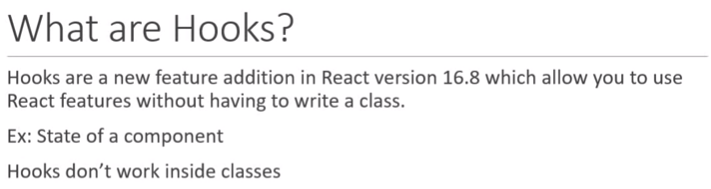
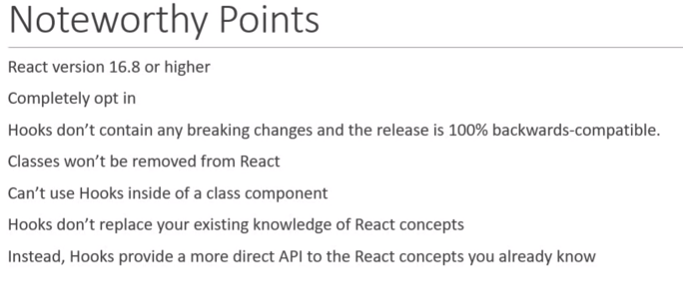
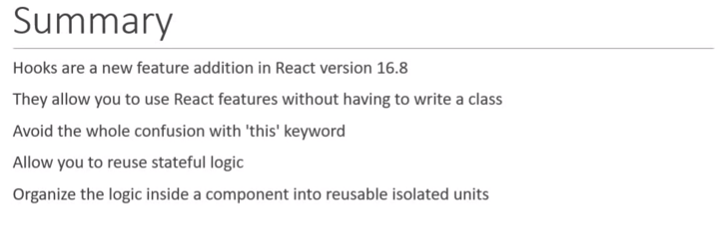
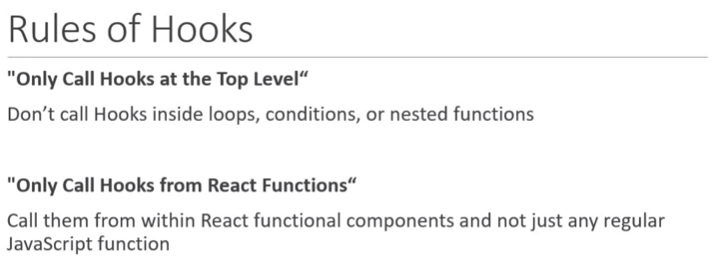
**Introduction**:









**useState Hook**

** **

**useState Hook with previous state:**

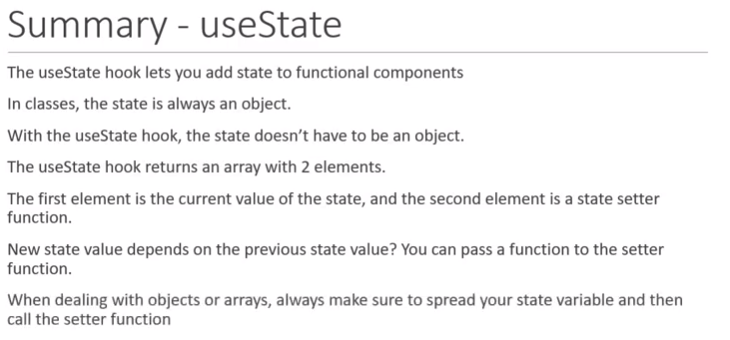
** **

**useState Hook with object:**

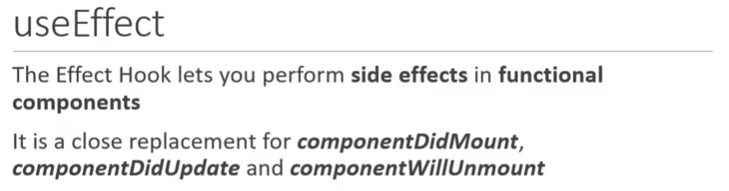
****

**useState Hook with Array:**

****

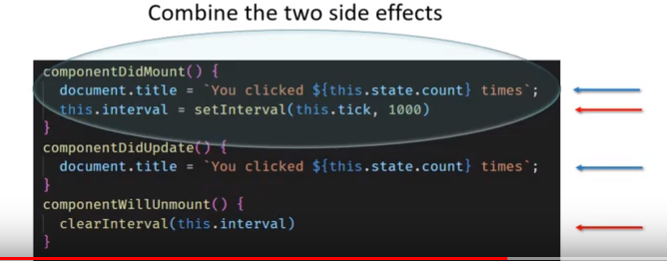
****

**useEffect Hook**

****

Can be used to overcome

* Same code is written twice in two different life cycle methods (didmount and didupdate)
* Related code is split into two different life cycle methods

****

**useEffect after render:**

useEffect is a function, we pass an arrow function as argument which gets executed after every render. It’s similar to what we are trying to achieve in class component i.e on Initial render and every render after that.

useEffect is placed inside the component, which helps useEffect to have access to component states

**Conditionally run useEffect:**

useEffect runs after every render, which might cause performance issues, so we need a way to conditionally run effect from the function.

useEffect(() => {

console.log('useEffect - updating document title')

document.title = `clicked ${count} times`

},[count])

// to conditionally execute an effect we pass an 2nd parameter which is an array

// within this array we need to specify the either props or state we need to watch for

// only those props or state specified in this array were to change the effect will be executed

//good optimization technique to keep in mind

**Run effect only once:**

useEffect(() => {

console.log("useEffect called")

window.addEventListener('mousemove',logMousePosition)

},[])

//passing an empty array executes effect only once, i.e initial render compoenetDidMount()

//it's a way of telling react, this effect deosn't depend on any prop or state, so don't call it on subsequent renders



**useEffect with cleanup:**

Event which have been setup or triggered on component mount, still listen even if the component has unmounted, so basically we have to cleanup the events on component unmount. i.e on componentWillUnmount()

Function passed to useEffect returns a function, which will be executed on componentWillMount(), so whatever you return is your cleanup function.

To cleanup always return a function with cleanup code from the function passed to useEffect

useEffect(() => {

console.log("useEffect called")

window.addEventListener('mousemove',logMousePosition)

return () => {

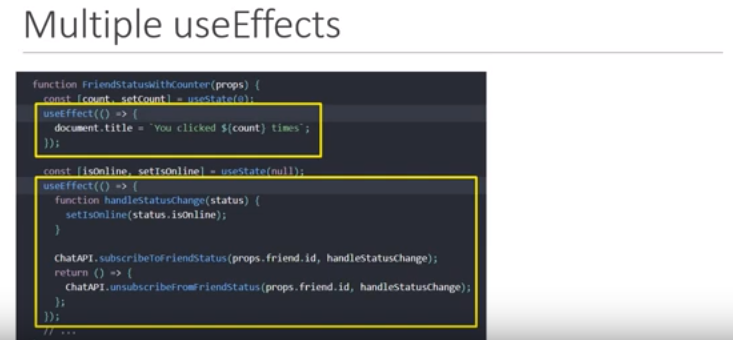
console.log('Component unmount code')

window.removeEventListener('mousemove',logMousePosition)

}

},[])



Its good to have multiple useEffects, instead of putting all the code in one useeffect.

**Data fetching with useEffects:**

****

**useContext Hook**

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

Context(without Hook) –

Creating and providing value

export const UserContext = React.createContext()

export const channelContext = React.createContext()

<UserContext.Provider value={'Ravi'}>

<channelContext.Provider value={'myOwn channel'}>

<ComponentA />

</channelContext.Provider>

</UserContext.Provider>

Consuming value

import {UserContext,channelContext} from '../../../TestHooks'

function ComponentC() {

return (

<div>

<UserContext.Consumer>

{

userpara => {

return (

<channelContext.Consumer>

{

channel => {

return <div>User Context value {userpara} , channel context value {channel}</div>

}

}

</channelContext.Consumer>

)

}

}

</UserContext.Consumer>

</div>

)

}

**useContext** Hook only makes the consumption part simpler, the first two things creating context and providing value to the context remain same as the normal way using context(without hooks).

**steps** to consume context using useContext

1. Import useContext from React
2. Import necessary context
3. Call the useContext function with context as argument

import React,{useContext} from 'react'

import {UserContext,channelContext} from '../../../TestHooks'

function ComponentC() {

const user = useContext(UserContext)

const channel = useContext(channelContext)

return (

<div>

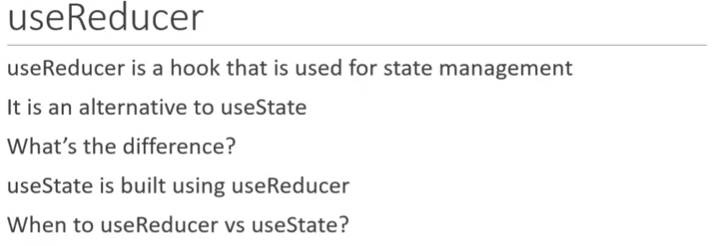
User Context value {user} , channel context value {channel}

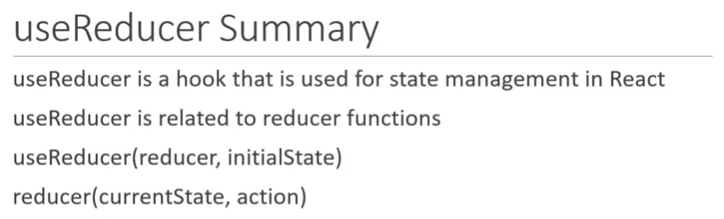
</div>

)

}

**useReducer Hook**

****

****

useReducer is basically changing the states in React components

**useReducer (simple state and action)**



**useReducer (complex state and action)**



**Multiple useReducer**

When dealing with multiple state variables that the same state transitions it is a good idea to have multiple useReducers making use of same reducer(). This will avoid the complexity of merging the state if it were to be an object and also prevents us from duplicating the code in reducer(), like seen in useReducer(complex state and action)



**useReducer with useContext**

useReducer – Local state management

Share state between components – Global state management

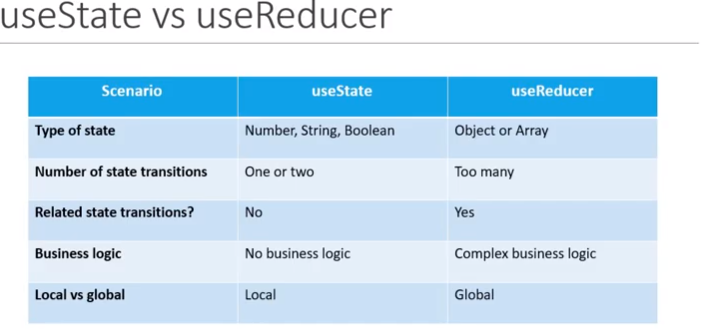
useReducer + useContext



**Fetching data with useReducer**

**useState vs useReducer**



**useCallback Hook**

Helps in performance optimization (check console logs of examples)

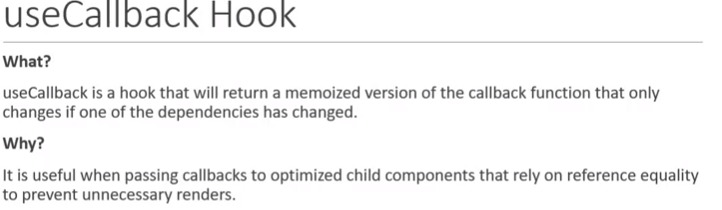
React.memo() renders component only if there is a change in components props or state

Fewer logs are removed but Button component are re-rendered because new increment functions are created each time ParentComponent re-renders. When dealing with functions we always have to consider reference equality, even though two function have exact same behaviour it doesn’t mean they are equal to each other, so the function before re-render is different to the fuction after the re-render. Since we have passed function as a prop React.memo() sees that the prop has changed and will not prevent re-render.

<Button handleClick={incrementSalary} >Increment Salary</Button>

How to fix?

Ans: useCallback



Optimized child components means React.memo() used in example, reference equality is of functions

How to use?

1. Import from react
2. Call useCallback – which takes two parameters:
3. Callback function
4. Dependency array

const incrementSalary = useCallback(

() => {

setSalary(salary + 1000)

},

[salary],

)

Function is cached which is then passes as props to child components