|  |  |
| --- | --- |
|  | React.memo |
| 1. | React.memo() is similar to PureComponent. in Both components will only re-render only if props changes. PureComponent works with classes while React.memo works with functional components |
|  | Wrapping a component –  const MyComponent = React.memo( (props) => {  return (  ………..some code  )  });  or  const MyComponent = (props) => {  return (  ………..some code  )  };  export default React.memo(MyComponent); |
|  | We Should not use React.memo all the time because –  every abstraction (and performance optimization) comes at a cost Specifically the cost for useCallback and useMemo are that you make the code more complex for your co-workers, you could make a mistake in the dependencies array, and you're potentially making performance worse by invoking the built-in hooks and preventing dependencies and memoized values from being garbage collected. Those are all fine costs to incur if you get the performance benefits necessary, but **it's best to measure first.** |
|  | When we pass functions as a parameter to a component, This component will render every time even it that function/prop doesn’t change even using component with react memo because two functions having exact same code are never equal because of reference datatype. To solve it we have to use useCallback for those functions along with using react memo for component |
|  |  |
|  |  |
|  |  |
|  |  |
|  | **USESTATE** |
|  | useState hooks allows to add state to functional component |
|  | The only argument to the useState hook is the initial state. Unlike classes, the sate doesn’t have to be an object. This can be a string, number, object, array or anything else. |
|  | const [count, setCount] = useState(0);  const increaseCount = () => {  setCount(count + 1)  } |
|  | const [counter, setCount] = useState({message: 'No Count', count:0})  const setMessage = () => {  setCount({...counter, message: 'New Count is'});  }  const increaseCount = () => {  setCount({...counter, count: counter.count+ 1})  } |
|  | const [user, setUser] = useState({firstName: '', lastName:''})  const handleChange = (e) => {  setUser({...user, [e.target.name]:e.target.value })  } |
|  | // Declaring multiple state variables!  const [age, setAge] = useState(42);  const [fruit, setFruit] = useState('banana');  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]); |
|  | **USEEFFECT** |
|  | useEffect allows to perform sideeffects in function component |
|  | Use effect can be considered as componentDidMount, componentDidUpdate and componentWillUnmount as combined |
|  | By Default useEffect runs both after first render and after every update |
|  | Unlike componentDidMount and ComponentDidUpdate, effects scheduled with useEffect don’t block the browser from updating the screen |
|  | Function passed to useEffect is different on every render. Every time we re-render , we schedule a different effect, replacing the previous one |
|  | Most of the effects are asynchronous. In case they do there is separate useLayoutEffect hook identical to useEffect |
|  | React performs cleanup when the component unmounts. However effects run on every render and not just once. This is why react also cleans up effect from the previous render before running the render |
|  | For cleanup any effect while using useEffect hook, we return a function within useEffect that includes cleanup login inside it. |
|  | **USEREF** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |