## React Hooks

- React Hooks revolutionized the way we build React applications.
- They offer a cleaner, more modular approach to managing state, lifecycle, and reusable logic in functional components.
- Without hooks, developers face challenges in writing maintainable, performant, and reusable code. Hence, adopting hooks is highly recommended for modern React development.
- React Hooks were introduced in React 16.8 as a way to use state and other React features in functional components.
- Prior to hooks, managing state and lifecycle methods was only possible in class components. This often led to more complex, less readable code.
- React Hooks are functions that let you "hook into" React's state and lifecycle features directly in functional components.
- They enable developers to manage side effects, lifecycle events, and state seamlessly without needing to use class components.

### Why Hooks Were Introduced

- **Reusability of Logic:** Class components made it difficult to share reusable stateful logic between components. Hooks allow encapsulating logic in custom hooks, making it easier to share across components.
- **Cleaner Code:** Functional components with hooks are concise and easier to read compared to class components.
- **Avoiding "Wrapper Hell":** Before hooks, HOCs (Higher-Order Components) and render props were used to inject functionality, leading to deeply nested hierarchies. Hooks simplify this problem.
- **Easy Testing:** Functional components with hooks are easier to test as they behave like pure functions.
- **Boilerplate Code:** Earlier, Developers had to bind methods to the component instance (this) manually in class components. With hooks, No need for binding this.
- **Improved Component Structure:** With hooks, there's no need for multiple lifecycle methods for managing side effects. Hooks like useEffect handle side effects in a unified way.
- **Fragmented Logic:** before hooks, Logic was often split across multiple lifecycle methods (componentDidMount, componentDidUpdate, componentWillUnmount), making components harder to understand and maintain.

**Problems Without Hooks**

- **Class Components Complexity:** Managing state and lifecycle methods was cumbersome. Multiple lifecycle methods were often used to handle a single feature, making the code difficult to maintain.
- **Verbose Code:** Boilerplate code like binding 'this' was necessary in class components.
- **Hard-to-Reuse Logic:** Reusing stateful logic required complex patterns like HOCs or render props.
- **Context API Complexity:** Using the Context API in class components required contextType or Consumer patterns, which were less intuitive.

**Advantages of Using React Hooks**

- **Simplified State Management:** Hooks like useState make it straightforward to add state to functional components.
- **Unified Side Effects:** useEffect combines functionalities of componentDidMount, componentDidUpdate, and componentWillUnmount.
- **Enhanced Performance:** Functional components are generally more performant than class components. Hooks minimize unnecessary re-renders with memoization (useMemo, useCallback).
- **Improved Composition:** Custom hooks enable easy sharing of logic without restructuring components.

**Hooks Philosophy**

React Hooks are guided by the principles of simplicity, reusability, and alignment with React's declarative nature. They aim to:

- Replace complex class components with simple, functional ones.
- Enable reusable and modular stateful logic.
- Provide seamless integration with modern React features, including Concurrent Rendering.
- Reduce boilerplate and improve the developer experience.

This philosophy ensures that React applications are scalable, maintainable, and future-proof, empowering developers to write cleaner and more efficient code.

React Hooks are a revolutionary addition to React that allow developers to use state and other React features in functional components. They follow a core philosophy designed to address common challenges and improve the developer experience.

Let's understand the philosophy of react hooks by understanding the following points:

- **Encapsulation of Logic:** Hooks allow developers to extract and reuse stateful logic across multiple components by creating custom hooks. This promotes modular, maintainable code, where logic can be encapsulated separately from the component's UI.

  Example: A useAuth custom hook could manage authentication logic for multiple components.

- **Side Effects Management:** Managing side effects (like data fetching, subscriptions, and timers) is simplified with hooks like useEffect. useEffect replaces lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount, consolidating them into a single, declarative API. This ensures side effects are efficiently managed without relying on imperative patterns.

- **Declarative Composition:** Hooks embrace React's declarative paradigm, focusing on describing "what" should happen rather than "how" to do it. This simplifies complex UI logic and improves readability, reducing reliance on imperative state management.

- **Concurrent Mode Compatibility (React 18+):** Hooks are designed to work seamlessly with React's Concurrent Rendering features. This allows for smoother user experiences by enabling deferred updates, transitions, and better handling of rendering priorities.

- **Elimination of Class Components:** Hooks eliminate the need for class components, making React simpler and easier to learn. By using functional components with hooks, developers avoid pitfalls like this binding or confusing lifecycle methods.

- **Improved Readability and Organization:** By separating UI logic from state management, hooks make components more readable and focused. Developers can keep their functional components small, handling only UI concerns, while logic resides in hooks.

- **Rich Ecosystem of Custom Hooks:** Hooks have led to the creation of a vibrant ecosystem of custom hooks. Examples include hooks for form handling (useForm), API requests (useFetch), authentication (useAuth), and more, which accelerate development by solving common problems.
- **Avoidance of "Wrapper Hell":** Before hooks, developers often relied on patterns like Higher-Order Components (HOCs) or render props, which could lead to deeply nested and hard-to-manage components (a.k.a. "wrapper hell").
  Hooks eliminate this by allowing logic to be shared directly through custom hooks, keeping the component tree shallow and clean.
- **Simplified Lifecycle Management:** Hooks like useEffect combine the logic for mounting, updating, and cleanup into a single API, reducing the boilerplate and complexity of managing lifecycle methods. This simplifies code while offering more control over side effects.
- **Simplified Testing:** With hooks, stateful logic is separated into reusable functions, making it easier to test components and their underlying logic independently. This aligns well with modern testing practices.
- **Dynamic State Management:** Hooks like useReducer and useState enable developers to manage both simple and complex state logic without requiring additional libraries. This flexibility allows for dynamic and powerful state handling directly within React.
- **Backward Compatibility:** Hooks were introduced in React 16.8 and are fully backward-compatible with class components. This means developers can incrementally adopt hooks without rewriting legacy components.
- **Reduced Boilerplate:** Hooks remove the need for verbose class syntax and lifecycle methods, reducing boilerplate code and improving productivity.
  For example, instead of defining separate lifecycle methods, a single useEffect handles initialization, updates, and cleanup.
- **TypeScript Compatibility:** Hooks are TypeScript-friendly, enabling better typing and error prevention during development. This is especially valuable in large-scale applications where type safety is crucial.
- **Functional Programming Mindset:** Hooks promote a functional programming mindset, where components are treated as pure functions of their props and state. This improves the predictability and maintainability of React applications.

- **Encouragement of Innovation:** By simplifying state management and logic encapsulation, hooks encourage developers to experiment and innovate. New patterns and libraries have emerged, enabling more efficient and creative solutions for complex problems.

**Important React Hooks (Till React Version 19)**

Basic Hooks

- **useState:** Manages local state in a functional component.

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

- **useEffect:** Handles side effects like fetching data, DOM manipulation, or subscriptions.

```
useEffect(() => {
    console.log('Component mounted or updated');
    return () => console.log('Cleanup on unmount');
}, [dependencies]);
```

- **useContext:** Consumes context values without using Consumer.

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

Additional Hooks

- **useReducer:** Alternative to useState for complex state logic or state dependent on previous values.

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

- **useCallback:** Memoizes callback functions to prevent unnecessary re-creation.

```
const memoizedCallback = useCallback(() => {
        doSomething(a, b);
}, [a, b]);
```

- **useMemo:** Memoizes computed values to optimize performance.

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

- **useRef:** Accesses DOM nodes or persists values across renders without causing re-renders.

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

- **useImperativeHandle:** Customizes the instance value exposed by useRef in parent components.

```
useImperativeHandle(ref, () => ({ customMethod }));
```

- **useLayoutEffect:** Similar to useEffect, but fires synchronously after all DOM mutations.

```
useLayoutEffect(() => {
  // DOM manipulation
}, []);
```

- **useDebugValue:** Adds labels to custom hooks for debugging.

```
useDebugValue(value, formatFunction);
```

Newer Hooks

- **useId (React 18+):** Generates unique IDs for accessibility.

```
const id = useId();
```

- **useTransition (React 18+):** Manages UI transitions without blocking the UI.

```
const [isPending, startTransition] = useTransition();
```

- **useDeferredValue (React 18+):** Defers updating a value to improve performance in high-stress updates.

```
const deferredValue = useDeferredValue(value);
```

- **useSyncExternalStore (React 18+):** Subscribes to external stores to read values in a concurrent-safe way.

```
const state = useSyncExternalStore(subscribe, getSnapshot);
```

- **useInsertionEffect (React 18+):** Optimizes CSS-in-JS libraries for injecting styles before rendering.

```
useInsertionEffect(effect: () => (void | (() => void)), deps?: DependencyList);
```

## useState

- useState is a React Hook introduced in React 16.8 that allows functional components to manage state.
- It enables you to add state to functional components, which were previously stateless.
- useState returns a state variable and a function to update that variable.

### Why useState is used?

- To make functional components stateful.
- It allows components to respond dynamically to user interactions or changes by maintaining and updating internal state.
- Simplifies state management compared to class components.

Syntax:

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

where,

state: Holds the current value of the state.
setState: Updates the state and triggers a re-render of the component.
initialValue: The initial value assigned to the state.

Example: Counter Component

App.js

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

function App() {
// Declare state variable 'count' with initial value 0
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      <button onClick={() => setCount(count - 1)}>Decrement</button>
    </div>
  );
}
export default App;
```

**Understanding State in Functional Components**

- State is a built-in React object used to store data that can change over time and influence the rendering of the component.
- In Functional Components, State is managed using hooks like useState.
- Hooks provide a more concise and readable way to handle state compared to class components.
- The component re-renders when the state is updated.

Example: Toggling Theme

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

function App() {
  const [isDark, setIsDark] = useState(false);
  return (
    <div style={{ backgroundColor: isDark ? "black" : "white", color: isDark ? "white"
: "black" }}>
      <p>The current theme is {isDark ? "Dark" : "Light"}.</p>
      <button onClick={() => setIsDark(!isDark)}>Toggle Theme</button>
    </div>
  );
}
export default App;
```

**Comparison**

| Aspect | Class Components | Functional Components |
|---|---|---|
| Syntax | State is managed using this.state. | State is managed using useState. |
| Initialization | State is initialized in the constructor. | State is initialized with useState. |
| Updating State | Use this.setState to update state. | Use the state updater function returned by useState. |
| Code Length | Longer and verbose. | Shorter and more concise. |
| Lifecycle Integration | Requires lifecycle methods like componentDidMount. | Combines with useEffect for side effects. |

## Example: Class Component Counter

```
import React, { Component } from "react";
class App extends Component {
 constructor() {
  super();
  this.state = { count: 0 };
 }
 increment = () => {
  this.setState({ count: this.state.count + 1 });
 };
 decrement = () => {
  this.setState({ count: this.state.count - 1 });
 };
 render() {
  return (
   <div>
    <p>Count: {this.state.count}</p>
    <button onClick={this.increment}>Increment</button>
    <button onClick={this.decrement}>Decrement</button>
   </div>
  );
}}
export default App;
```

### Declaring State Variables and Their Setters

The useState hook in React allows you to declare state variables within functional components. When you call useState, you provide an initial value for the state, and it returns two items:

- The current state value.
- A function to update the state (setter function).

Syntax:

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

### Updating State with the Setter Function

The setter function returned by useState allows you to update the state. When you use the setter function, React re-renders the component with the updated state.

**Rules for Updating State:**

- Do Not Modify State Directly:
  - Never modify state variables directly (e.g., state = newValue) as it won't trigger a re-render.
  - Always use the setter function (e.g., setState(newValue)).
- State Updates Can Be Based on Previous State: When updating state based on the previous value, pass a function to the setter function.

Example: Updating State Based on Previous State

```
function App() {
  const [count, setCount] = useState(0);
  const increment = () => {
 // Updates count based on its previous value
    setCount((prevCount) => prevCount + 2);
  };
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment by 2</button>
    </div>
  );
}
```

**Combining Multiple State Variables**

You can declare multiple state variables with useState in a single component.

Example: Managing Form Input State

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

function App() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
        alert(`Name: ${name}, Email: ${email}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>Name: </label>
        <input type="text" value={name} onChange={(e) =>
setName(e.target.value)} />
      </div>
      <div>
        <label>Email: </label>
        <input type="email" value={email} onChange={(e) =>
setEmail(e.target.value)} />
      </div>
      <button type="submit">Submit</button>
    </form>
  );
}

export default App;
```

Example: Managing Strings

```
import React, { useState } from "react";
function App() {
  const [name, setName] = useState(""); // Initial state is an empty string
  return (
    <div>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)} // Update state with input value
        placeholder="Enter your name"
      />
      <p>Your name is: {name}</p>
    </div>
  );
}
export default App;
```

Example : Managing Booleans

```
import React, { useState } from "react";
function App() {
  const [isVisible, setIsVisible] = useState(false); // Initial state is false

  return (
    <div>
      <button onClick={() => setIsVisible(!isVisible)}>
        {isVisible ? "Hide" : "Show"} Text
      </button>
      {isVisible && <p>Welcome To Learn2Earn Labs Training Institute.</p>}
    </div>
  );
}
export default App;
```

## Example: Managing Objects

```
import React, { useState } from "react";
function App() {
  const [profile, setProfile] = useState({ name: "", age: 0 });
  const updateProfile = () => {
    setProfile((prevProfile) => ({
      ...prevProfile, // Copy existing properties
      age: prevProfile.age + 1, // Update the age property
    }));
  };
  return (
    <div>
      <p>Name: {profile.name}</p>
      <p>Age: {profile.age}</p>
      <button onClick={() => setProfile({ ...profile, name: "Shikha" })}>Set
Name</button>
      <button onClick={updateProfile}>Increase Age</button>
    </div>
  );}
export default App;
```

## Example : Managing Arrays

```
import React, { useState } from "react";
function App() {
  const [todos, setTodos] = useState([]);
  const addTodo = () => {
    setTodos([...todos, { id: todos.length + 1, task: "My task to complete" }]);
  };
  return (
    <div>      <ul>
        {todos.map((todo) => (
          <li key={todo.id}>{todo.task}</li>        ))}
      </ul>
      <button onClick={addTodo}>Add Todo</button>
    </div>  );}
export default App;
```

**Passing a Function to the Setter for State Updates (Functional Updates)**

When updating state based on the previous state value, you should pass a function to the setter function returned by useState. This approach ensures that React works with the most up-to-date state, avoiding potential bugs caused by asynchronous updates.

syntax:

```
setState((prevState) => {
  // Logic to calculate new state
  return newState;
});
```

Where,

prevState: Represents the previous state value.

newState: The updated value that will replace the current state.

Example: Counter with Functional Updates

```
import React, { useState } from "react";
function App() {
  const [count, setCount] = useState(0);
  const increment = () => {
    setCount((prevCount) => prevCount + 1); // Update based on the previous state
  };
  const decrement = () => {
    setCount((prevCount) => prevCount - 1);
  };
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}
export default App;
```

Explanation:

- setCount receives a function where prevCount is the current state value.
- React guarantees that the prevCount passed to this function is the latest state, ensuring a consistent update even if multiple updates are queued.

**Why Use Functional Updates?**

- **When State Depends on Previous State:** Any logic that calculates new state based on the old state requires functional updates to avoid using outdated values.
- **Avoid Issues with Asynchronous Updates:** React batches state updates for performance reasons. This means that multiple setState calls might not immediately update the state. Functional updates ensure you work with the latest state value.

Example of Potential Bug Without Functional Updates:

```
import React, { useState } from "react";
function App() {
  const [count, setCount] = useState(0);
  const handleClick = () => {
    setCount(count + 1); // Uses old `count` value
    setCount(count + 1); // Still uses the old `count` value
  };
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment Twice</button>
    </div>
  );
}
export default App;
```

Problem: Clicking the button increments the count by 1 instead of 2 because both setCount calls use the same stale count value.

Solution: Use Functional Updates

```
const handleClick = () => {
  setCount((prevCount) => prevCount + 1);
  setCount((prevCount) => prevCount + 1);
};
```

Result: Now the count is incremented by 2 as expected.

**When to Use Functional Updates?**

- **For Multiple Consecutive Updates:** If multiple updates depend on the same state value, functional updates ensure consistency.

  Example: Like Counter

```
import React, { useState } from "react";
function App() {
  const [likes, setLikes] = useState(0);
  const addLikes = () => {
    setLikes((prevLikes) => prevLikes + 1);
    setLikes((prevLikes) => prevLikes + 1);
  }
  return (
    <div>
      <p>Likes: {likes}</p>
      <button onClick={addLikes}>Add Likes</button>
    </div>
  );
}
export default App;
```

  Without functional updates, only one like would be added. With functional updates, both increments are processed correctly.

- **For Complex Calculations Based on Previous State:** If state updates involve more logic than simple arithmetic, functional updates help streamline the calculation.

  Example: Dynamic Step Counter

```
import React, { useState } from "react";
function App() {
  const [count, setCount] = useState(0);
  const [step, setStep] = useState(1);
  const increment = () => {
    setCount((prevCount) => prevCount + step);
  };
  return (
    <div>
    <input
```

```
        type="number"
        value={step}
        onChange={(e) => setStep(Number(e.target.value))}
      />
      <button onClick={increment}>Increment by Step</button>
      <p>Count: {count}</p>
    </div>
  );
}
export default App;
```

The increment function dynamically calculates the new count based on the current step value.

**Updating State Based on User Inputs**

The useState hook can manage and update state dynamically as users interact with components like text fields, checkboxes, or buttons. This is typically done by listening to events like onChange, onClick, etc., and using the setter function to update the state.

Example: Text Field Input

```
import React, { useState } from "react";
function App() {
  const [text, setText] = useState(""); // Initialize state with an empty string
  const handleChange = (e) => {
    setText(e.target.value); // Update state with the input value
  };
  return (
    <div>
      <input      type="text"      value={text}
        onChange={handleChange} // Trigger state update on input change
        placeholder="Type something"
      />
      <p>You typed: {text}</p>
    </div>
  );}
export default App;
```

Example: Checkbox Input

```
import React, { useState } from "react";
function App() {
  const [isChecked, setIsChecked] = useState(false);

  const toggleCheckbox = () => {
    setIsChecked((prev) => !prev); // Toggle the checkbox state
  };

  return (
    <div>
      <label>
        <input
          type="checkbox"
          checked={isChecked}
          onChange={toggleCheckbox}
        />
        Accept Terms and Conditions
      </label>
      <p>{isChecked ? "Accepted" : "Not Accepted"}</p>
    </div>
  );
}
export default App;
```

**Event-Driven State Changes**

State can be updated in response to various user-triggered events like button clicks, mouse actions, or form submissions.

Example: Form Submission

```
import React, { useState } from "react";
function App() {
  const [name, setName] = useState("");
  const [email, setEmail] = useState("");
  const [submitted, setSubmitted] = useState(false);

  const handleSubmit = (e) => {
    e.preventDefault(); // Prevent page reload
    setSubmitted(true); // Update state to reflect submission
  };

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <div>
          <label>Name: </label>
          <input
            type="text"
            value={name}
            onChange={(e) => setName(e.target.value)} // Update name state
          />
        </div>
        <div>
          <label>Email: </label>
          <input
            type="email"
            value={email}
            onChange={(e) => setEmail(e.target.value)} // Update email state
          />
        </div>
        <button type="submit">Submit</button>
      </form>
```

```
      {submitted && (
        <p>
          Details Submitted
          <br/>
          Name: {name}, Email: {email}
        </p>
      )}
    </div>
  );
}
export default App;
```

Example: Mouse Events

```
import React, { useState } from "react";
function App() {
  const [position, setPosition] = useState({ x: 0, y: 0 });

  const handleMouseMove = (e) => {
    setPosition({ x: e.clientX, y: e.clientY }); // Update state with cursor position
  };

  return (
    <div style={{ height: "200px", border: "1px solid black" }}
onMouseMove={handleMouseMove}>
      <p>Mouse Position: X: {position.x}, Y: {position.y}</p>
    </div>
  );
}
export default App;
```