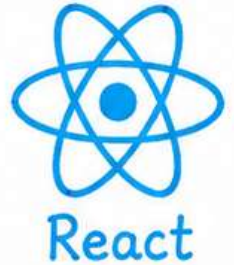


Chapter 5: React Hooks

5.1 Introduction to Hooks

5.1.1 What are Hooks?

- * Hooks are **special functions** introduced in **React 16.8**.
- * They allow us to use **state** and other **React features** in functional components.
- * Before Hooks, **state** and **lifecycle methods** could only be used in class components.
- * Hooks make functional components more **powerful** and **reusable**.



Key Points :

- ✓ Hooks let you use state and other React features without writing a class.
- ✓ They make code shorter, cleaner and easier to maintain.
- ✓ Hooks work only inside functional components.



5.1.2 Why Do We Need Hooks?

- * Class components are harder to understand and reuse.
- * This leads to complex code and "**wrapper hell**" (using too many nested components).
- * Hooks solve these problems by allowing logic reuse.
- * They help in writing **clean** and **maintainable** code.



5.1.3 Common Built-in Hooks

Hook	Purpose	Example
useState	Add state to functional components	<code>const [count, setCount] = useState(0);</code>
useEffect	Perform side effects in components	<code>useEffect(() => { /* effect */ }, []);</code>
useContext	Access context values	<code>const value = useContext(MyContext);</code>
useRef	Access DOM elements or persist values	<code>const ref = useRef(null);</code>
useReducer	Manage complex state logic	<code>const [state, dispatch] = useReducer(...);</code>
useCallback	Memoize functions	<code>const memoFn = useCallback(() => {}, []);</code>
useMemo	Memoize values	<code>const memoValue = useMemo(() => {}, []);</code>

Key Takeaway :



Hooks let us use React features like state, lifecycle and context in functional components, making our code **simpler**, **cleaner** and **reusable**.



5.2 Rules of Hooks

- * Hooks are JavaScript functions, but they have **special rules**.
- * Following these rules ensures Hooks work **correctly**.
- * If you break the rules, React will show an **error**.
- * There are **two main rules** of Hooks.



5.2.1 Rule 1: Only Call Hooks at the Top Level

- * Do not call Hooks inside **loops**, **conditions** or **nested functions**.
- * Always call Hooks at the top level of your React function.
- * This ensures Hooks are called in the **same order** every time.

❌ Incorrect Example

```
function MyComponent() {  
  if (count > 0) {  
    const [count, setCount] = useState(0); // X  
  }  
  return <div>...</div>;  
}
```

✅ Correct Example

```
function MyComponent() {  
  const [count, setCount] = useState(0);  
  const name = useState("");  
  return (  
    <div>...</div>  
  );  
}
```

Hooks must be called
in the same order on
every render.



5.2.2 Rule 2: Only Call Hooks from React Functions

- * Do not call Hooks from regular JavaScript functions.
- * Only call Hooks from **React functional components** or **custom Hooks**.
- * This ensures React can manage Hook state properly.

❌ Incorrect Example

```
function normalFunction() {  
  const [count, setCount] = useState(0); // X  
}  
  
normalFunction();
```

✅ Correct Example

```
function MyComponent() {  
  const [count, setCount] = useState(0);  
  return <div>...</div>;  
}  
// or from a custom Hook  
function useMyHook() {  
  const [value, setValue] = useState("");  
  return value;  
}
```



Key Takeaway:

- ✅ Call Hooks at the top level - not inside loops, conditions or nested functions.
- ✅ Call Hooks only from React functional components or custom Hooks.



These rules help React maintain the correct **order** and **state** of Hooks.



5.3 useState()

- * `useState()` is a React Hook that lets you add **state** to **functional components**.
- * It returns an **array** with two elements.
- * The first element is the **current state value**.
- * The second element is a function that lets you **update the state**.



Syntax :

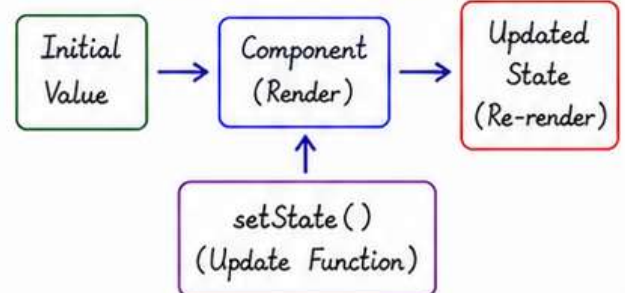
```
const [state, setState] = useState(initialValue);
```

state → Holds the current state value.

setState → Function to update the state.

initialValue → The initial value of the state.

How `useState()` Works



5.3.1 Example

```
import React, { useState } from "react";
function Counter() {
  const [count, setCount] = useState(0); // initial value is 0
  return (
    <div>
      <h2>Count: {count} </h2>
      <button onClick = {() => setCount(count + 1)}> Increment </button>
      <button onClick = {() => setCount(count - 1)}> Decrement </button>
    </div>
  );
}
export default Counter;
```

Output :

Count : 0

Increment

Decrement



5.3.2 Key Points

- * `useState()` can be used **multiple times** in a single component.
- * Updating state using `setState()` will **re-render** the component.
- * State updates are **asynchronous**.

Note :

Never update state directly.
Always use the **`setState()`** function.

5.3.3 Real-World Analogy



Think of state as a variable that React remembers.
When the value **changes**, React updates the **UI** automatically.

Key Takeaway :



`useState()` is the most basic Hook in React.
It helps us **add** and **manage state** in functional components easily.



5.4 useEffect()

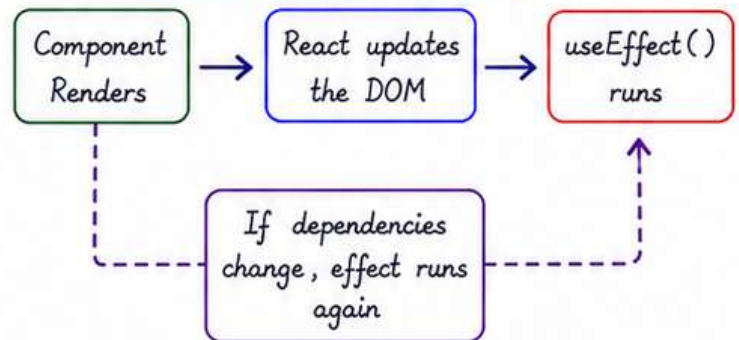
- * `useEffect()` is a React Hook that lets you perform **side effects** in functional components.
- * It runs **after** the component renders.
- * It can be used for **data fetching**, **subscriptions**, **timers**, **DOM manipulation**, and more.
- * It can also **clean up effects** when the component unmounts or before the effect runs again.



Syntax :

```
useEffect(() => {  
  // Side effect code here  
  // Optional: return cleanup function  
  return () => {  
    // Cleanup code here  
  };  
}, [dependencies]);
```

How useEffect() Works



5.4.1 Example

```
import React, { useState, useEffect } from "react";  
function User() {  
  const [user, setUser] = useState(null);  
  
  useEffect(() => {  
    // Fetch data from API  
    fetch("https://jsonplaceholder.typicode.com/users/1")  
      .then((res) => res.json())  
      .then((data) => setUser(data));  
  }, []); // Empty dependency array -> runs only once (after first render)  
  
  return (  
    <div>  
      {user ? <h2>{user.name} </h2> : <p>Loading...</p>}  
    </div>  
  );  
}  
export default User;
```

Output :

Loading...

Leanne Graham

Note :

The empty array `[]` means the effect runs **only once** after the first render.



5.4.2 Dependency Array

Dependency Array	When does it run?	Use Case
<code>[]</code>	Runs only once after the first render	Fetch data, set up subscriptions
<code>[value]</code>	Runs on first render and when "value" changes	Re-run effect when specific value changes
No array	Runs after every render	Rarely used (may cause performance issues)



Key Takeaway :

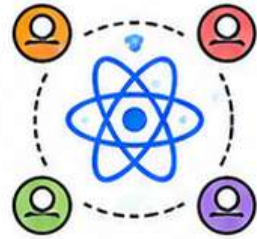


`useEffect()` lets you handle **side effects** in functional components.
Use the **dependency array** to control when the effect runs.



5.5 useContext()

- * `useContext()` is a React Hook that lets you **access context values** without passing props through every level.
- * Context provides a way to share data between components without **prop drilling**.
- * `useContext()` helps us **consume the context** easily.
- * It subscribes the component to the **closest matching Provider**.



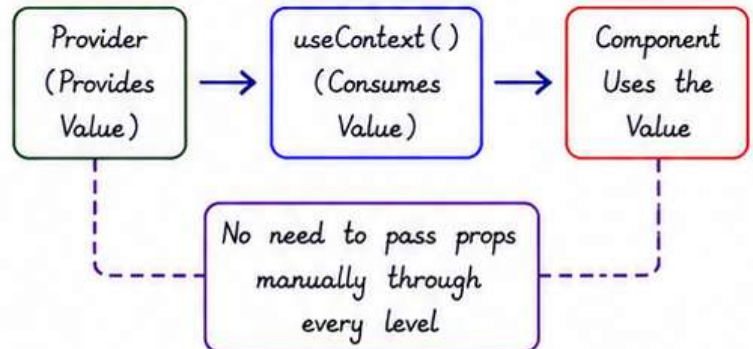
Syntax :

```
const value = useContext(MyContext);
```

value → The current context value.

MyContext → The context object created using `React.createContext()`.

How useContext() Works



5.5.1 Example

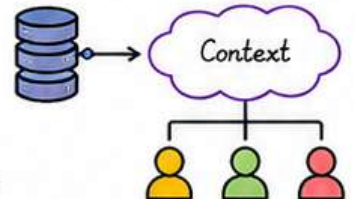
```
import React, { createContext, useContext } from "react";  
// 1. Create Context  
const ThemeContext = createContext();  
// 2. Provider Component  
function App() {  
  const theme = "dark";  
  return (  
    <ThemeContext.Provider value={theme}>  
      <Toolbar />  
    </ThemeContext.Provider>  
  );  
}  
// 3. Component that uses useContext  
function Toolbar() {  
  const theme = useContext(ThemeContext); // Consume context  
  return <h2>Current Theme: {theme}</h2>;  
}  
export default App;
```

Output :

Current Theme: dark

Note :

- * `useContext()` must be used inside **functional components**.
- * The component must be wrapped inside the matching **Provider**.
- * When the context value changes, all consuming components **re-render**.



5.5.2 Use Cases

- ☑ Sharing themes (light / dark mode) across the application.
- ☑ Managing user authentication state.
- ☑ Handling language / localization.
- ☑ Sharing global data like settings, user info, cart data, etc.

Key Takeaway :

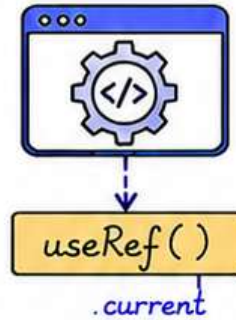


`useContext()` makes it easy to access **shared data** across the component tree without **prop drilling**, improving code readability and maintainability.



5.6 useRef()

- * `useRef()` is a React Hook that lets you **persist values** between renders.
- * It returns a **mutable** ref object with a **.current** property.
- * Updating **.current** **does not re-render** the component.
- * It is commonly used to **access DOM elements** or store **mutable values**.
- * The value inside `useRef()` **persists** for the lifetime of the component.

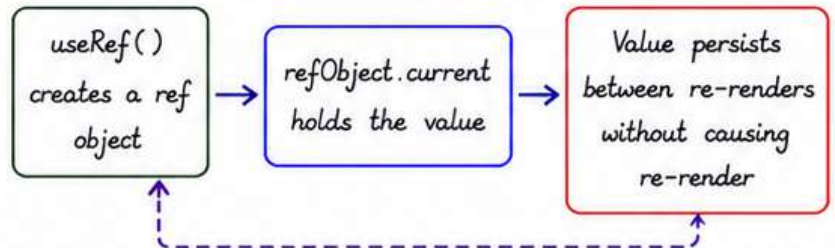


Syntax :

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

refObject → The ref object returned.
refObject.current → The mutable value.
initialValue → The initial value (optional).

How useRef() Works



5.6.1 Example

```
import React, { useRef, useState } from "react";
function Counter() {
  const countRef = useRef(0); // does not cause re-render
  const [count, setCount] = useState(0); // causes re-render

  const handleClick = () => {
    countRef.current = countRef.current + 1; // update ref value
    setCount(count + 1); // update state
  };

  return (
    <div>
      <h2>Count (State): {count} </h2>
      <h2>Count (Ref): {countRef.current} </h2>
      <button onClick={handleClick}> Increment </button>
    </div>
  );
}
export default Counter;
```

Output :

```
Count (State): 3
Count (Ref): 3
```

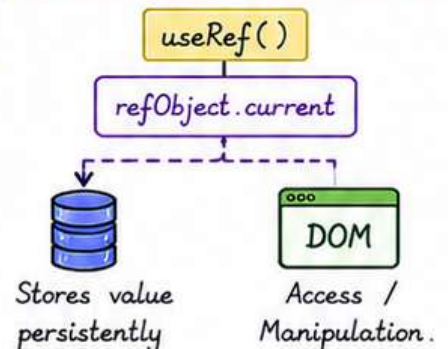
Increment

Note :

- * Changing `countRef.current` **does not re-render** the component.
- * Changing state using `setCount()` **re-renders** the component.
- * `useRef()` is perfect for values that **don't affect the UI**.

5.6.2 Common Use Cases

- ✓ Accessing and manipulating DOM elements.
- ✓ Storing previous values.
- ✓ Managing timers (`setTimeout`, `setInterval`).
- ✓ Avoiding unnecessary re-renders.
- ✓ Storing mutable values that don't affect the UI.



Key Takeaway :

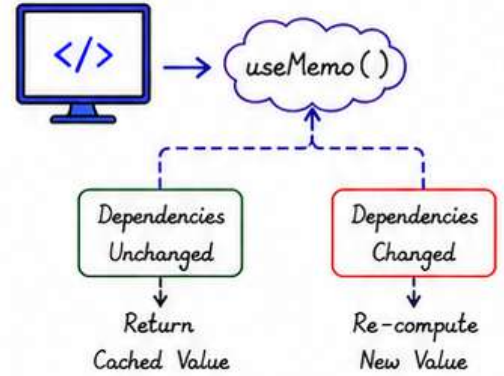


`useRef()` gives us a **mutable** value that persists between renders **without causing re-renders** → ideal for DOM access and storing values that **don't impact** the UI.



5.7 useMemo()

- * `useMemo()` is a React Hook that lets you **memoize the result** of a computation.
- * It returns a **memoized value**.
- * The value is re-computed only when its **dependencies** change.
- * It helps **optimize performance** by avoiding expensive calculations on every render.
- * It is commonly used for **heavy calculations** or data transformations.



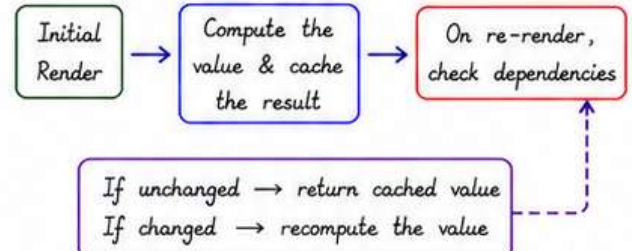
Syntax :

```
const memoizedValue = useMemo(() => computeFunction, [dependencies]);
```

computeFunction → Function that returns the value.

dependencies → Array of values the computation depends on.

How `useMemo()` Works



5.7.1 Example

```
import React, { useState, useMemo } from "react";
function ExpensiveComponent() {
  const [count, setCount] = useState(0);
  const [number, setNumber] = useState(1);
  const factorial = useMemo(() => {
    console.log("Calculating factorial...");
    let result = 1;
    for (let i = 1; i <= number; i++) {
      result *= i;
    }
    return result;
  }, [number]); // Recompute only when 'number' changes
  return (
    <div>
      <h2>Factorial of {number} is {factorial}</h2>
      <button onClick={() => setNumber(number + 1)}>Increase Number</button>
      <button onClick={() => setCount(count + 1)}>Re-render ({count})</button>
    </div>
  );
}
export default ExpensiveComponent;
```

Output :

Initial Render:

Factorial of 1 is 1

After increasing number to 3:

Factorial of 3 is 6

Re-render without changing number:

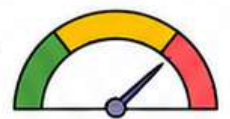
(No "Calculating factorial..." log)
Cached value is used.

Note :

- * `useMemo()` **does not replace** `useEffect()`.
- * It is used only for **memoizing values**, not for side effects.
- * Always include all **dependencies** to get correct results.
- * Overusing `useMemo()` can make code complex and harder to maintain.

5.7.2 Common Use Cases

- ✓ Heavy calculations (e.g., sorting, filtering, complex mathematical operations).
- ✓ Transforming large data sets.
- ✓ Preventing unnecessary re-renders in child components.
- ✓ Optimizing performance in large applications.



Better Performance

Key Takeaway :

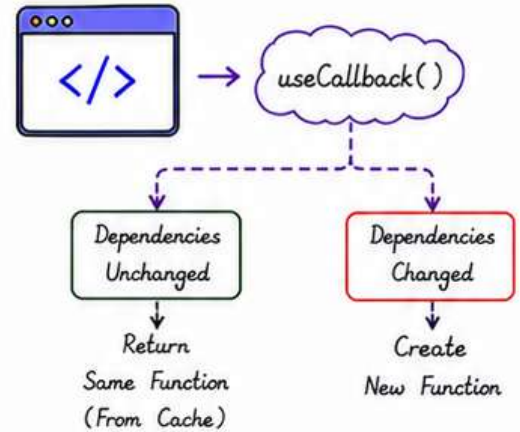


`useMemo()` helps you **cache the result** of expensive calculations and re-compute only when the **dependencies change**, improving **performance**.



5.8 useCallback()

- * `useCallback()` is a React Hook that lets you **memoize** a **function**.
- * It returns a **memoized callback function**.
- * The function is re-created only when its **dependencies** change.
- * It helps to prevent unnecessary re-renders in child components that rely on reference equality.
- * It is commonly used when passing functions to child components or in dependency arrays.



Syntax :

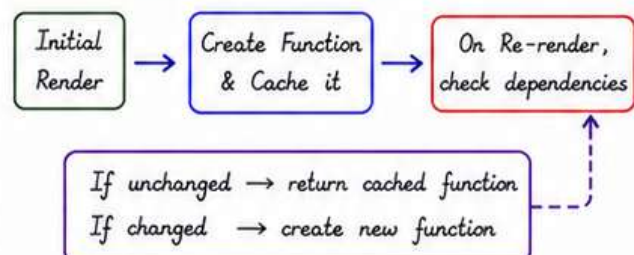
```
const memoizedCallback = useCallback(() => function, [dependencies]);
```

memoizedCallback → The memoized function.

function → The callback function.

dependencies → Array of values the function depends on.

How `useCallback()` Works



5.8.1 Example

```
import React, { useState, useCallback } from "react";
function Child({ onClick }) {
  console.log("Child re-rendered");
  return <button onClick={onClick}>Click Me</button>;
}
function Parent() {
  const [count, setCount] = useState(0);
  const handleClick = useCallback(() => {
    console.log("Button Clicked");
    setCount(count + 1);
  }, [count]); // Re-create only when 'count' changes
  return (
    <div>
      <h2>Count: {count}</h2>
      <Child onClick={handleClick} />
    </div>
  );
}
export default Parent;
```

Output :

Initial Render:

→ Child re-rendered

After clicking button:

→ Button Clicked

→ Child re-rendered

Re-render without count change:

→ Child does NOT re-render
(because function reference is same)

Note :

- * `useCallback()` **does not call** the function.
- * It only returns a memoized reference.
- * Always include all **dependencies**, otherwise you may get **stale** values.
- * It is useful only when the function is passed to optimized child components (e.g., `React.memo`).

5.8.2 Common Use Cases

- ✓ Passing callback functions to child components.
- ✓ Preventing child re-renders when callback reference changes.
- ✓ Using callbacks in `useEffect()` dependency arrays.
- ✓ Optimizing performance in large or complex applications.

Key Takeaway :



`useCallback()` helps you **memoize** callback functions and return the same reference unless its **dependencies change**, reducing unnecessary re-renders and **improving performance**.



5.9 useReducer()

- * useReducer() is a React Hook that lets you manage **complex state logic**.
- * It is an **alternative to useState()** for state with **multiple sub-values** or when the next state depends on the previous state.
- * It uses a **reducer function** to update the state.
- * It is inspired by **Redux** but built into React.
- * It returns the **current state** and a **dispatch function**.

Syntax :

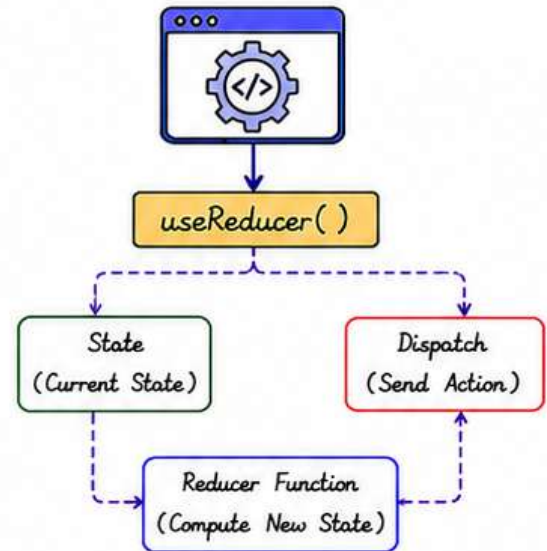
```
const [state, dispatch] = useReducer(reducer, initialState);
```

reducer → Function that specifies how the state changes based on an action.

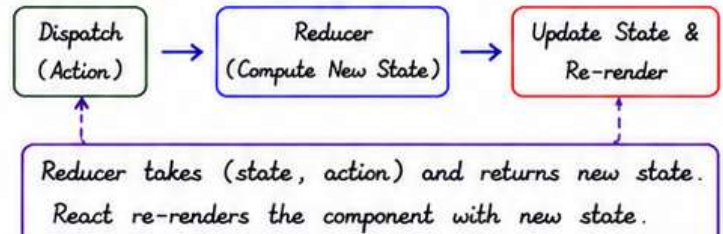
initialState → The initial state value.

state → The current state.

dispatch → Function used to send actions.



How useReducer() Works

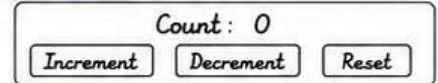


5.9.1 Example

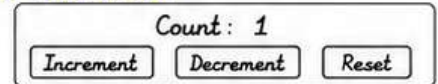
```
import React, { useReducer } from "react";
// 1. Initial State
const initialState = { count: 0 };
// 2. Reducer Function
function reducer(state, action) {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    case "reset":
      return initialState;
    default:
      return state;
  }
}
// 3. Component
function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <div>
      <h2>Count: {state.count}</h2>
      <button onClick={() => dispatch({ type: "increment" })}>Increment</button>
      <button onClick={() => dispatch({ type: "decrement" })}>Decrement</button>
      <button onClick={() => dispatch({ type: "reset" })}>Reset</button>
    </div>
  );
}
export default Counter;
```

Output :

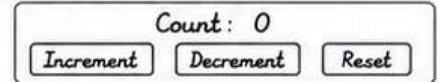
Initial Render:



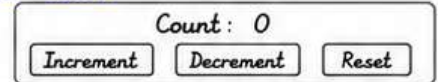
After Increment:



After Decrement:



After Reset:



Note :

- * useReducer() is useful when state logic is **complex** or involves **multiple sub-values**.
- * Reducer must be a **pure function**.
- * Actions are plain objects with a **type** property and optional **payload**.

5.9.2 Common Use Cases

- ✓ Managing complex state with multiple sub-values.
- ✓ State transitions based on specific actions.
- ✓ Forms with many fields.
- ✓ Undo/Redo functionality.
- ✓ When next state depends on previous state or action.



Key Takeaway :



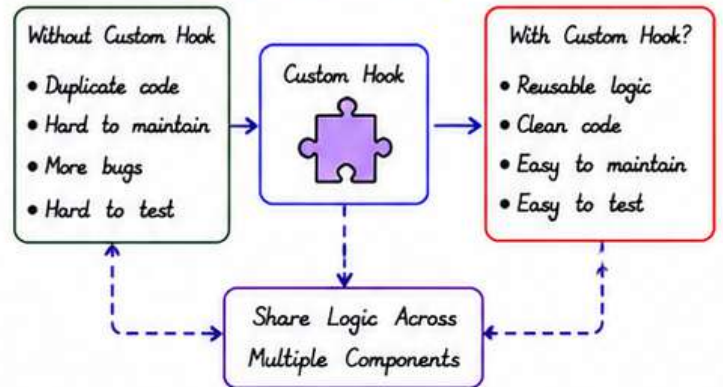
useReducer() helps you manage **complex state logic** using a **reducer function** and **actions**, making your code **predictable**, **scalable** and easier to maintain.



5.10 Custom Hooks

- * Custom Hooks are JavaScript functions whose names start with "use" and that may call other Hooks.
- * They let you **extract** and **reuse** component logic across multiple components.
- * Custom Hooks help to keep your code **DRY (Don't Repeat Yourself)** and more **maintainable**.
- * They don't work inside classes - only in **function components** or other Hooks.
- * Custom Hooks allow you to share **stateful logic**, **side effects**, **subscriptions**, etc.

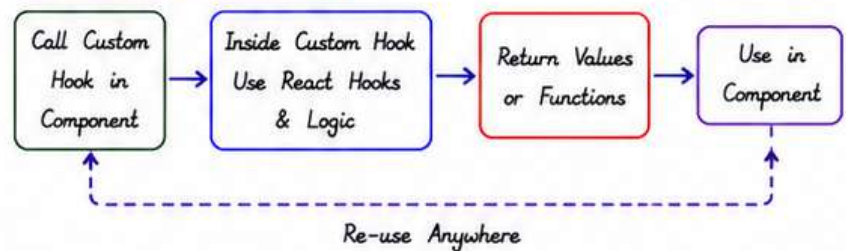
Why Custom Hooks?



Syntax :

```
function useCustomName(parameters) {  
  // Use other Hooks (useState, useEffect, etc.)  
  // Add your custom logic here  
  return { value1, value2, ..., functionName };  
}
```

How Custom Hooks Work



5.10.1 Example - Custom Hook : useCounter()

```
// Custom Hook  
import { useState } from "react";  
function useCounter(initialValue = 0, step = 1) {  
  const [count, setCount] = useState(initialValue);  
  const increment = () => setCount(count + step);  
  const decrement = () => setCount(count - step);  
  const reset = () => setCount(initialValue);  
  return { count, increment, decrement, reset };  
}  
export default useCounter;
```

Using the Custom Hook in a Component

```
import React from "react";  
import useCounter from "../useCounter";  
function Counter() {  
  const { count, increment, decrement, reset } =  
    useCounter(10, 2);  
  return (  
    <div>  
      <h2>Count: {count}</h2>  
      <button onClick={increment}>+2</button>  
      <button onClick={decrement}>-2</button>  
      <button onClick={reset}>Reset</button>  
    </div>  
  );  
}  
export default Counter;
```

5.10.2 Key Points

- * Custom Hooks always start with "use".
- * Call other Hooks only inside custom Hooks.
- * Return values or functions to be used in components.
- * Can accept parameters and return multiple values.
- * Helps in code reusability and separation of concerns.

5.10.3 Common Use Cases

- ✓ Sharing stateful logic (e.g., counters, forms).
- ✓ Fetching data from APIs.
- ✓ Handling subscriptions (WebSocket, events).
- ✓ Managing localStorage / sessionStorage.
- ✓ Reusing side effects (logging, analytics, etc.).



Key Takeaway :



Custom Hooks let you **extract**, **reuse** and share logic across components, making your React code **clean**, **modular** and **maintainable**.



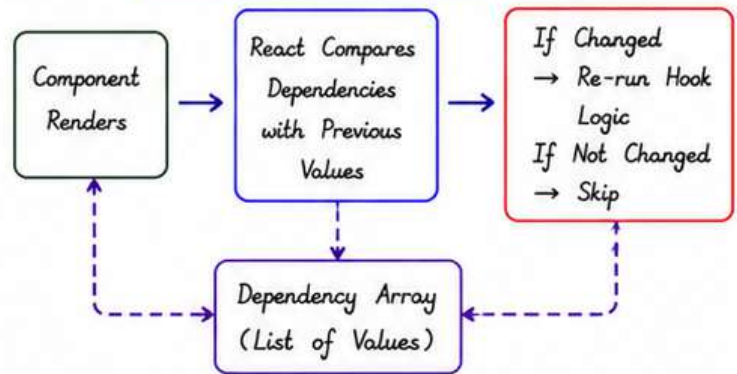
5.11 Hook Dependencies

- Many React Hooks like `useEffect()`, `useMemo()`, `useCallback()` accept a second argument called dependencies (dependency array).
- React uses this array to decide when the Hook should re-run or re-compute.
- It helps prevent unnecessary executions and improves performance.
- Correct dependency management is crucial to avoid bugs, stale values and infinite loops.

Syntax :

```
useEffect(() => {  
  // effect logic  
}, [dep1, dep2, ...]);  
  
useMemo(() => computeValue, [dep1, dep2, ...]);  
  
useCallback(() => fn, [dep1, dep2, ...]);
```

How Dependency Array Works



Dependency Array Cases

Dependency Array	Meaning	Behavior
<code>[]</code> (empty array)	No dependencies	Runs once on mount (<code>useEffect</code>), value cached once (<code>useMemo/useCallback</code>)
<code>[a, b]</code>	Depends on a and b	Re-runs when 'a' or 'b' changes
No array	No dependencies specified	Runs on every render

5.11.1 Example - `useEffect()` with Dependencies

```
import React, { useState, useEffect } from "react";  
function User({ userId }) {  
  const [user, setUser] = useState(null);  
  
  // useEffect will run when userId changes  
  useEffect(() => {  
    console.log("Fetching user with id:", userId);  
    // fetch user data here  
  }, [userId]); // dependency array  
  
  return (  
    <div> User ID: {userId}</div>  
  );  
}  
// If userId doesn't change, effect will not re-run
```

5.11.2 Stale Closure Problem

If you omit dependencies, the Hook may use stale values from previous renders.

✗ Wrong (Missing Dependency)

```
useEffect(() => {  
  console.log(count); // may log old value  
}, []); // count missing!
```

✓ Correct

```
useEffect(() => {  
  console.log(count); // always latest  
}, [count]);
```



5.11.3 Tips

- Always include all values used inside the Hook that can change over time.
- Use ESLint (`eslint-plugin-react-hooks`) to get warnings for missing dependencies.
- For functions/objects, use `useCallback` or `useMemo` to stabilize references if needed.
- If you intentionally want to run only once, use an empty array `[]`.



Key Takeaway :



Dependency arrays help React Hooks run only when needed. Use them correctly to avoid performance issues, stale values and infinite loops.

5.11.4 Dependency Usage in Common Hooks

Hook	Uses Dependency Array For
<code>useEffect()</code>	Controls when side effects run
<code>useMemo()</code>	Controls when a value is recalculated
<code>useCallback()</code>	Controls when a function is recreated



Right Dependencies
Better Performance



5.12 Side Effects

- Side Effects are operations that affect something **outside the component** or outside the render cycle.
- In React, we handle side effects using the **useEffect()** Hook.
- React components and Hooks must be **pure** functions. Side effects are separated and run **after rendering**.
- Examples:** Fetching data, updating the DOM, subscriptions, timers, logging, etc.

Syntax :

```
useEffect(() => {  
  // Side effect code here  
  // Optional: Cleanup function  
  return () => {  
    // Cleanup code (runs before effect re-runs  
    // or when component unmounts)  
  };  
}, [dependencies]);
```

5.12.1 Example - Fetch Data on Mount

```
import React, { useState, useEffect } from "react";  
function UserList() {  
  const [users, setUsers] = useState([]);  
  const [loading, setLoading] = useState(true);  
  
  useEffect(() => {  
    // Side Effect: Fetch data from API  
    fetch("https://jsonplaceholder.typicode.com/users")  
      .then(res => res.json())  
      .then(data => {  
        setUsers(data);  
        setLoading(false);  
      });  
  }, []); // Empty dependency: run only once on mount  
  if (loading) return <p>Loading...</p>;  
  return (  
    <div>  
      <h2>Users</h2>  
      <ul>  
        {users.map(user => (  
          <li key={user.id}>{user.name}</li>  
        ))}  
      </ul>  
    </div>  
  );  
}  
export default UserList;
```

5.12.3 Tips & Best Practices

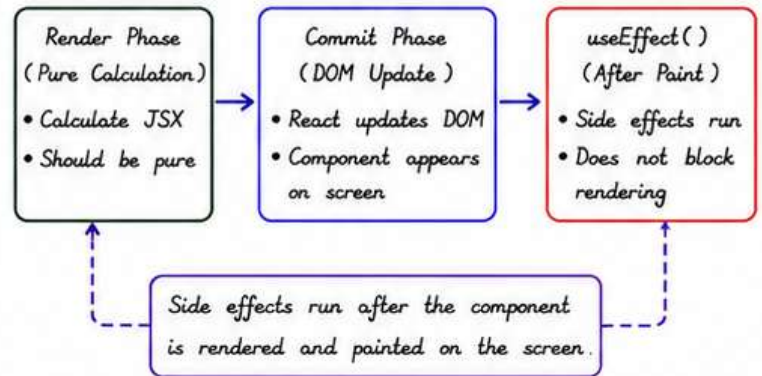
- Always use **useEffect** for side effects, not during rendering.
- Include correct dependencies to avoid bugs.
- Return a cleanup function when needed.
- Avoid unnecessary side effects to keep components fast and predictable.
- Keep side effects small and focused.

Key Takeaway :

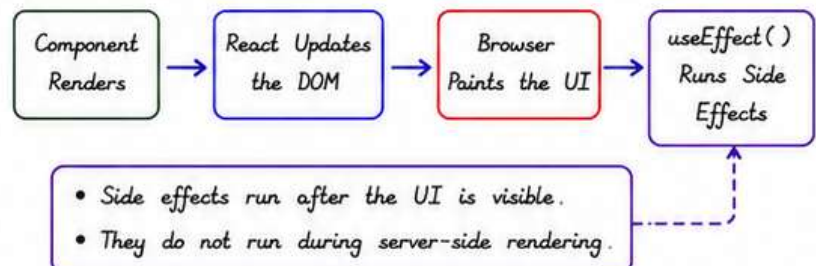


Side effects let your React components interact with the outside world. Use **useEffect()** correctly to keep your app efficient, bug-free and easy to maintain.

Where Side Effects Happen



How Side Effects Work



5.12.2 Common Side Effects

	1. Fetching Data Get data from APIs or databases.
	2. Subscriptions Subscribe to events, WebSockets, or observables.
	3. Timers Set timeouts or intervals.
	4. DOM Manipulation Directly interacting with the DOM (rare in React).
	5. Local Storage / Session Storage Save or read values from browser storage.
	6. Logging / Analytics Send logs, track events, or analytics data.

5.12.4 Cleanup on Unmount

If your effect creates subscriptions, timers, or event listeners, clean them up to avoid memory leaks.

Example:

```
useEffect(() => {  
  const timer = setInterval(() => console.log("Hi"), 1000);  
  return () => clearInterval(timer);  
}, []); // Cleanup runs on unmount
```



Fetch



Subscribe



Timer



Storage



Analytics



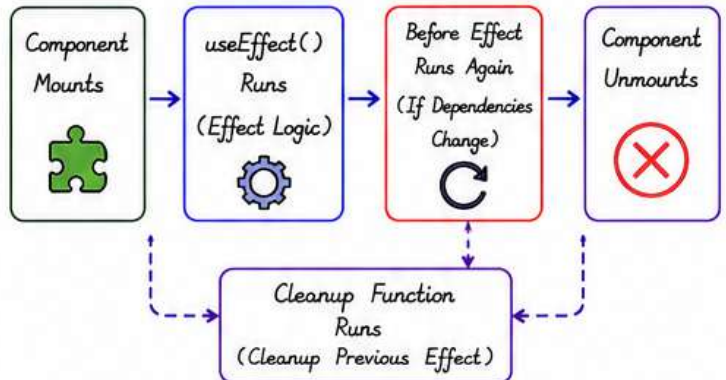
5.13 Cleanup Functions

- * A **cleanup function** in React is a function returned from `useEffect()`.
- * It runs when the component unmounts or before the effect runs again (when **dependencies change**).
- * It is used to clean up **subscriptions**, **timers**, **event listeners**, or any resources that were created.
- * Cleanup prevents **memory leaks**, unwanted behavior and improves **performance**.

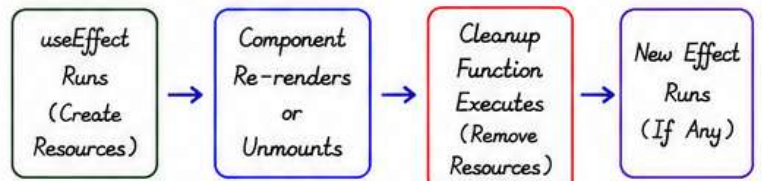
Syntax :

```
useEffect(() => {  
  // Effect logic (runs after render)  
  // e.g., subscribe, set timer, add event listener  
  
  return () => {  
    // Cleanup logic  
    // Runs before effect re-runs or when  
    // component unmounts  
  };  
}, [dependencies]);
```

When Cleanup Function Runs?



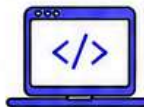
How Cleanup Works



- * Cleanup runs before the next effect execution.
- * Ensures app stays clean and leak-free.

5.13.1 Example - Event Listener with Cleanup

```
import React, { useEffect, useState } from "react";  
function WindowSize() {  
  const [width, setWidth] = useState(window.innerWidth);  
  
  useEffect(() => {  
    function handleResize() {  
      setWidth(window.innerWidth);  
    }  
    window.addEventListener("resize", handleResize);  
    // Cleanup: remove event listener  
    return () => {  
      window.removeEventListener("resize", handleResize);  
    };  
  }, []); // Empty array → run once on mount  
  
  return (  
    <div>  
      <h2>Window Width: {width}px</h2>  
    </div>  
  );  
}  
export default WindowSize;
```



5.13.2 Example - setInterval with Cleanup

```
import React, { useEffect, useState } from "react";  
function Timer() {  
  const [count, setCount] = useState(0);  
  
  useEffect(() => {  
    const id = setInterval(() => {  
      setCount(count => count + 1);  
    }, 1000);  
  
    // Cleanup: clear interval  
    return () => clearInterval(id);  
  }, []); // Run once on mount  
  
  return <h2>Count: {count}</h2>;  
}  
export default Timer;
```



5.13.3 Common Use Cases

- ✓ Removing event listeners
- ✓ Clearing timers (setTimeout / setInterval)
- ✓ Cancelling network requests
- ✓ Unsubscribing from subscriptions (e.g., WebSocket)
- ✓ Cleaning up resources (e.g., media streams)



5.13.4 Best Practices

- ✓ Always return a cleanup function when your effect creates side effects.
- ✓ Include correct dependencies in the array.
- ✓ Use cleanup to prevent memory leaks.
- ✓ Keep cleanup logic simple and focused.
- ✓ Test your components for mount, update and unmount.



Key Takeaway :



Cleanup functions in `useEffect()` are essential for managing resources. They run before the next effect or on unmount, keeping your React app **efficient**, **bug-free** and **memory-leak free**.

