REACT HOOKS

Hooks were added to React in version 16.8.

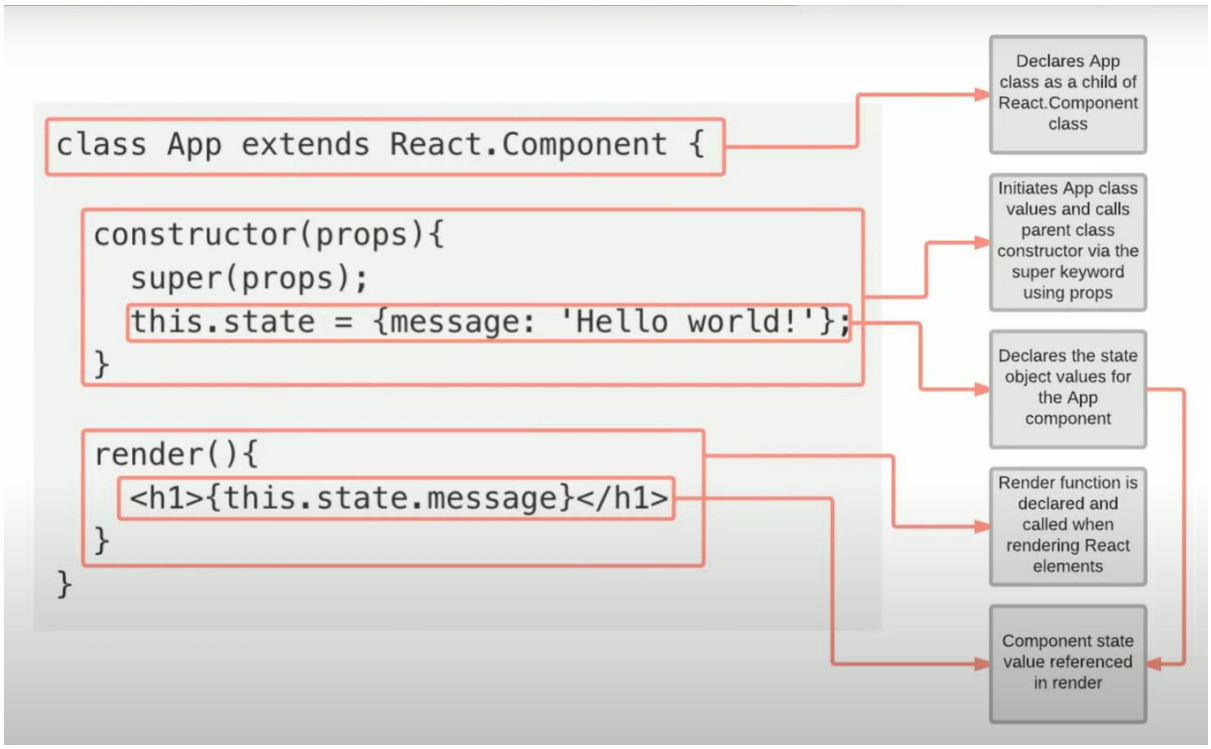
Hooks are functions that let you "hook into" React state and lifecycle features from function components.

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed.

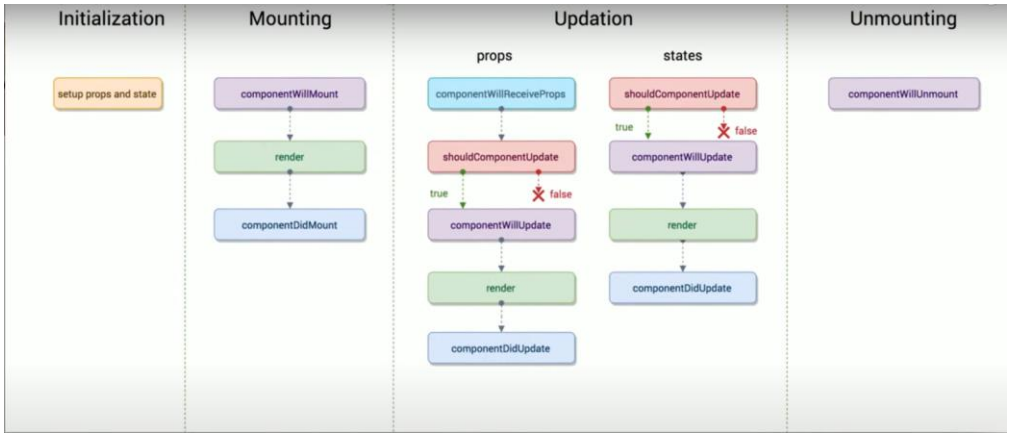
contains data propagated from the parent components. Since functional components do not create their own state properties, they rely on the parents for input data, which made them "Stateless Components."

1. Stateful components (created using classes).
2. Stateless components (created using functions).

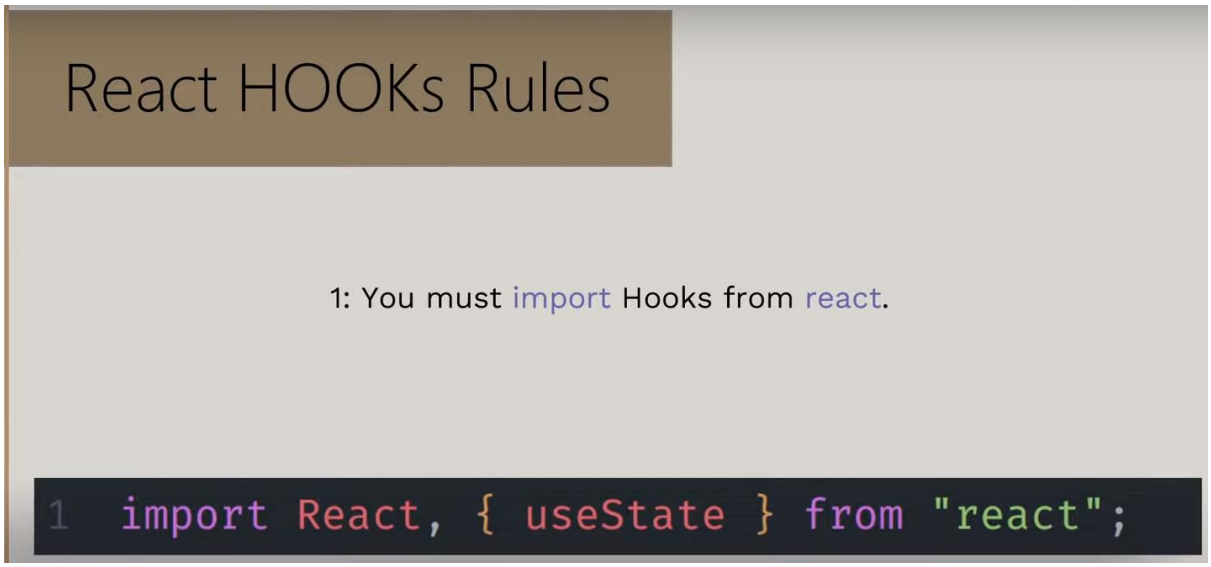
In class we used to do this

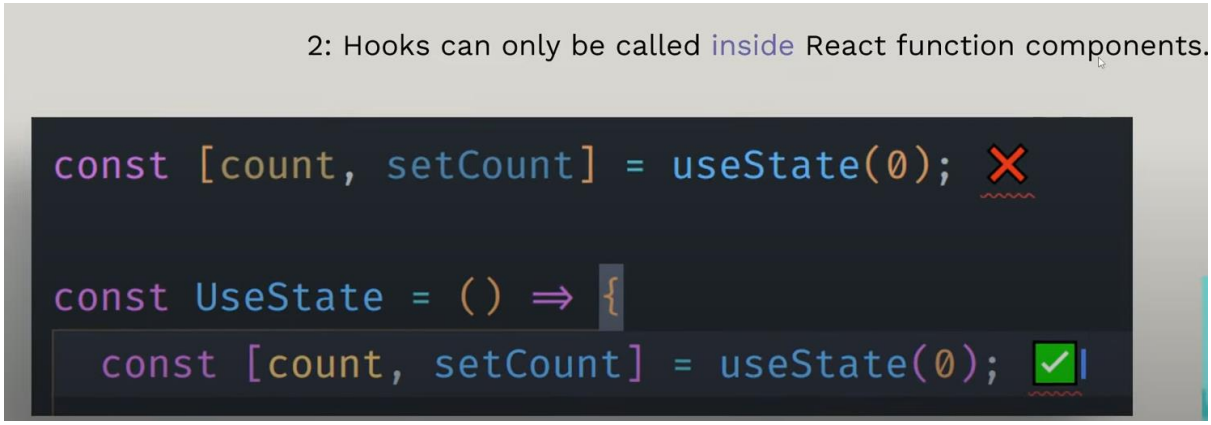


Then came functional components;



In earlier days we didn't find state management tool in functional components,



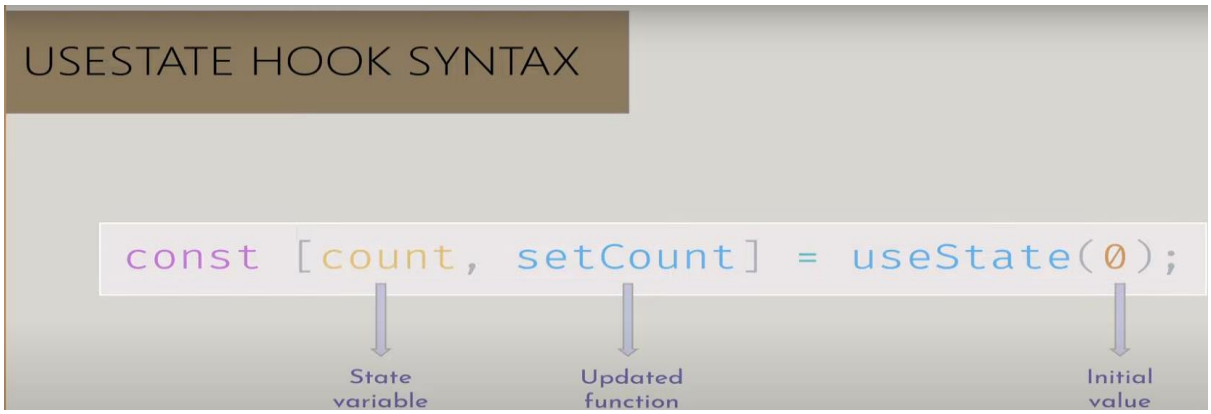


3: Hooks can only be called at the top level of a component.

```
const UseState = () => {
  const [count, setCount] = useState(0);
```

4: Hooks cannot be conditional.

useState Hook



It returns an array of 2 elements

Example:

```
const [count, setCount] = useState(0);
```

Increase

```
<button onClick={() => setCount(count + 1)}>
  <BiPlusMedical className="icon" />
</button>
```

Decrease

```
<button
  onClick={() => (count === 0 ? setCount(0) : setCount(count - 1))}>
  <FaMinus className="icon minus_icon" />
</button>
```

Another example:

```
const [formData, setFormData] = useState({
  username: "",
  email: "",
  password: "",
  confirm_password: "",
});
```

```
const handleInput = (event) => {
  // form ka naam like name, email, etc
  const name = event.target.name;
  // form ki values like age: 6 so 6 is value
  const value = event.target.value;

  // updating state
  setFormData((prev) => {
    // ..prev means cloning data
    return { ...prev, [name]: value };
  });
};
```

In the given code fragment, the spread operator (...prev) is used to create a new object that is a copy of the previous state object, with one or more properties updated to reflect the new input value entered by the user.

```
<div className="form-group">
  <input type="text" className="form-control" id="name" name="username" placeholder="Name"
  autoComplete="off" value={formData.username} onChange={handleInput}/>
</div>
<div className="form-group">
  <input type="email" className="form-control" id="email" name="email" autoComplete="off"
  placeholder="Email" value={formData.email} onChange={handleInput}/>
</div>
<div className="form-group">
  <input type="password" className="form-control" id="password" name="password"
    placeholder="Password" autoComplete="off" value={formData.password} onChange={handleInput}/>
</div>
<div className="form-group">
  <input type="password" className="form-control" id="confirm-password" name="confirm_password"
    placeholder="confirm-password" autoComplete="off" value={formData.confirm_password} onChange={handleInput}/>
</div>
<div className="d-flex flex-row align-items-center justify-content-between">
  <button className="btn btn-primary">Create Account</button>
</div>
</form>
<div>
  <p>{`My name is ${formData.username} and email is ${formData.email} and password is ${formData.password}`} </p>
</div>
</div>
```

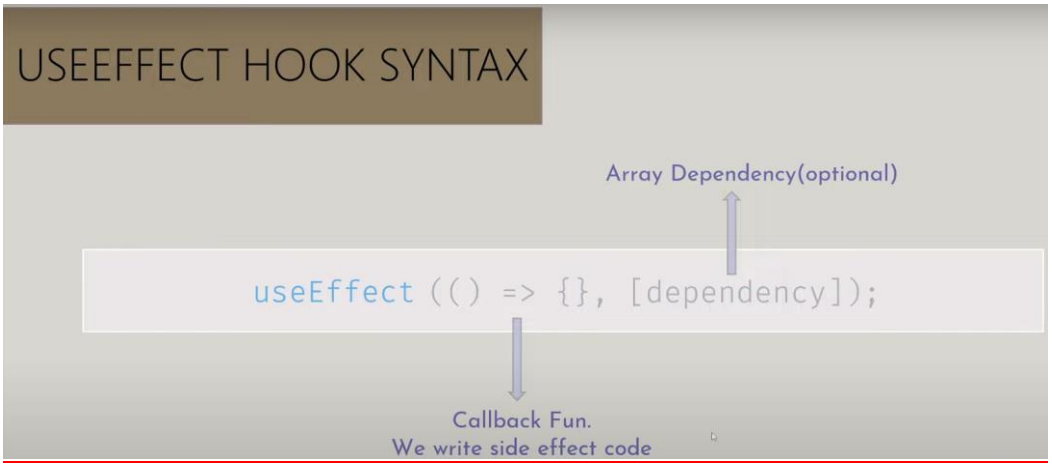
useEffect

The *Effect Hook* lets you perform side effects in function components

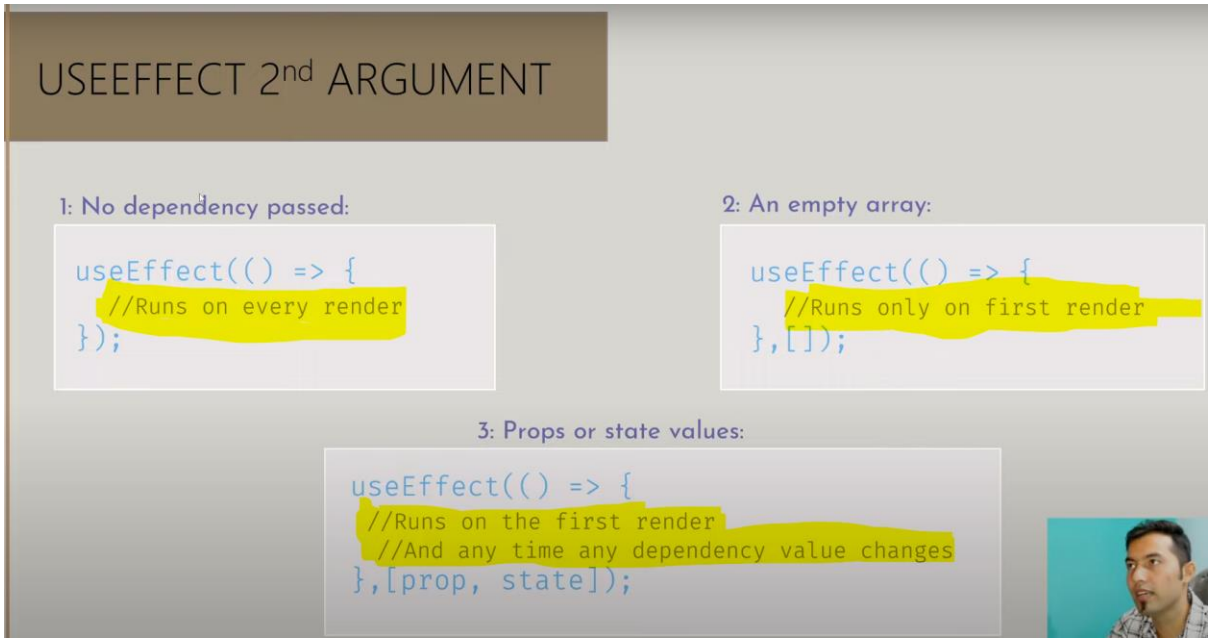
Data fetching, setting up a subscription, and manually changing the DOM in React components are all examples of side effects.

It replaces the classic React functions `componentDidMount` , `componentDidUpdate` , and `componentWillUnmount` with a shorter and simpler syntax.

useEffect is ran every time the component is rendered.



And most importantly



1. 1st case main saare renders pr chalega useEffect
2. 2nd case main sirf first render pr chalega useEffect
3. 3rd case main pehle render aur har **state** and **props** update hone chalega.

useEffect takes two params.

1. A function
2. An array of dependencies (optional)

Example.

> *useEffect(() => {*
>
> *},[${dependency}])*

The moment state is changed useEffect rerenders the component

Example:

```
const countUpdate = (val) => {
  if (val === "inc") return setCount(count + 1);
  if (val === "dec") return setCount(count - 1);
};

useEffect(() => {
  document.title = count;
}, [count]);
```

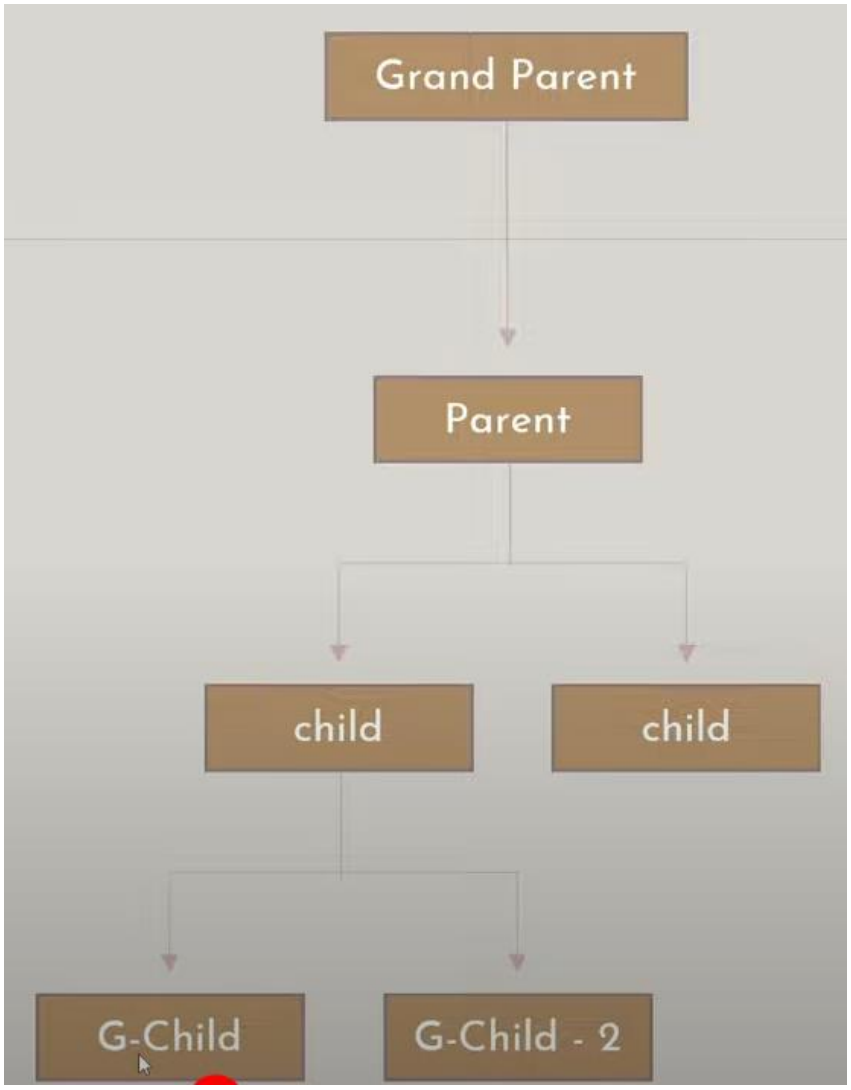
Here, we passed count in dependency array to define that jab bhi ***count*** *state* change hogi tabhi useEffect chalega!

Another example:

```
const [widthCount, setWidthCount] = useState(window.screen.width);

const currentScreenWidth = () => {
  setWidthCount(() => window.innerWidth);
};

useEffect(() => {
  window.addEventListener("resize", currentScreenWidth);
  return () => {
    window.removeEventListener("resize", currentScreenWidth);
  };
});
```

useContext

React Context is a way to manage state globally.



React useContext Hook

React Context is a way to manage state globally.

It can be used together with the `useState` Hook to share state between deeply nested components more easily than with `useState` alone.

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
    </>
  );
}

function Component2({ user }) {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 user={user} />
    </>
  );
```

```
function Component5({ user }) {
  return (
    <>
      <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Component1 />);
```

Even though components 2-4 did not need the state, they had to pass the state along so that it could reach component 5.

Create Context

To create context, you must Import `createContext` and initialize it:

```
import { useState, createContext } from "react";
import ReactDOM from "react-dom/client";

const UserContext = createContext()
```

Next we'll use the Context Provider to wrap the tree of components that need the state Context.

Context Provider

Wrap child components in the Context Provider and supply the state value.

```
function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
    </UserContext.Provider>
  );
}
```

Now, all components in this tree will have access to the user Context

In order to use the Context in a child component, we need to access it using the `useContext` Hook.

First, include the `useContext` in the import statement:

```
import { useState, createContext, useContext } from "react";
```

Then you can access the user Context in all components:

```
function Component5() {
  const user = useContext(UserContext);

  return (
    <>
      <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}
```

useReducer

USEREDUCER HOOK SYNTAX

```
const [state, dispatch] = useReducer(reducer, initial_val);
```

State & Action

The `useReducer` Hook is similar to the `useState` Hook.

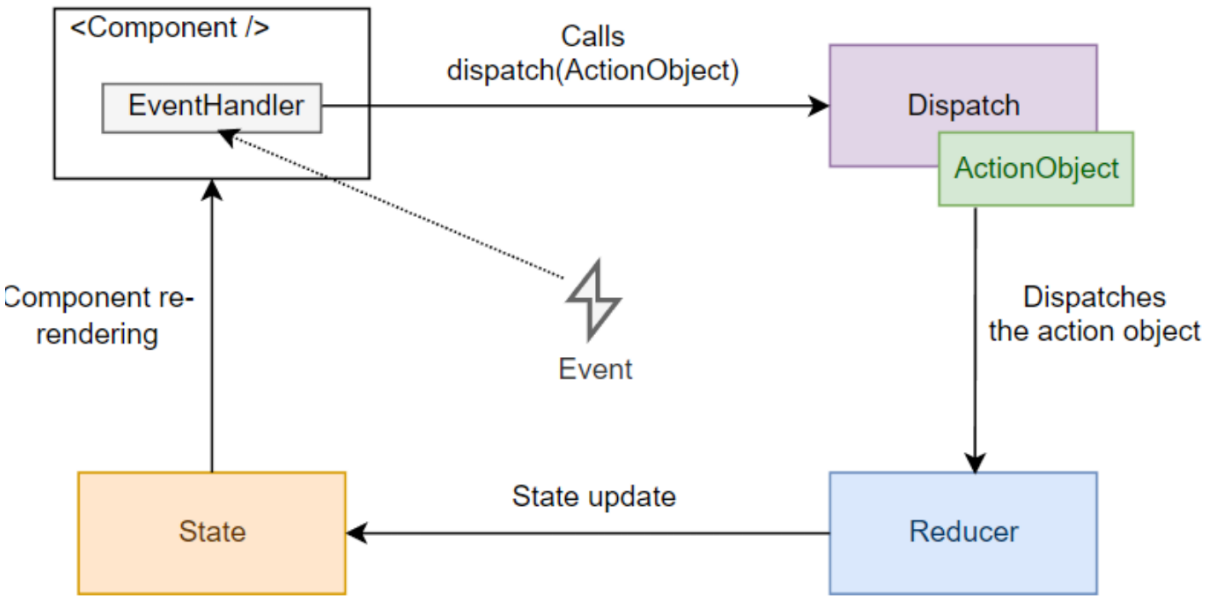
It allows for custom state logic.

The useReducer Hook accepts two arguments.

```
useReducer(<reducer>, <initialState>)
```

The `reducer` function contains your custom state logic and the `initialState` can be a simple value but generally will contain an object.

The `useReducer` Hook returns the current `state` and a `dispatch` method.



The **dispatch function** with the **action object** is **called as a result of an event handler or completing a fetch request.**

Then **React redirects the action object and the current state value to the reducer function.**

The **reducer function** uses the **action object** and **performs a state update**, returning the new state.

React then checks whether the new state differs from the previous one. If the state has been updated, React re-renders the component, and useReducer() returns the new state value: [newState, ...] = useReducer(...).

Example:

```
const [count, dispatch] = useReducer(reducer, initialValue);

return (
  <>
    <Wrapper>
      <div className="container">
        <button onClick={() => dispatch({ type: "INC" })}>
          <BiPlusMedical className="icon" />
        </button>
        <h1>{count}</h1>
        <button onClick={() => dispatch({ type: "DEC" })}>
          <FaMinus className="icon minus_icon" />
        </button>
      </div>
    </Wrapper>
  </>
);
};
```

onClick() dispatch trigger reducer function

below is the reducer function

App.jsx | ReducerHook.jsx | reducer.jsx

c > components > usereducer > reducer.jsx > ...

```
1  const reducer = (count, action) => {
2    switch (action.type) {
3      case "INC":
4        return (count += 1);
5      case "DEC":
6        let newCount = 0;
7        count >= 1 ? (newCount = count - 1) : (newCount = 0);
8        return newCount;
9      default:
10       return count;
11   }
12 };
13 export default reducer;
```

Else Condition

useRef Hook

```
const refContainer = useRef(initialValue);
```

useRef(initialValue) is a built-in React hook that accepts one argument as the initial value and returns a reference. A **reference is an object having a single property "current", which can be accessed and changed (mutated).**

returns a mutable ref object whose .current property is initialized to the passed argument (initialValue).

To run it on every render we pass it in useEffect.

// it create a mutable variable which will not re-render the components

// Used to access the DOM element directly

`useRef` serves a couple main purposes:

1. Access DOM elements
2. Persist values across successive renders]

Example:

```
const RefHook1 = () => {
  const [userInput, setUserInput] = useState("");
  const count = useRef(0);
  // on every rerender
  useEffect(() => {
    count.current = count.current + 1;
  });
```

```
<input type="text" value={userInput} onChange={(e) => setUserInput(e.target.value)}/>
<p>the number of times comp render:{count.current} </p>
```

Another example: (accessing dom element)

Here input main ref use kiya gya hai useRef k hook ko

```jsx
const RefHook = () => {
  const inputRef = useRef();
  const changeBorder = () => {
    inputRef.current.focus();
    inputRef.current.style.backgroundColor = "#82E0AA";
  };

  return (
    <Wrapper>
      <input type="text" ref={inputRef} />
      <br />
      <button onClick={changeBorder}>submit</button>
    </Wrapper>
  );
};
```

useLayoutEffect hook

useLayoutEffect, runs **synchronously**(pehle ek kaam khatam hoga then other kaam honge) after a render but before the screen is updated.

useEffect runs asynchronously and after a render is painted to the screen.

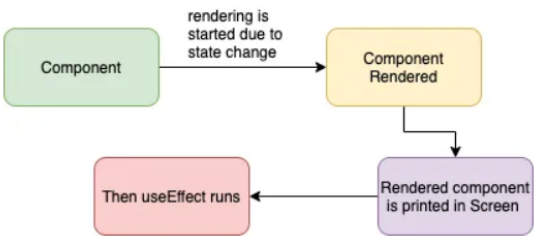
Difference between **useEffect** and **useLayoutEffect?**

useEffect is an asynchronous React hook that runs after the component has rendered and the browser has painted the screen. It does not block the rendering or painting process and is suitable for most side effects.

useLayoutEffect is a synchronous React hook that runs after the component has rendered but before the browser paints the screen. It blocks the rendering process and is useful for side effects that need to be applied synchronously, such as measuring elements or interacting with the DOM layout.

UseEffect

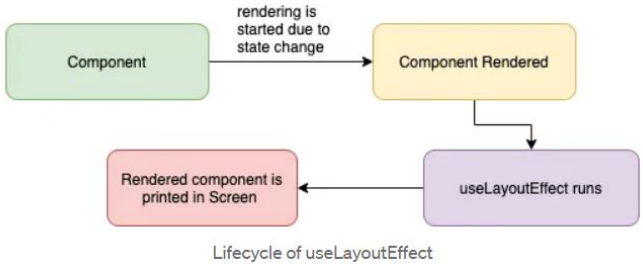
This runs *asynchronously* after rendered elements are printed on the screen.
So basically what happened it this.



Lifecycle of useEffect

UseLayoutEffect

This runs synchronously after the render but before elements are printed on the screen.



Lifecycle of useLayoutEffect

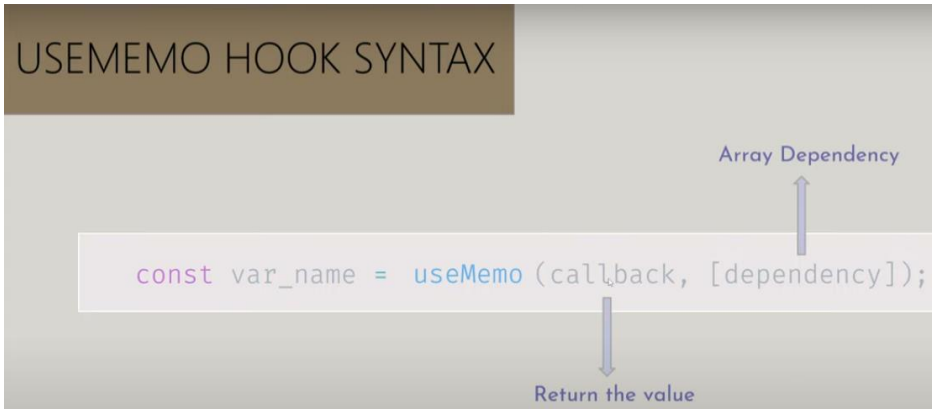
Basically sabse pehle **useLayoutEffect** run hoga then useEffect.

Example:

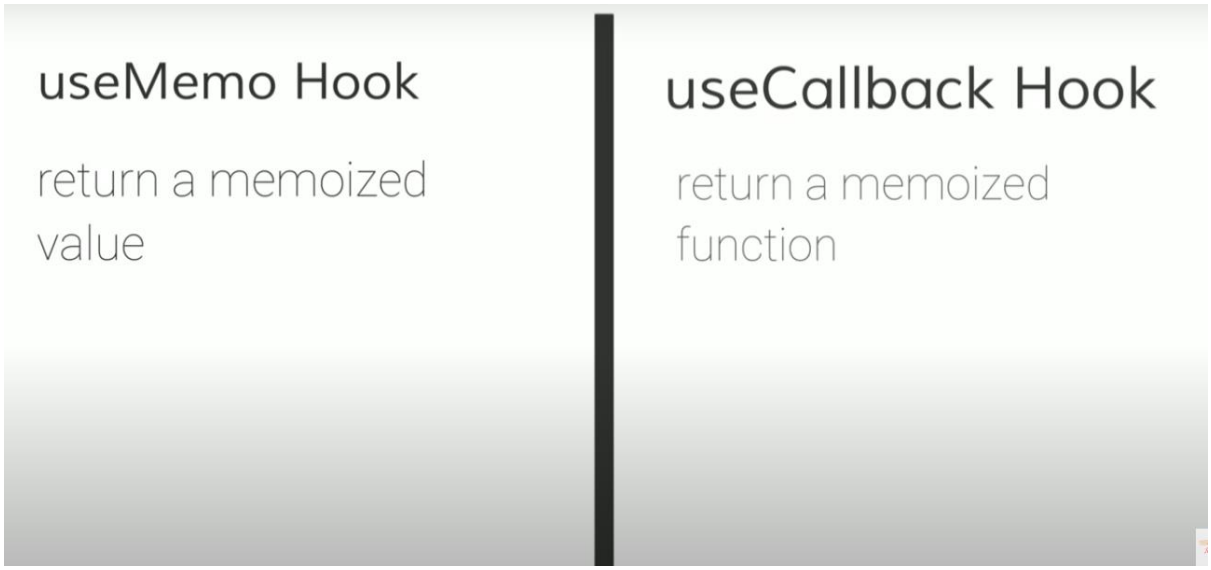
```
const LayoutEffect = () => {
  const [num, setNum] = useState(0);

  useLayoutEffect(() => {
    if (num === 0) setNum(num + Math.random() * 100);
  }, [num]);

  return (
    <Wrapper>
      <p>my new random number {num}</p>
      <button onClick={() => setNum(0)}>random num</button>
    </Wrapper>
  );
};
```

useMemo hook

basically enhance our apps performance.



useCallback hook



Memorization: vvimp

- Memoization in React is a technique used to optimize the performance of components by caching the results of expensive computations.
- It works by storing the output of a function or component and returning the cached result when the same input is provided again.
- This helps avoid unnecessary re-computations, especially when dealing with complex calculations or rendering processes.
- By memoizing components or functions, React can quickly retrieve and reuse previously computed results, resulting in faster rendering and improved user experience.

> Memoization is an optimization feature in React which, when used in the right place, increases the performance of the program.

The React useCallback Hook **returns a memoized callback function**.

Custom Hooks

Following need to be taken note of when working with custom hooks

1. "use" appended to function name representing Custom Hooks
2. The Custom Hook above is returning some values
3. The Custom Hook can then be Re-used

Example:

```
function useCustomHooks() {
  let userName = "Mayank";
  let userDesignation = "Trainer";
  return [userName, userDesignation];
}

function ApplicationComponent() {
  const [name, designation] = useCustomHooks();
  return (
    <div>
      <h1>User Name: {name}</h1>
      <h2>User Designation: {designation}</h2>
    </div>
   )
}
```

Another example:

```
function useCustomHooks(userName, userDesignation) {
  let userName = "User Name: " + userName;
  let userDesignation = "User Designation: " + userDesignation;
  return [userName, userDesignation];
}

function ApplicationComponent() {
  const [name, designation] = useCustomHooks("Mayank", "Trainer");
  return (
    <div>
      <h1>{name}</h1>
      <h2>{designation}</h2>
    </div>
   )
}
```