Import React, {useEffect, useState} from "react";
Koi export const useState = () => { } → inside react its written
so we call it in react. why we use curly braces

~~component did mount~~

UseEffect. - Update, mount, unmount.

1) i) Koi data fetch 2) Calculation on same page 3) Koi aisa kaam ~~mandatory~~ mandatory jab hum us screen par aye it should process.

useEffect make for 3 conditions.

i) to give dependencies. ← parameter

useEffect(function, dependency)
↑ mandatory, ↑ not mandatory (prop)

```
i) useEffect(()=>{
     console.log('hello')
   });
```
lots of memory will it take.

```
2) useEffect(()=>{
     console.log('hello1')
   },[]);
```
↑ dependency — update (just once it will work).

```
3) useEffect(()=>{
     console.log('hello2');
   },[value]);
```
depend on change in value/or
↑ Only when this is change will work only.
mount.

3 condition it is used.

useRef() =

1) if on your screen on UI, if you are doing always re-render & if we have take value in state (saved in state) & we don't want to loose it refresh na hojaye or chalina jaye.

So to save it we will put our value in ref. (ref.current). in current object direct value sent it is saved inside it only. & we know ref returns object.. (it is used to store value, state give.

2) In DOM, any component ke properties ke element or components ke functions ko hum directly access kar sakte hai with help of ref.

3) Unwanted renders like jo component bar-bar re-render hote rehte hai So that component ko bar-bar re-render hone se bachate hai ref ka use karte.

Some states which are useless, re-renders, hum state ki helps se change nahi karte, ref ke helps se, we can use know ref to store value then later & remove it

One of the most used from above 3.

```
const RefHookDemo = () => {
  const ref = useRef()
    console.log(ref)
    return (
    <View>
  <TextInput placeholder="Enter Name" ref={ref} />
  <Text onPress={() => {
          ref.current.focus()
    }}>Focus</Text>
    </View>
  )
}
export default RefHookDemo.
```

useContext() Hook.

Props drilling → Props ko drill kar rahe hai.

Parent → to Child → to grand child

Global ← let x = 'hello how are you'

```
Const Hooks = () => {
① return (
    <View ... >
      <ComponentA x={x} />
    </View>
  )
}
export default Hooks.
```

```
const ComponentA = () => {
② return (
    <View>
      <Component B x={x} />
    </View>
  )
}
export default Component A
```

```
③ const Component B = () => {
  return (
    <View>
      <Text> Component B {x} </Text>
    </View>
  )
} export default componentB
```

o/p -
Component B
Hello How are you

A, is missed not need to value.

```
let x = 'hellow how are you'
export const myContext = createContext();
const Hooks = () => {
  return(
  <myContext.Provider value={x} >
    <View>
      <ComponentA />
    </View>
  </MyContext.Provider >
  )
}; export default Hooks.
```

```
const ComponentB = () => {
const values = useContext(MyContext)
  return(
    <View>
<Text>ComponentB {values} </Text>
    </View>
  )
}
export default ComponentB
```

o/p - Component B hellow how are you

For useContext → Fixed value can be passed from parent component to child component value can be passed.
But jyada value ko manipulate nahi kar sakte.

Child to parent & in complication in value change then don't use this.
Best to use useReducer.

Redux - state manage.

without Redux also can be done By this way

useReducer.

Increment 0 Decrement
To pass value with help of action of reducers.

```
const reducer = (state = 0, action) => {
    switch (action.type) {
        case 'INCREMENT':
            return state + 1;
        case 'DECREMENT':
            return state - 1;
        default:
            return state;
    }
};

export const MyContext = createContext();
const Hooks = () => {
    const [state, dispatch] = useReducer(reducer, 0);
    return (
        <MyContext.Provider value={state}>
            <View .....>
                <ComponentB />
            </View>
        </MyContext.Provider>
    )
} export default Hook.

const ComponentB = () => {
    const values = useContext(MyContext);
    return (
        <View>
            <Text style={{fontSize: 20}}> {values} </Text>
        </View>
    )
}
export default ComponentB.
```

Side notes: can be made as global; first class function; first class citizen; ability to be function value; Props; All Here parent to child.

O/P

[Increment]   0   [Decrement]

Increment → 1 2 3 4 5 6
Decrement → 5, 4, 3, 2, 1

useCallback.

usememo - values ke sath deal karta hai

usecallback => Function ke sath deal karta hai.

```
Function App() {

const [add, setAdd] = useState(0);
const [count, setCount] = useState(0);
  const Learning = useCallback(() => {
     // some operation
  }, [count])
```

→ dependency [ ] update
[count] only this will work.

```
  return (
    <div className="App">
    <h1> Learning useCallback </h1>
<ChildA Learning={Learning} count={count} />
    <h1> {add} </h1>
<button onClick={() => setAdd(add + 1)}> Addition </button>
    <h1> {count} </h1>
<button onClick={() => setCount(count + 2)}> Count </button>

    </div>
  )
};
export default App;

function ChildA({Learning, count}) {
     console.log("Child Component")
  return (
     <>

     </>
  )
}
export default memo(ChildA)
```

O/p.

Learning useCallback

2

[Addition]

0

[Count]

console.log -
Also update
← Child Component
← Then when click on Count then only it'll work

React application ki performance ko achha karne ka, enhance karne ke liye use hota hai.

useMemo()

```
function App(){
    const [add, setAdd] = useState(0);
    const [minus, setMinus] = useState(100);

  const multiplication = useMemo(function multiply(){
     console.log("******")
     return add * 10;
    }, [add]
        dependency        update as we learn before 'mount'.

    return (
     <div className="App">
     <h1>Learning useMemo </h1>
      {multiplication} <br/>

     <button onClick={()=setAdd(add+1)}>Addition </button>
     <span>{add} </span><br/>

  <button onClick={()=>setMinus(minus-1)}>Substraction</button>
     <span>{minus} </span>
   </div>
  );
}

 export default App;
```

o/p:

Learning useMemo
10
[Addition] 0 → only when click on it this multiplication will only work
[Substraction] 100 → Here multiplication will not work. this will save memory.

when you make big-big application that time use of unnecessary function creates problem our app gets slow to make it fast. useMemo.

It is very simple to use, it is like useEffect function type only

it takes (function & dependency) deta hai. → kis dependency may chalna hai

This will make our function fast.

Lifting State Up.

M T W T F S S

Page No.: FIONA

Date:

Child to Parent.

As we know how to process from parent to child but know we are learning child to parent how to pass data.

It is easy by jus

```
function App(){
    function getData(data){
      console.log(data)
    }
    return (
      <>
        <Child getData={getData}/>
      </>
    )
}; export default App

function Child(props){
  const [name, setName] = useState();
  function handleSubmit(e){
    e.preventDefault();
    props.getData(name)
  }

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <input type='text' value={name}
onChange={(e)=>{setName(e.target.value)}}></input>
        <button>Submit</button>
      </form>
    </div>
  )
}
export default Child.
```

child to parent will go.

Custom Hooks

1) A custom Hook is a JavaScript function whose name starts with "use".
2) We can use other hooks in custom hooks as well.

Why to use Custom Hook?
To remove the duplicated logic in components and we can extract that logic to custom hook.
(or we can use higher order HOC, 2 or more same code we write so to remove JavaScript function we call anytime, as many times we wanted but we have to define it just once.

File Name to write as "use" → useCounter.js.

useCounter.js

```
function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue)

  function Increment() {
    setCount(count + 1)
  }

  function Decrement() {
    setCount(count - 1)
  }

  return [count, Increment, Decrement]
}

export default useCounter.
```

```
function Counter1() {
  const [count, Increment, Decrement] = useCounter(10);
  return (
    <div>
      <div> {count} </div>
      <button onClick={Increment}> Increment </button>
      <button onClick={Decrement}> Decrement </button>
    </div>
  );
}
export default Counter1.

function Counter2() {
  const [count, Increment, Decrement] = useCounter(0);
  return (
    <div>
      <div> {count} </div>
      <button onClick={Increment}> Increment </button>
      <button onClick={Decrement}> Decrement </button>
    </div>
  );
}
export default Counter2.

function App() {
  return (
    <div className="app">
      <Counter1 />
      <Counter2 />
    </div>
  );
}
export default App.
```

Hoisting

- variables such as var, let, const are hoisted.

var ⌴ (data, any) = undefined → This is by default.

let ⌴ (any) = nothing, it is in temporary indead zone.
const ⌴ (any) =

Functions are hoisted.
As we know Arrow function are not hoisted i.e.
var func2 = () => { console.log("this is to check"); };

```
var rate = 10
function getRate () {
    var rate = undefined;
    if (rate == undefined) {
        var rate = 6;
        return rate;
    } else {
        return 10;
    }
}
console.log("Rate is", getRate());
```

→ var is global value/function & local variable as we know by default it is undefined. local compare with global system will give priority to local variable then global. so ⌴ this will process

~~guess the o/p~~
o/p everybody will say 10 is output but it is wrong.

o/p is 6 because