Component Life Cycle

In React, the component life cycle refers to the series of methods and stages that a component goes through from its creation (mounting) to its removal (unmounting). The life cycle is divided into three main phases:

1. **Mounting:** This phase occurs when a component is being inserted into the DOM. Key lifecycle methods/hooks here include:
    - **constructor:** Initializes state and props.
    - **componentDidMount:** Invoked immediately after a component is mounted, great for data fetching or subscriptions.
2. **Updating:** This phase takes place when a component's state or props change. Key lifecycle methods/hooks include:
    - **componentDidUpdate:** Called after a component updates. Useful for responding to prop or state changes.
    - **shouldComponentUpdate:** Allows you to prevent unnecessary updates (for class components).
    - useEffect: Can be used in functional components for side effects when state or props change.
3. **Unmounting:** This phase occurs when a component is being removed from the DOM. The key method/hooks include:
    - **componentWillUnmount:** Used for cleanup tasks like invalidating timers or cancelling network requests.
    - **Cleanup function in useEffect:** A cleanup function can be returned to handle cleanup when the component unmounts.

Understanding these life cycle stages is crucial for managing side effects, optimizing performance, and ensuring proper resource management in your React applications.

Let’s Deep Dive into them.

1. What is the Mounting Phase?

The **Mounting Phase** refers to the stage when a component is **created and inserted into the DOM** for the first time. It includes the initial setup such as:

- Defining state
- Setting up props
- Preparing the UI before rendering
- Performing actions like API calls, subscriptions, or DOM manipulation after the component appears on the screen

This phase runs **only once** when the component is rendered for the first time.

Methods and Hooks Involved

In Class Components

Order	Method	Purpose
1	`constructor()`	Initialize state and bind methods
2	`static getDerivedStateFromProps()`	Set state based on initial props
3	`render()`	Describe what to render
4	`componentDidMount()`	Run side effects like API calls, event listeners

In Functional Components (with Hooks)

Hook	Purpose
`useEffect(() => { ... }, [])`	Acts like **`componentDidMount()`**; runs once after the initial render

Real-World Use Cases

- Fetching data from an API
- Starting animations
- Setting focus to an input field
- Subscribing to events or WebSocket
- Setting up timers or intervals

Example – Class Component (Mounting Phase)

```
import React from "react";

class Welcome extends React.Component {
  constructor(props) {
    super(props);
    this.state = { user: null };
    console.log("Constructor called");
  }

  componentDidMount() {
    console.log("Component mounted");
    // Simulate API call
    setTimeout(() => {
      this.setState({ user: "John Doe" });
    }, 1000);
  }

  render() {
    return (
      <div>
        <h2>Welcome, {this.state.user || "Loading..."}</h2>
      </div>
    );
  }
}
```

Explanation:

- `constructor()` initializes the component with default state.
- `componentDidMount()` fetches the user data after the component is mounted.

Example – Functional Component (Mounting Phase with Hooks)

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

function Welcome() {
  const [user, setUser] = useState(null);

  useEffect(() => {
    console.log("Component mounted");
    setTimeout(() => {
      setUser("John Doe");
    }, 1000);
  }, []); // Empty dependency array = run once after mount
  return (
    <div>
      <h2>Welcome, {user || "Loading..."}</h2>
    </div>
  );
}
```

Explanation:

- `useEffect` with an empty array acts like `componentDidMount()`.
- The effect runs only once after the initial render.

2. What is the Updating Phase?

The **Updating Phase** occurs when a component **re-renders** due to changes in:

- **Props**
- **State**

This phase may happen multiple times during a component's life. Every time the state or props change, React evaluates whether the component needs to update the DOM.

Lifecycle Methods and Hooks

In Class Components

Order	Method	Purpose
1	**`static getDerivedStateFromProps()`**	Sync state with updated props (rare use)
2	**`shouldComponentUpdate()`**	Decide whether to proceed with re-render
3	**`render()`**	Re-render JSX based on updated props/state
4	**`getSnapshotBeforeUpdate()`**	Capture DOM snapshot before DOM updates
5	**`componentDidUpdate()`**	Run side effects after DOM updates

In Functional Components (with Hooks)

Hook	Purpose
`useEffect(() => { ... }, [dependencies])`	Reacts to specific state/prop changes

A functional component has no "shouldComponentUpdate", but React internally optimizes rendering. For deep control, use `React.memo()` or `useMemo()`.

Use Cases of the Updating Phase

- Reacting to changes in props (e.g., new data from parent)
- Re-fetching or re-processing data on state change
- Syncing scrolling position or resizing logic
- Running animations on data change
- Logging or debugging updates

Example – Class Component (Updating Phase)

```
import React from "react";

class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  shouldComponentUpdate(nextProps, nextState) {
    console.log("Should component update?", nextState.count);
    return true; // allows the update
  }

  componentDidUpdate(prevProps, prevState) {
    if (prevState.count !== this.state.count) {
      console.log("Count updated from", prevState.count, "to",
this.state.count);
    }
  }
  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={() => this.setState({ count: this.state.count + 1
})}>
          Increment
        </button>
      </div>
    );
  }
}
```

Explanation:

- `shouldComponentUpdate` allows logging and optimization.
- `componentDidUpdate` detects and reacts to changes in state.

Example – Functional Component (Updating Phase with Hooks)

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

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

  useEffect(() => {
    console.log("Count updated to", count);
    // Can run animation, API, etc. here
  }, [count]); // This effect runs ONLY when `count` changes

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```

Explanation:

- `useEffect` reacts specifically to changes in `count` due to `[count]` dependency.
- Acts similar to `componentDidUpdate`.

3. What is the Unmounting Phase?

The **Unmounting Phase** is the **final stage** of a component's lifecycle. It occurs **only once** – when a component is about to be **removed from the DOM**.

This is the best time to **clean up** any resources (like event listeners, timers, subscriptions, or API connections) that were created during the component's lifetime.

Lifecycle Methods and Hooks

In Class Components

Method	Purpose
`componentWillUnmount()`	Cleanup tasks like unsubscribing, clearing intervals, or cancelling requests

In Functional Components (with Hooks)

Hook Syntax	Purpose
`useEffect(() => { return () => { ...cleanup } }, [])`	Cleanup runs once on unmount if effect was initialized with empty dependencies

---

Use Cases of Unmounting Phase

- Clear `setInterval` / `setTimeout`
- Unsubscribe from WebSocket or event listeners
- Abort in-progress API requests
- Remove DOM event handlers
- Reset local/global application state

Example – Class Component (Unmounting Phase)

```
import React from "react";

class Timer extends React.Component {
  componentDidMount() {
    this.interval = setInterval(() => {
      console.log("Timer running...");
    }, 1000);
  }


  componentWillUnmount() {
    clearInterval(this.interval);
    console.log("Timer stopped and cleaned up");
  }


  render() {
    return <p>Timer running... check console</p>;
  }
}
```

Explanation:

- `componentWillUnmount()` is used to **clear the interval** and prevent memory leaks.

Example – Functional Component (Unmounting Phase with Hooks)

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

function Timer() {
  useEffect(() => {
    const interval = setInterval(() => {
      console.log("Timer running...");
    }, 1000);

    return () => {
      clearInterval(interval);
      console.log("Timer cleaned up");
    };
  }, []);

  return <p>Timer running... check console</p>;
}
```

Explanation:

- The cleanup function inside `useEffect` acts **like `componentWillUnmount`**.
- It ensures the timer is cleared when the component is removed.