

In React, the **component lifecycle** refers to the sequence of events that happen from the creation of a component to its removal from the DOM. Understanding this helps you manage side effects, optimize performance, and write cleaner code.

Three Main Phases

1. Mounting (Component is being created and inserted into the DOM)


- `constructor()` – Initializes state and binds methods.
- `getDerivedStateFromProps()` – Syncs state with props before rendering.
- `render()` – Returns JSX to display UI.
- `componentDidMount()` – Runs after the component is added to the DOM (great for API calls or subscriptions).

2. Updating (Component is re-rendered due to state/props change)

- `getDerivedStateFromProps()` – Called again before re-render.
- `shouldComponentUpdate()` – Lets you control whether re-rendering is needed.
- `render()` – Re-renders the UI.
- `getSnapshotBeforeUpdate()` – Captures info (like scroll position) before DOM updates.
- `componentDidUpdate()` – Runs after the update is flushed to the DOM.

3. Unmounting (Component is removed from the DOM)

- `componentWillUnmount()` – Cleanup tasks like clearing timers or unsubscribing from services.

 In **functional components**, you use the `useEffect()` Hook to mimic lifecycle behavior:

```
useEffect(() => {  
  // componentDidMount + componentDidUpdate logic  
  return () => {  
    // componentWillUnmount logic  
  };  
}, [dependencies]);
```

In React, the **constructor()** is the **first method called** during the **mounting phase** of a class component's lifecycle. It's where you typically:

- **Initialize state**
- **Bind event handlers**
- **Set up any initial values**

Syntax

```

class MyComponent extends React.Component {
  constructor(props) {
    super(props); // Always call super(props) first!
    this.state = {
      count: 0,
    };
    this.handleClick = this.handleClick.bind(this);
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
  handleClick() {
    this.setState({ count: this.state.count + 1 });
  }
}

```

Key Points

- You **must call super(props)** before using this in the constructor.
- It's the **ideal place to set initial state** and bind methods.
- It only runs **once**, when the component is first created.

After the constructor, React proceeds to call `getDerivedStateFromProps()`, then `render()`, and finally `componentDidMount()`.

In React, the **render()** method is a **core part of the component lifecycle**, responsible for describing what the UI should look like. It's called during both the **mounting** and **updating** phases.

When render() is called:

1. **Mounting** – When the component is first added to the DOM.
2. **Updating** – When state or props change, triggering a re-render.

Key Characteristics:

- It must be **pure**: given the same props and state, it should return the same JSX.
- It **shouldn't modify state** or interact with the DOM directly.
- It returns **JSX**, which React then converts into actual DOM elements.

Example:

```
class Welcome extends React.Component {  
  render() {  
    return <h1>Hello, {this.props.name}!</h1>;  
  }  
}
```

Behind the scenes:

React uses the output of `render()` to build a **virtual DOM**, compares it with the previous one (diffing), and updates only the changed parts in the real DOM—making it super efficient.

In React class components, **`componentDidMount()`** is a **lifecycle method** that runs **once**, right after the component is inserted into the DOM. It's part of the **mounting phase**, and it's the perfect place for:

- **Fetching data from APIs**
- **Setting up subscriptions**
- **Interacting with the DOM**
- **Starting timers or animations**

Syntax

```
class MyComponent extends React.Component {  
  componentDidMount() {  
    console.log('Component mounted!');  
    // Example: fetch data or set up a subscription  
  }  
  render() {  
    return <div>Hello, world!</div>;  
  }  
}
```

Why it matters

- It ensures the component is fully rendered before running side effects.
- It only runs **once**, so it's great for one-time setup logic.

Functional Component Equivalent

In functional components, you mimic this behavior using `useEffect()` with an empty dependency array:

```
import { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    console.log('Component mounted!');
  }, []); // empty array = run once
  return <div>Hello, world!</div>;
}
```

In React class components, **`componentDidUpdate()`** is a **lifecycle method** that runs **after the component updates**—meaning after a re-render caused by changes in props or state. It's your go-to place for handling **side effects** that depend on the updated DOM or new data.

Syntax

```
componentDidUpdate(prevProps, prevState, snapshot) {
  // Compare previous and current props/state
  if (this.state.count !== prevState.count) {
    console.log('Count changed!');
  }
}
```

Use Cases

- **Fetching new data** when props change
- **Triggering animations** after layout updates
- **Logging or debugging** state transitions
- **Interacting with third-party libraries** that need the updated DOM

Important

- Always compare `prevProps` or `prevState` with current ones to avoid **infinite loops**.
- Don't call `setState()` unconditionally inside this method.

Functional Equivalent

In functional components, you'd use `useEffect()` with dependencies:

```
useEffect(() => {
  console.log('Component updated!');
}, []);
```

```
}, [count])); // runs when `count` changes
```

In React class components, **shouldComponentUpdate()** is a **lifecycle method** that lets you control whether a component should re-render when its props or state change. It's a powerful tool for **performance optimization**.

Purpose

By default, every update to props or state triggers a re-render. But sometimes, you know the changes don't affect the UI—so you can skip the render by returning false.

Syntax

```
shouldComponentUpdate(nextProps, nextState) {  
  // return true to allow re-render  
  // return false to skip it  
}
```

Example

```
class Greeting extends React.Component {  
  shouldComponentUpdate(nextProps) {  
    // Only re-render if the message has changed  
    return nextProps.message !== this.props.message;  
  }  
  render() {  
    console.log('Greeting component re-rendered');  
    return <h1>{this.props.message}</h1>;  
  }  
}
```

Important Notes

- It's only available in **class components**.
- Always compare nextProps and nextState with current ones.
- Returning false skips render() and componentDidUpdate().

Functional Component Equivalent

In functional components, you can achieve similar behavior using:

- React.memo() for shallow prop comparison
- useMemo() and useCallback() for memoizing values and functions

In React class components, **componentWillUnmount()** is a **lifecycle method** that runs **just before a component is removed from the DOM**. It's your cleanup crew—perfect for stopping timers, canceling network requests, or removing event listeners to prevent memory leaks.

Typical Use Cases

- Clear intervals or timeouts
- Unsubscribe from WebSocket or API listeners
- Remove event listeners
- Cancel in-flight fetch requests

Syntax

```
class MyComponent extends React.Component {  
  componentWillMount() {  
    console.log('Component is about to unmount!');  
    clearInterval(this.timer);  
    window.removeEventListener('resize', this.handleResize);  
  }  
  render() {  
    return <div>Goodbye, world!</div>;  
  }  
}
```

Functional Component Equivalent

In functional components, you use `useEffect()` with a cleanup function:

```
useEffect(() => {  
  const timer = setInterval(() => console.log('tick'), 1000);  
  return () => {  
    clearInterval(timer); // cleanup  
    console.log('Component unmounted!');  
  };  
}, []);
```