CipherSchools

Class 6 ReactJS

UseMemo Hook

Definition: The useMemo hook in React is a performance optimization tool designed to prevent unnecessary re-computations of expensive values on every component re-render.

The Problem: Unnecessary Re-renders and Performance Bottlenecks:

Code:

```
import React, { useMemo, useState } from "react";

function MemoComp() {
  const [count, setCount] = useState(0);
  const [count2, setCount2] = useState(0);

  const handleResult =() => {
    console.log("Function called..");
    let a = count;
    for (var i = 0; i < 2000000000; i++) {
      a++;
    }
    return a;
  };

  return (
    <main>
      <h1>{handleResult()}</h1>
      <button onClick={() => setCount((prev) => prev + 1)}>CountByOne</button>
      <button onClick={() => setCount2((prev) => prev + 2)}>CountByTwo
{count2}</button>
    </main>
  );
}
```

```
export default MemoComp;
```

When countByOne is incremented, the UI experiences a noticeable delay because the slow handleResult function is re-executed to update the “Count” display. This is expected.

However, a critical problem arises when countByTwo is incremented. Even though countByTwo has no direct relation to the handleResult function or countByOne, the UI still experiences a delay. This is because "every time the state updates, the component re-renders and when the component re-renders this function is called again." Since the handleResult function is slow, "even when we update count2 the UI update is slow."

This highlights the core problem: expensive calculations are being re-executed unnecessarily on every render, even when their dependencies haven't changed, leading to a sluggish user interface.

Solution

```
const handleResult = useMemo(() => {
   console.log("Function called..");
   let a = count;
   for (var i = 0; i < 2000000000; i++) {
     a++;
   }
   return a;
 }, [count]);
```

1. When count changes, the handleResult function (within useMemo) re-executes, and the UI will still experience a delay, which is expected because the value genuinely needs to be recomputed.
2. When count2 changes, the component re-renders, but useMemo checks its dependencies (count). Since count has not changed, "react is now using the cached value of its handleResult function to display the returned value since the value never changed for count, there is no need to recompute this value. React will simply use the cached

value from the previous render." This results in "updates are way faster" for countByTwo.

UseCallback Hook

Definition: useCallback is a React Hook that returns a memoized (cached) version of a callback function. It only re-creates the function if the dependencies change.

Why Use useCallback?

useCallback is "useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders."

- *Passing callbacks:* This refers to functions like incrementAge and incrementSalary in our code.
- *Optimized child components:* This specifically means components that have been wrapped with React.memo (or similar optimization techniques).
- *Rely on reference equality:* This highlights the core problem useCallback solves – ensuring that a function prop's reference remains the same unless its dependencies change.

How to Use useCallback

1. Import useCallback:
2. import { useCallback } from 'react';
3. Wrap the callback function: useCallback takes two arguments:
    a. First parameter: The callback function itself.
    b. Second parameter: An array of dependencies. The memoized function will only change if any of these dependencies change.

Examples:

Example for incrementAge:

```
const incrementAge = useCallback(() ⇒ {
    setAge(age ⇒ age + 1);
}, [age]); // Dependency: age
```

Example Without useCallback:

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

  const handleClick = () ⇒ {
    console.log('Clicked');
  };

  return (
    <>
      <button onClick={() ⇒ setCount(count + 1)}>Increment</button>
      <Child onClick={handleClick} />
    </>
  );
};

const Child = React.memo(({ onClick }) ⇒ {
  console.log('Child rendered');
  return <button onClick={onClick}>Click me</button>;
});
```

Here, even if handleClick has the same logic, it's a new function on each render, so Child re-renders.

With useCallback:

```
const Parent = () ⇒ {
  const [count, setCount] = useState(0);
  const handleClick = useCallback(() ⇒ {
    console.log('Clicked');
  }, []);

  return (
    <>
      <button onClick={() ⇒ setCount(count + 1)}>Increment</button>
      <Child onClick={handleClick} />
    </>
  );
};
```

Now Child will not re-render when count changes, because the function reference is memoized.

useCallback vs useMemo vs React.memo

Hooks	Purpose
useCallback	Memoizes a function to prevent re-creation unless deps change
useMemo	Memoizes a computed value to avoid recalculation
React.memo	Memoizes a component, avoids re-render if props haven't changed

Pure Component (Memo)

A Pure Component in React is a component that does not re-render if the props and state have not changed. It avoids unnecessary re-renders, which helps improve performance.

PureComponent in Class Components

React provides React.PureComponent, which is similar to React.Component, but it implements shouldComponentUpdate() with a shallow prop and state comparison.

Syntax:

```
import React, { PureComponent } from 'react';

class MyComponent extends PureComponent {
  render() {
    console.log('Rendering MyComponent');
    return <div>{this.props.message}</div>;
  }
}
```

When Props Don't Change:
If the parent re-renders but passes the same props, MyComponent won't re-render.

PureComponent Behavior – Shallow Comparison

PureComponent only compares:

- primitive values (like numbers, strings, booleans)
- references of objects and arrays (not deep comparison)

CODE:

```
this.props = { name: "John" } // === previous props → ✅ no re-render
this.props = { user: { name: "John" } } // ❌ object reference changed → re-render
```

Pure Component in Functional Components

Functional components don’t have shouldComponentUpdate(), but we can use:

- React.memo() – Functional Equivalent of PureComponent

```
const MyComponent = React.memo(function MyComponent({ message }) {
  console.log("Rendering MyComponent");
  return <div>{message}</div>;
});
```

This avoids re-render unless props change (shallow comparison).

Real Use Case – Performance Optimization

In a large app:

- When many components re-render unnecessarily
- Especially components that receive the same props
- Example: item lists, buttons, UI cards

Debugging with Console

```
const MyComponent = React.memo(({ message }) => {
  console.log("Re-rendered:", message);
  return <div>{message}</div>;
});
```

Even if the parent updates, this component won't re-render unless message changes.

Why Not Use PureComponent or memo Everywhere?

Reason	Explanation
Shallow comparison only	Won't detect nested changes
Cognitive overload	Over-optimization makes code harder to maintain

Overhead	memo adds processing for comparison

Renders anyway if functions/objects are passed as props and re-created each time

Common Mistake Example

```
const Child = React.memo(({ data }) => {
  console.log("Re-rendered");
  return <div>{data.name}</div>;
});

function Parent() {
  const data = { name: "CipherSchools" }; // new object on every render
  return <Child data={data} />;
}
```

Solution:

```
const data = useMemo(() => ({ name: "CipherSchools" }), []);
```

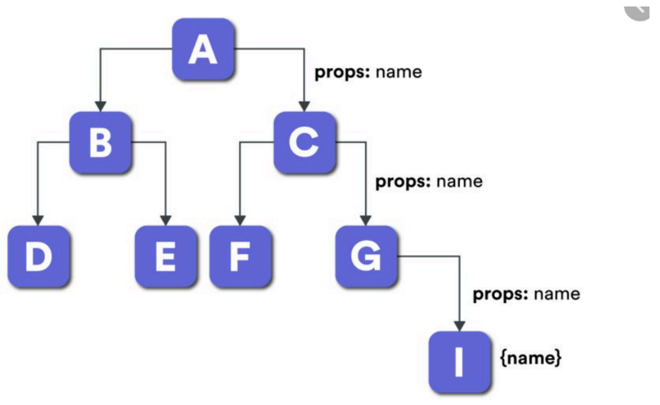
Context Api & useContext Hook

What is the Context API?

Context API is a built-in state management tool in React used to share data between components without passing props manually at every level (prop drilling).

Problem: Prop Drilling

Prop Drilling happens when you need to pass data from a parent to deeply nested child components, even though some intermediate components don’t need the data.



Solution: Context API

Context helps you create global data (like user, theme, language) that can be accessed by any component directly - no need for prop drilling.

Components of Context API

1. React.createContext()
   Creates a new context object.

2. Provider
   Component that wraps parts of your app and provides the context value.

3. Consumer (for class components)
   Component that consumes the context value (less used with hooks).

4. useContext() (for functional components)
   Hook to access context value easily.

Traditional Context API with Consumer

App Structure:

App.js → ContextComp → ComponentD → ComponentF → ComponentG

ContextComp:

```
import React, { useState } from "react";
import ComponentD from "./ComponentD";

export const collegeContext = React.createContext("");
export const nameContext = React.createContext("");

function ContextComp() {
  const [name, setName] = useState("Puneet");

  return (
    <>
      <nameContext.Provider value={name}>
        <collegeContext.Provider value={"LPU"}>
          <ComponentD />
        </collegeContext.Provider>
      </nameContext.Provider>
    </>
  );
}

export default ContextComp;


// 3. Consumer - Traditional Way ⇒ ComponentG
import React from "react";

import { nameContext, collegeContext } from "./ContextComp";

function ComponentG() {
  return (
    <main>
      <nameContext.Consumer>
        {(name) ⇒ {
          return (
            <collegeContext.Consumer>
              {(college) ⇒ {
                return (
                  <h1>
                    Hello {name} from {college}
                  </h1>
                );
              }}
            </collegeContext.Consumer>
          );
        }}
      </nameContext.Consumer>
    </main>
  );
}

export default ComponentG;
```

Note:

- You must use the Consumer inside the Provider to get access to the context.
- You can’t use hooks here, so nesting can get messy.

Modern Way: useContext Hook

```
import React, { useContext } from "react";
import { collegeContext, nameContext } from "./ContextComp";

function ComponentE() {
  const name = useContext(nameContext);
  const college = useContext(collegeContext);
  return (
    <main>
      <h1>
        Hello {name} from {college}
      </h1>
    </main>
  );
}

export default ComponentE;
```

Class 7 ReactJS

UseReducers Hook

Introduction:

The useReducer hook is a fundamental tool in React for state management. It serves as an alternative to useState, which is itself "built using useReducer," making useReducer a "more primitive hook."

While both hooks manage state, the specific scenarios where useReducer is preferred over useState will be clarified after exploring practical examples.

What is reduce() in JavaScript?

The reduce() method is used on arrays to reduce them to a single value.

Syntax:
array.reduce(callback, initialValue)

Parameters:

- callback: Function that runs on each item.
- initialValue: Starting value (optional but recommended).

Example:

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, curr) ⇒ acc + curr, 0);
console.log(sum); // 10
```

- acc is the accumulator.
- curr is the current item.

What is a Reducer Function?

- A reducer is just a pure function that:
- Takes the current state and an action.
- Returns a new state.

Example:

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
}
```

What is useReducer in React?

useReducer is a React Hook for managing complex state logic.

Syntax:

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

- reducer: Function that handles state changes.
- initialState: Starting state value.
- dispatch: Function used to send actions.

Why useReducer?

- Useful for complex states (like objects, nested values).
- Better state transition control.
- Easier to scale when managing multiple actions.
- Similar to Redux style state management.

Example of useReducer Hook

```
import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment': return { count: state.count + 1 };
    case 'decrement': return { count: state.count - 1 };
    default: return state;
  }
}

export default function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <h2>{state.count}</h2>
      <button onClick={() ⇒ dispatch({ type: 'increment' })}>Increase</button>
      <button onClick={() ⇒ dispatch({ type: 'decrement' })}>Decrease</button>
    </div>
  );
}
```

Difference: useState vs useReducer

Feature	useState	useReducer
Use Case	Simple states	Complex state logic (e.g., objects)
Syntax	const [state, setState]	const [state, dispatch]
State Transition	Imperative (setState)	Declarative (action-based)

Multiple State Pieces	Manage separately	Manage as a single object/state
Readability Great for	simple logic	Better for predictable state updates

7. **When to use useReducer over useState?**
   - Prefer useReducer when:
   - State logic is complex.
   - The next state depends on the previous state.
   - You want Redux-style architecture inside components.
   - Managing forms, arrays, objects, or multiple actions.

Complex useReducer Example:

```
import React, { useReducer } from "react";

const initialValue = {
  counterOne: 0,
  counterTwo: 0
};

const reducer = (state, action) ⇒ {
  // action: {type}
  //state : {counterOne, counterTwo}
  switch (action.type) {
    case "increament_counter_1":
      return { ...state, counterOne: state.counterOne + 1 };
    case "decreament_counter_1":
      return { ...state, counterOne: state.counterOne - 1 };
    case "increament_counter_2":
      return { ...state, counterTwo: state.counterTwo + 1 };
    case "decreament_counter_2":
      return { ...state, counterTwo: state.counterTwo - 1 };
    case "reset":
      return initialValue;
    default:
      return state;
  }
};

function RedMultiCounter() {
  const [count, dispatch] = useReducer(reducer, initialValue);
  return (
    <main>
      <h1>{count.counterOne}</h1>
      <button onClick={() ⇒ dispatch({ type: "increament_counter_1" })}>Increament
by 1</button>
      <button onClick={() ⇒ dispatch({ type: "decreament_counter_1"
})}>Decreament by 1</button>
      <h1>{count.counterTwo}</h1>
      <button onClick={() ⇒ dispatch({ type: "increament_counter_2"
})}>Increament by 1</button>
      <button onClick={() ⇒ dispatch({ type: "decreament_counter_2"
})}>Decreament by 1</button>
      <button onClick={() ⇒ dispatch({ type: "reset" })}>Reset</button>
    </main>
  );
}

export default RedMultiCounter;
```

Multi Reducer Example:

```
import React, { useReducer } from "react";

const initialValue = 0;
const initialValue2 = 0;

const reducer = (state, action) ⇒ {
  switch (action.type) {
    case "increament":
      return state + action.payload;
    case "decreament":
      return state - action.payload;
    case "reset":
      return 0;
    default:
      return state;
  }
};

function MultiRedCounter() {
  const [count, dispatch] = useReducer(reducer, initialValue);
  const [count2, dispatch2] = useReducer(reducer, initialValue2);
  return (
    <main>
      <h1>{count}</h1>
      <button onClick={() ⇒ dispatch({ type: "increament", payload: 1
})}>Increament by 1</button>
      <button onClick={() ⇒ dispatch({ type: "decreament", payload: 1
})}>Decreament by 1</button>
      <button onClick={() ⇒ dispatch("reset")}>Reset</button>
      <h1>{count2}</h1>
      <button onClick={() ⇒ dispatch2({ type: "increament", payload: 5
})}>Increament by 1</button>
      <button onClick={() ⇒ dispatch2({ type: "decreament", payload: 5
})}>Decreament by 1</button>
      <button onClick={() ⇒ dispatch2("reset")}>Reset</button>
    </main>
  );
}

export default MultiRedCounter;
```

UseCallback + useReducer Hook

Why Combine useContext and useReducer?

- useReducer: Manages complex state logic.
- useContext: Shares state globally without prop drilling.

Combining both helps to manage and access global state easily in a React-only way - without using Redux or any third-party library.

Example:
Parent.js

```
import React, { useReducer } from "react";
import Counter from "./Counter";
import ComponentA from "./ComponentA";

const initialValue = 0;

const reducer = (state, action) => {
  switch (action) {
    case "increament":
      return state + 1;
    case "decreament":
      return state - 1;
    case "reset":
      return 0;
    default:
      return state;
  }
};

export const counterContext = React.createContext(initialValue);

function Parent() {
  const [state, dispatch] = useReducer(reducer, initialValue);
  return (
    <main>
      <h1>Parent</h1>
      <counterContext.Provider value={{ state, dispatch }}>
        <ComponentA />
      </counterContext.Provider>
    </main>
  );
}

export default Parent;
```

ComponentA.js

```
import React from "react";
import Counter from "./Counter";

function ComponentA() {
  return (
    <main>
      <Counter />
    </main>
  );
}

export default ComponentA;
```

Counter.js

```
import React, { useContext } from "react";

import { counterContext } from "./Parent";

function Counter() {
  const { state, dispatch } = useContext(counterContext);
  return (
    <main>
      <h1>Counter: {state}</h1>
      <button onClick={() ⇒ dispatch("increament")}>Increament by 1</button>
      <button onClick={() ⇒ dispatch("decreament")}>Decreament by 1</button>
      <button onClick={() ⇒ dispatch("reset")}>Reset</button>
    </main>
  );
}

export default Counter;
```

Advantages

- No prop drilling
- Clean separation of logic
- Great for medium-scale apps
- Better performance than lifting state too high

Custom Hook

What is a Custom Hook?

- A custom hook is a JavaScript function whose name starts with "use" and that can call other hooks inside it.
- It allows you to reuse logic across multiple components, avoiding duplication and making your code cleaner and more modular.

Why Do We Need Custom Hooks?

- Code Reusability: Avoid writing the same logic in multiple components.
- Abstraction: Separate business logic from UI code.
- Separation of Concerns: Keep components focused only on rendering.
- Clean and Maintainable Code.

Syntax of a Custom Hook

```
// useCounter.js
import { useState } from 'react';

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

  const increment = () ⇒ setCount((prev) ⇒ prev + 1);
  const decrement = () ⇒ setCount((prev) ⇒ prev - 1);
  const reset = () ⇒ setCount(initialValue);

  return { count, increment, decrement, reset };
}

export default useCounter;
```

Usage:

```
// CounterComponent.jsx
import React from 'react';
import useCounter from './useCounter';

function CounterComponent() {
  const { count, increment, decrement, reset } = useCounter(10);

  return (
    <>
      <h2>{count}</h2>
      <button onClick={increment}>+</button>
      <button onClick={decrement}>-</button>
      <button onClick={reset}>Reset</button>
    </>
  );
}
```

Rules of Custom Hooks

- Must start with use (e.g., useAuth, useTheme, etc.)
- Can use other React hooks like useState, useEffect, useReducer, etc.
- Cannot be conditionally called.
- Must be used inside a functional component or another custom hook.

Example - useFetch

CODE:

```
import React, { useEffect, useState } from "react";

function useFetch(arg) {
  const { url, method } = arg;
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
  const [data, setData] = useState(null);

  const handleFetch = () ⇒ {
    setLoading(true);
    fetch(url, { method: method })
      .then((res) ⇒ res.json())
      .then((result) ⇒ {
        setData(result);
      })
      .catch((error) ⇒ setError(error));
    setLoading(false);
  };

  useEffect(() ⇒ {
    handleFetch();
  }, [url]);

  return { loading, data, error };
}

export default useFetch;
```