum1 is exactly same but they are not referentially equal . Same for a and b variable .

In case of primitive data types :

No re-rendered happened as expected since in upper example a==b is true(they both are same as value). But they are not referentially same .

```
src > App.jsx > Demo
import { useEffect, useMemo, useState } from 'react'
import { memo } from 'react';

import axios from "axios";
import './App.css'
import { set } from 'zod';
function App() {
let [count,setCount]=useState(1);
var a=1; // primitive data types
  return (
    <>
    <button onClick={()=>{setCount(count=count+1)}}>Counter ({count})</button>
    <Demo a={a}></Demo>
    </>
  )
}
// memo will only run this function when somthing is changed in a .
const Demo=memo(function({a}){
  console.log("re rendered demo");  // we put a log to check this
  return <div>
    hello there {a}
  </div>
})
export default App
```

Output -



Only the intial rendered occur . And after that when we click on counter . It triggers re rendered in App component . But after initial render Demo component not rendered until and unless value of the variable changes .

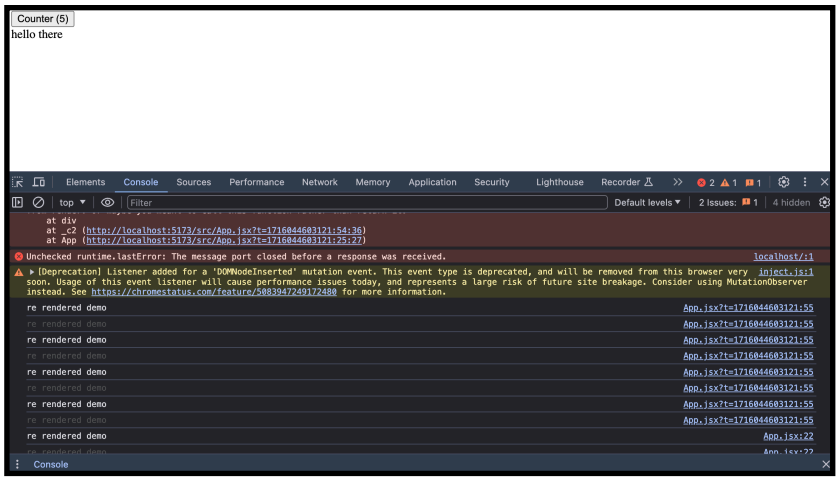
In case of function -

```
src > App.jsx > App
import { useEffect, useMemo, useState } from 'react'
import { memo } from 'react';

import axios from "axios";
import './App.css'
import { set } from 'zod';
function App() {
let [count,setCount]=useState(1);
// changed to function
function a(){
  console.log("hi there")
}
  return (
   <>
   <button onClick={()=>{setCount(count=count+1)}}>Counter ({count})</button>
   <Demo a={a}></Demo>
   </>
  )
}
// memo will only run this function when somthing is changed in a .
const Demo=memo(function({a}){
  console.log("re rendered demo");  // we put a log to check this
  return <div>
    hello there {a}
  </div>
})
export default App
```

Re - rendered happened every time App component re - renders . Because the react thinks that there is might be a change in the value .hence it triggers re - renders .

As expected in upper example sum==sum1 came out to be false . Even tho they are exactly same.but not referentially same .

Output -



Re rendered keeps happening every time the App component re renders .

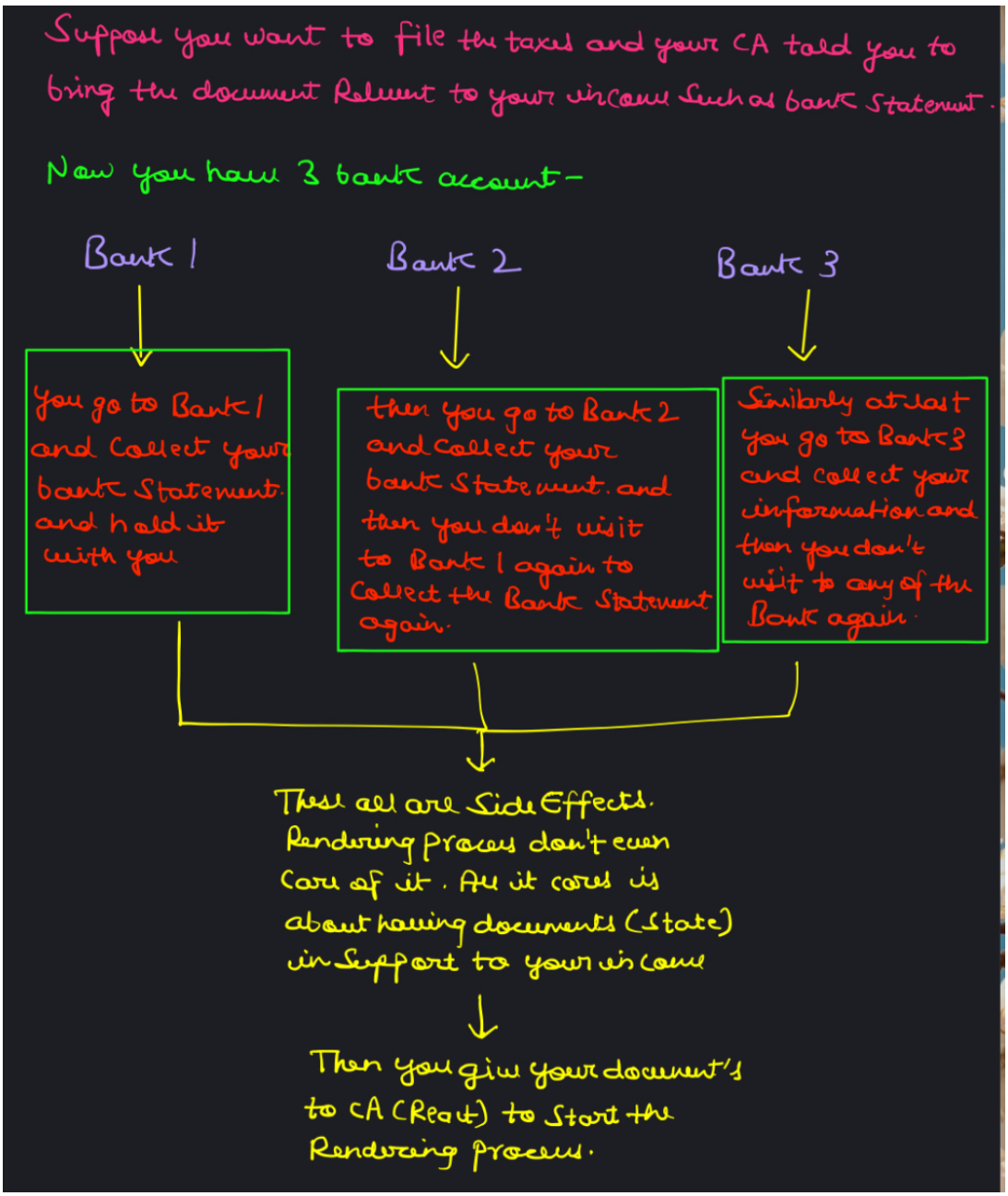
Even tho the function does not changes . But it referencially changes .

So to prevent this unnecessary re renders we often **use callbacks** hooks .

```
src > App.jsx > App
import { useCallback, useEffect, useMemo, useState } from 'react'
import { memo } from 'react';

import axios from "axios";
import './App.css'
import { set } from 'zod';
function App() {
let [count,setCount]=useState(1);
// wrapping the function inside the useCallback Hook .
var a = useCallback(()=>{
  console.log("hello there")
},[])
  return (
   <>
   <button onClick={()=>{setCount(count=count+1)}}>Counter ({count})</button>
   <Demo a={a}></Demo>
   </>
  )
}
// memo will only run this function when somthing is changed in a .
const Demo=memo(function({a}){
  console.log("re rendered demo");  // we put a log to check this
  return <div>
    hello there {a}
  </div>
})
export default App
```

Class 6.3

useEffect hook with CA example -



When you are filling taxes ,you might need to get your data from various sources before you give it to your CA (side Effects).

1. You might have to wait for 10 days before you can talk to your bank manager (setTimeout)
2. You might have to go to your exchange brokers office to get your trading information .

You will update your CA with this information as you get it .

Q. Will you, in any case ,do any of these more than once in a single year ?
The answer is no .

For example lets see another Code base -

src > App.jsx > App

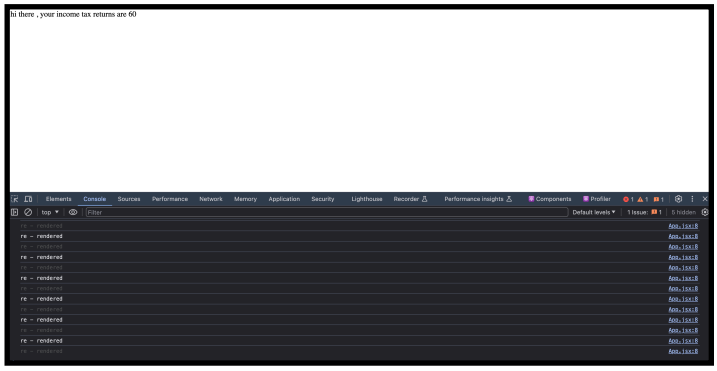
```jsx
import { useState } from 'react'
import './App.css'

function App() {
  const [exchangeData,setExchangeData] = useState({});
  const [bankdata,setBankData]=useState({})

  console.log("re - rendered")

  setTimeout(()=>{
    setBankData({income:100})
  },3000)

  setTimeout(()=>{
    setExchangeData({returns:100})
  },1000);

  const incomeTax=(bankdata.income+exchangeData.returns)*0.3
  return (
    <>
    hi there , your income tax returns are {incomeTax}
    </>
  )
}
export default App
```

You might have to wait for your bank data

Similarly for your Exchange data.

And once you get your data you don't need to visit them again. But the problem is its keep on getting rendered.

Out put -



Renders keep on happening

Even when we get the result has 60.

So what is the solution for it ?

Answer - we need useEffect hook.

Solution -

src > App.jsx > App > useEffect() callback > setTimeout() callback

```jsx
import { useEffect, useState } from 'react'
import './App.css'

function App() {
  const [exchangeData,setExchangeData] = useState({});
  const [bankdata,setBankData]=useState({})

  console.log("re - rendered")

  useEffect(function(){
    setTimeout(()=>{
      setBankData({income:100})
    },3000)
  },[])

  useEffect(function(){
    setTimeout(()=>{
      setExchangeData({returns:100})
    },1000);
  },[])



  const incomeTax=(bankdata.income+exchangeData.returns)*0.3
  return (
    <>
    hi there , your income tax returns are {incomeTax}
    </>
  )
}
export default App
```

useMemo effect

```
src > App.jsx > App

function App() {
  const [exchange1Data,setExchange1Data] = useState({});
  const [exchange2Data,setExchange2Data] = useState({});
  const [bankdata,setBankData]=useState({})

  console.log("re - rendered")

  useEffect(function(){
    setTimeout(()=>{
      setBankData({income:100})
    },3000)
  },[])

  useEffect(function(){
    setTimeout(()=>{
      setExchange1Data({returns:100})
    },1000);
  },[])

  useEffect(function(){
    setTimeout(()=>{
      setExchange2Data({returns:100})
    },1000);
  },[])

  const exhangereturns = exchange1Data.returns+exchange2Data.returns
  const incomeTax=(bankdata.income+exhangereturns)*0.3
  return (
    <>
    hi there , your income tax returns are {incomeTax}
    </>
  )
}
export default App
```

we will get this after 3 Sec.

we get the Exchange in 1 Sec.

we get the Exchange in 1 Sec.

we computed it after 1 Sec.

So the question is why do we Compute it again and again unless it's value change.

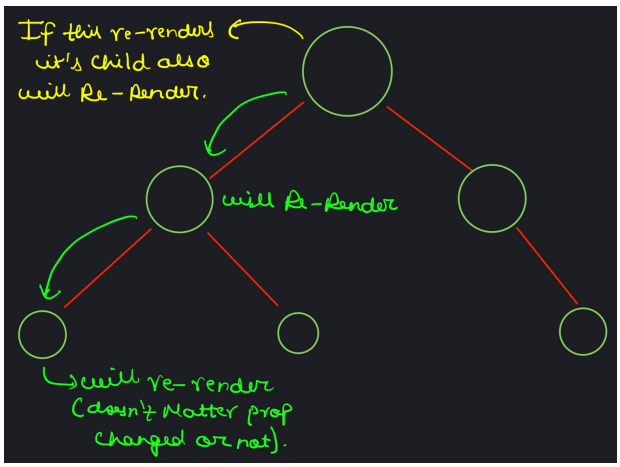
So to avoid these computing again and again we wrap it inside the useMemo hook .

```
const exhangereturns = useMemo(()=>{
  return exchange1Data.returns+exchange2Data.returns;
},[exchange1Data,exchange2Data])
const incomeTax=(bankdata.income+exhangereturns)*0.3
```

memo hook

Note - memo is different from useMemo hook.

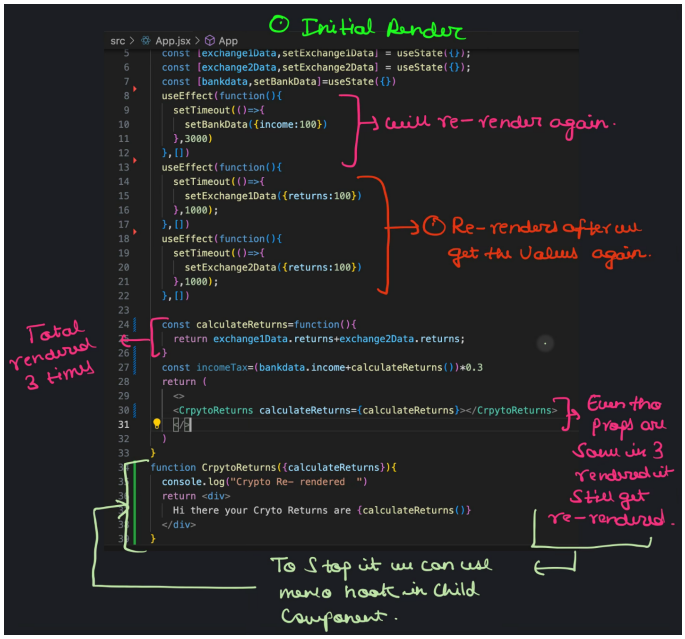
As we know -
If parent Components re renders its child also re renders irrespective of weather the props has changed or not .



What memo lets you do is ?

With the help of memo you can skip re - rendering if props doesn't changes .

For example - if parent component re renders we can skip the re rendering of its child if the props of child doesn't changes .



Even tho the props of child components doesn't change after 2 rendered happen . It still re -rendered .

So to avoid it we will use memo hook on it .

After using memo hook(wrapping the child components into the memo) it still re - rendered 3 time .

```
const CrpytoReturns=memo(function({calculateReturns}){
  console.log("Crypto Re- rendered  ")
  return <div>
    Hi there your Cryto Returns are {calculateReturns()}
  </div>
})
```

Why ?

Because Due to referential Equality . While passing the function in the props . The react thought that something might changed(even tho it doesn't) .it will re render the child component.

So to correct it we will use **useCallback hook** .

```
const calculateReturns=function(){
  return exchange1Data.returns+exchange2Data.returns;
}
```

—->

```
const calculateReturns=useCallback(function(){
  return exchange1Data.returns+exchange2Data.returns;
},[exchange1Data,exchange2Data])
```

Now it will work fine as expected . It Will only re - render when either exchange1data is changed or exchange2data is changed or both .

useRef hook

Lets say you want to do some tax evasion .you want to override what your CA calculated as your income tax.

How would u do it?
You would report an incorrect value to the government .

In react terms - how would you override the screen irrespective of what react has done there ?

Basically you want manually manipulate the DOM .

So basically how would you do it .

```
src > App.jsx > App
import { memo, useCallback, useEffect, useMemo, useState } from 'react'
import './App.css'

function App() {

  useEffect(()=>{setTimeout(()=>{

    document.getElementById("incomeTaxContainer").innerHTML=10


  },5000)},[])
  const incomeTax=2000
  return (
    <>
    hi your income tax is <div id="incomeTaxContainer">{incomeTax}</div>
    </>
  )
}

export default App
```

Now doing the same thing with **useRef hook**

```
# App.css    App.jsx M ×    # index.css
src > App.jsx > App
import { memo, useCallback, useEffect, useMemo, useRef, useState } from 'react'
import './App.css'

function App() {

const divRef=useRef();

  useEffect(()=>{setTimeout(()=>{

   divRef.current.innerHTML=100

  },5000)},[])
  const incomeTax=2000
  return (
    <>
    hi your income tax is <div ref={divRef}>{incomeTax}</div>
    </>
  )
}
export default App
```

NOTE - even tho you don't try to manually manipulate the DOM.