

# Lifecycle methods(Class Component)

## 1. What are lifecycle methods in React class components? Describe the phases of a component's lifecycle.

**ANS:** In React, lifecycle methods are special methods that get called automatically during life of a class component. These methods allow you to run specific code at different stages of the component's life, like when it is created, updated, or removed from the screen.

Phases of a component's lifecycle:

There are three main phases in React component's lifecycle:

### 1. Mounting

- This phase happens when the component is first rendered on the page. The following methods are called during this phase.
  - `Constructor()`: Initialize the component that is setting up state and props.
  - `static getDerivedStateFromProps()`: Syncs state with props if needed before rendering
  - `render()`: Render the component's UI to the DOM.
  - `componentDidMount()`: Runs after the component's displayed on the screen. Great for API calls or setting up subscriptions.

### 2. Updating

This phase happens when the component's data (state or props) changes. The following methods are triggered.

- `static getDerivedStateUpdate()`: Updates state based on new props (called before rendering).
- `shouldComponentUpdate()`: Decides whether the component should re-render or not (useful for optimization).
- `render()`: Re-render the UI based on update data.
- `getSnapshotBeforeUpdate()`: Capture some information before the DOM is updated (To get the old value).
- `componentDidUpdate()`: Runs after the DOM has been updated. Ideal for performing actions based on changes (e.g. updating data in response to state changes)

### 3. unmounting

This phase happen when the component is no longer needed and is removed. The only methods in this phase is:

- `componentWillUnmount()`: Used to clean up things like event listeners, timers, or subscriptions to prevent memory leaks.

## 2. Explain the purpose of `componentDidMount()`, `componentDidUpdate()`, and `componentWillUnmount()`.

### ANS: `ComponentDidMount()`:

- It is called right after the component is added to the DOM (rendered for the first time).
- This is the perfect place to set up things that the component needs after being displayed.
- **Uses:**
  - Fetching data from an API.
  - Starting subscriptions.
  - Updating the DOM.
- **`componentDidUpdate()`**
  - It is called after the component is rendered because of changes to props or state.
  - It is used to perform actions based on updates (but not during the initial render)
  - **Uses**
    - Triggering further API calls when state/props change.
    - Updating the DOM based on changes.
    - Comparing previous props/state with the current ones to decide further actions.
- **`componentWillMount()`**
  - It is called before the component is removed from the DOM.
  - Clean up resources or subscriptions to prevent memory leaks.
  - **Uses:**
    - Remove event listeners.
    - Clearing timers or intervals.
    - Stopping network requests or WebSocket connections.

## 3. What are React hooks? How do `useState()` and `useEffect()` hooks work in functional components?

**ANS:** React hooks are special functions introduced in React that allows you to use state and lifecycle features in functional components.

Hooks make it easier to manage state, side effects, and other React features in a more readable and reusable way.

### 1. `useState()` Hook

- Allow you to add and manage state in a functional component.
- You initialize a state variables and a function to update it.
- The state value persists across renders and updater function.

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

- **state:** The current value of the state.
- **setState:** A function to update the state.
- **initialValue:** The initial value of the state can be a string, object, number, object, etc.

## 2. useEffect() Hook

- let you perform side effects in functional components.
- It is like combining componentDidMount, componentDidUpdate, and componentWillUnmount in one place.
- The hooks runs after the component renders, and it can optionally clean up or re-run based on dependencies.

```
jsx Copy code
useEffect(() => {
  // Effect code (e.g., API calls, subscriptions)
  return () => {
    // Cleanup code (e.g., removing listeners)
  };
}, [dependencies]);
```

## 4. What problems did hooks solve in React development? Why are hooks considered an important addition to React?

**ANS:** Before hooks were introduced in React 16.8, developers often relied on class components to manage state, lifecycle methods, and logic reuse.

However, class components had several problems that hooks effectively solved.

### 1. Complexity of class Components

- Class components required understanding of complex features like this binding, constructor functions, and lifecycle methods.

- Hooks allow developers to use state (useState) and lifecycle features (useEffect) directly in functional components, which are simpler and easier to work with.

## **2. Difficulty in Reusing Logic**

- Reusing stateful logic in class components was challenging.
- Developer had to use techniques like higher-order components or render props, which often led to messy and less readable code.
- Hooks like custom hooks enable without affecting the component hierarchy.

## **3. Lifecycle Management**

- Class components required splitting logic across multiple lifecycle methods.
- It was hard to related logic.
- With useEffect, you can combine all related logic in one place
- Cleanup (like unsubscribing) is handled in the same function, improving readability.

## **4. Large and Hard-to-Maintain Components**

- Class components often became larger and cluttered because all logic had to fit within a single class. This made them hard to maintain.
- Functional components with hooks let you break down logic into smaller, reusable parts(custom hooks), making components easier to manage and understand.

## **5. Code Duplication**

- Without hooks, developer often duplicated code when implementing similar logic in different components.
- Custom hooks eliminate duplication by allowing shared logic to be written once and reused.

## **Importance of hooks in react**

Hooks were game changer for React because they simplified the development process and made functional components much more powerful.

- Hooks remove the need for class components, reducing boilerplate code that is constructor, this, or lifecycle methods spread out.
- Custom hooks allow developers to extract, share, and reuse stateful logic across components without complex pattern like HOCs or render props.
- Hooks allow related code that is state management and effect to stay together, making components easier to understand and maintain.
- Previously, functional components were only used for rendering static UI. Hooks brought state and side effects to functional components, making them as powerful as class components

- Hooks are 100% backward compatible, so existing state and side effects, reducing errors and improving developer experience.

## 5. What are some common event handlers in React.js? Provide examples of `onClick`, `onChange`, and `onSubmit`.

**ANS:** In React, event are functions that get triggered when certain events happen, such as a user clicking a button, typing in a text field, or submitting a form.

React uses a camelCase naming for these event handlers.

### 1. `onClick`

- handles when a user clicks on an element like a button or a div.

```
jsx Copy code

import React, { useState } from 'react';

function ClickExample() {
  const [message, setMessage] = useState("Hello!");

  const handleClick = () => {
    setMessage("You clicked the button!");
  };


  return (
    <div>
      <p>{message}</p>
      <button onClick={handleClick}>Click Me</button>
    </div>
  );
}

export default ClickExample;
```

### 2. `onChange`

- handles when a user types in an input field or change the value of an input.

jsx

 Copy code

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

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

  const handleChange = (event) => {
    setName(event.target.value); // Updates the state with the current input value
  };

  return (
    <div>
      <input type="text" placeholder="Enter your name" onChange={handleChange} />
      <p>Your name is: {name}</p>
    </div>
  );
}

export default ChangeExample;
```

### 3. onSubmit

- handles when a user submits a form.