🧠 React `useEffect()` vs. Class Lifecycle Methods

📌 1. `useEffect(() => { ... }, [deps])` = Similar but not same as `componentDidUpdate()`

They **behave similarly**, but they are **not the same**! Let's break it down 🔍

⬛ `useEffect` (Functional Component)

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

const Counter = () => {
  const [count1, setCount1] = useState(0);
  const [count2, setCount2] = useState(0);

  // ☑ useEffect triggers only when count1 or count2 changes
  useEffect(() => {
    console.log("🔁 Either count1 or count2 changed!");
  }, [count1, count2]);

  return (
    <div>
      <h2>Count1: {count1}</h2>
      <h2>Count2: {count2}</h2>
      <button onClick={() => setCount1(count1 + 1)}>➕ Count1</button>
      <button onClick={() => setCount2(count2 + 1)}>➕ Count2</button>
    </div>
  );
};
```

⬛ `componentDidUpdate()` (Class Component)

```
import React from "react";

class CounterClass extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count1: 0,
      count2: 0,
    };
  }

  // ☑ Runs after every render — we must manually check what changed
  componentDidUpdate(prevProps, prevState) {
```

```
    if (
      prevState.count1 !== this.state.count1 ||
      prevState.count2 !== this.state.count2
    ) {
      console.log("🔁 Either count1 or count2 changed (class version)!");
    }
  }

  render() {
    const { count1, count2 } = this.state;
    return (
      <div>
        <h2>Count1: {count1}</h2>
        <h2>Count2: {count2}</h2>
        <button onClick={() => this.setState({ count1: count1 + 1 })}>➕
Count1</button>
        <button onClick={() => this.setState({ count2: count2 + 1 })}>➕
Count2</button>
      </div>
    );
  }
}
```

🔍 Behavior Comparison Chart

Feature	`useEffect` (Functional)	`componentDidUpdate` (Class)
Triggered after render?	☑ Yes	☑ Yes
Automatically checks changes?	☑ Yes (via `[deps]`)	❌ No (you check manually using `prevState`)
Initial render runs?	❌ No (if deps exist)	❌ No (not on first render)
Manual check needed?	❌ No	☑ Yes
Cleanup function supported?	☑ Yes (`return () => {}`)	☑ Use `componentWillUnmount()`
Simpler to read?	☑ Yes	❌ Slightly more verbose

❗ Important: Not the Same!

> `useEffect(() => {}, [count1, count2])` and `componentDidUpdate()` are similar **in purpose** but not **identical in behavior**.

For example:

◇ `useEffect()` **won't run on first render** if you provide dependencies. ◇ `componentDidUpdate()` also doesn't run on first render — but it requires **manual checks**.

```
useEffect(() => {
  // logic
}, [count1, count2]);
```

is replicated in **Class-based components**.

☑ Hook Behavior Recap:

```
useEffect(() => {
  // runs when either count1 or count2 changes
}, [count1, count2]);
```

This means: "*Run this effect whenever `count1` or `count2` changes.*"

🔁 Equivalent in Class Components:

In class components, **you use `componentDidUpdate(prevProps, prevState)`** to detect changes in specific state values:

☑ Example (Class-Based)

```
import React from 'react';

class CounterComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count1: 0,
      count2: 0,
    };
  }

  componentDidUpdate(prevProps, prevState) {
    // Check if count1 or count2 has changed
    if (
      prevState.count1 !== this.state.count1 ||
      prevState.count2 !== this.state.count2
    ) {
      console.log("🔄 count1 or count2 changed!");
      // Perform your logic here...
    }
  }

  render() {
    const { count1, count2 } = this.state;
```

```
    return (
      <div>
        <h2>Count1: {count1}</h2>
        <h2>Count2: {count2}</h2>
        <button onClick={() => this.setState({ count1: count1 + 1 })}>
          ➕ Increase Count1
        </button>
        <button onClick={() => this.setState({ count2: count2 + 1 })}>
          ➕ Increase Count2
        </button>
      </div>
    );
  }
}

export default CounterComponent;
```

🧠 Summary

- ☑ Use `useEffect(() => {...}, [deps])` to track specific state/prop changes in functional components
- 🧠 In class components, use `componentDidUpdate(prevProps, prevState)` and compare values manually
- ⚠ Don't assume they are 1:1 interchangeable — their **timing and cleanup behavior differ**

🔍 Summary Table

Hook Version	Class-Based Equivalent
`useEffect(() => {}, [])`	`componentDidMount()`
`useEffect(() => {}, [var1, var2])`	`componentDidUpdate(prevProps, prevState)` with condition
`useEffect(() => { return () => {} })`	`componentWillUnmount()`

🗄 `componentWillUnmount()` in React Class Components

☑ It is a lifecycle method that runs just **before a component is removed (unmounted)** from the DOM.

☑ Use Cases of `componentWillUnmount()` 🔥

Use Case	Why it's needed
🔄 **Clear timers** (`setInterval`, `setTimeout`)	Avoid memory leaks & unwanted behavior
🎧 **Remove event listeners**	Avoid duplicate listeners
📡 **Cancel API requests**	Prevent state updates after unmount
🎯 **Cleanup animations/subscriptions**	Prevent side effects from inactive components

---

🚫 Problem without `componentWillUnmount()`

When you don't clear `setInterval` → The callback continues to run **even after the component is gone**. This causes:

- ❌ Memory leaks
- ❌ Errors like "Can't update state on unmounted component"
- ❌ Extra API calls or rendering

☑️ Example: `setInterval` without cleanup (❌ Problem)

```
class TimerComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { seconds: 0 };
  }

  componentDidMount() {
    // 🔁 Start a timer
    this.interval = setInterval(() => {
      this.setState({ seconds: this.state.seconds + 1 });
    }, 1000);
  }

  render() {
    return <h2>⏱️ Timer: {this.state.seconds}s</h2>;
  }
}

// ⚠️ If this component is unmounted, interval continues! Memory leak 😱
```

☑️ Solution: Use `componentWillUnmount()` to clean it 🧹

```
class TimerComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { seconds: 0 };
```

```
  }

  componentDidMount() {
    this.interval = setInterval(() => {
      this.setState((prev) => ({ seconds: prev.seconds + 1 }));
    }, 1000);
  }

  // 🧹 Cleanup before unmount
  componentWillUnmount() {
    clearInterval(this.interval); // ✅ Stop the timer
    console.log("🧹 TimerComponent unmounted, interval cleared");
  }

  render() {
    return <h2>⏱️ Timer: {this.state.seconds}s</h2>;
  }
}
```

🔁 Switch Component Example to Show Cleanup in Action

```
class Parent extends React.Component {
  constructor() {
    super();
    this.state = { showTimer: true };
  }

  toggle = () => {
    this.setState({ showTimer: !this.state.showTimer });
  };

  render() {
    return (
      <div>
        <button onClick={this.toggle}>
          {this.state.showTimer ? "❌ Stop Timer" : "▶️ Start Timer"}
        </button>
        {this.state.showTimer && <TimerComponent />}
      </div>
    );
  }
}
```

👉 When you press the button, `TimerComponent` is removed. ✅ `componentWillUnmount()` fires → clears interval ✅ No memory leaks 💧

📦 Other Use Cases – Mini Examples

🎧 Remove Event Listener

```
componentDidMount() {
  window.addEventListener("resize", this.handleResize);
}

componentWillUnmount() {
  window.removeEventListener("resize", this.handleResize);
}
```

📡 Cancel API Call (with AbortController)

```
componentDidMount() {
  this.controller = new AbortController();

  fetch("https://api.example.com/data", { signal: this.controller.signal })
    .then(res => res.json())
    .then(data => this.setState({ data }));
}

componentWillUnmount() {
  this.controller.abort(); // ❌ Stop pending fetch
}
```

🔄 React Hook Equivalent (For Functional Components)

```
useEffect(() => {
  const interval = setInterval(() => {
    // logic
  }, 1000);

  return () => {
    clearInterval(interval); // 🧹 same cleanup
  };
}, []);
```

🧠 Summary

Aspect	`componentWillUnmount()`
Runs when?	Just before component is removed
Common Use	Clearing intervals, listeners, subscriptions

Aspect	`componentWillUnmount()`
Hook Equivalent	`useEffect(() => {...; return () => {...}}, [])`
Helps prevent	Memory leaks, console warnings, invalid updates

---

🧠 How to Clear `setTimeout` in `useEffect()` + Execution Flow Explained

☑️ Code Snippet You Asked:

```
useEffect(() => {
  console.log("💧 useEffect runs");

  return () => {
    console.log("🧹 Cleanup runs (if any)");
  };
}, []);

console.log("🖨️ Render runs");
```

📌 🔄 Output Order:

```
🖨️ Render runs
💧 useEffect runs
```

Because:

- `console.log("render")` happens on **each render**
- `useEffect()` runs **after** render (📦 commit phase)
- Cleanup function runs **before next effect OR on unmount**

💣 If you use `setTimeout` without clearing:

```
useEffect(() => {
  setTimeout(() => {
    console.log("⏰ Timeout executed!");
  }, 5000);
}, []);
```

💥 **Problem**: If component unmounts before 5s, the callback **still runs!** → This might lead to memory leaks or trying to update unmounted components.

☑️ Clear `setTimeout` like this:

```
useEffect(() => {
  console.log("💧 useEffect with timeout");

  const timeoutId = setTimeout(() => {
    console.log("⏰ Timeout fired!");
  }, 5000);

  // 🧹 Cleanup
  return () => {
    clearTimeout(timeoutId);
    console.log("🗑️ Timeout cleared!");
  };
}, []);
```

🧪 Real Example with Toggle

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

const TimeoutComponent = () => {
  const [visible, setVisible] = useState(true);

  return (
    <div>
      <button onClick={() => setVisible(!visible)}>
        {visible ? "❌ Hide" : "☑️ Show"} Component
      </button>

      {visible && <ChildWithTimeout />}
    </div>
  );
};

const ChildWithTimeout = () => {
  useEffect(() => {
    const timeout = setTimeout(() => {
      console.log("⏰ Timeout triggered!");
    }, 5000);

    return () => {
      clearTimeout(timeout);
      console.log("🧹 Timeout cleared!");
    };
  }, []);
```

```
  return <h3>👋 Hello! I will timeout in 5s unless unmounted.</h3>;
};

export default TimeoutComponent;
```

🧠 Summary

🔍 Concept	☑ React Way
Set a timeout	`const id = setTimeout(..., time)`
Clean it up	`return () => clearTimeout(id)`
Where?	Inside `useEffect()`
Cleanup timing	When component unmounts / effect re-runs
Prevents	Memory leaks, zombie callbacks

❓ Why Can't We Write `async` Directly in `useEffect()`?

☑ Short Answer:

You **can't make the `useEffect` callback itself `async`** because it is **expected to return either:**

- `undefined` (nothing), or
- a **cleanup function**

> But `async` functions **always return a Promise**, which breaks the rules of `useEffect`.

🧪 Let's See What Fails ❌

```
useEffect(async () => {
  const data = await fetch("https://api.example.com");
  console.log(data);
}, []);
```

🔴 Error:

> **Effect callbacks are synchronous to prevent race conditions**. You wrote an `async` function that returns a Promise instead of a cleanup function.

🚫 Why React Says NO to `async` `useEffect()`

- `useEffect()` expects:
  - a **sync function**
  - that optionally returns a **cleanup function**

🔁 But `async` always returns a **Promise**, like:

```
async function x() {
  return "hello";
}
// x() returns Promise<"hello">
```

So when you do:

```
useEffect(async () => {
  // ...
}, []);
```

You're giving React something like:

```
useEffect(() => Promise<...>) // ❌ Invalid
```

Which React does **not know how to handle**!

✅ The Correct Pattern: Define Async Inside

```
useEffect(() => {
  const fetchData = async () => {
    try {
      const res = await fetch("https://api.github.com/users/dpvasani");
      const data = await res.json();
      console.log(data);
    } catch (err) {
      console.error("❌ Fetch error:", err);
    }
  };

  fetchData(); // ✅ Call async inside sync function
}, []);
```

🧠 Analogy: "🚕 Uber Driver"

- `useEffect()` is like a driver who:
  - 🚗 Picks you up
  - 🧹 Cleans the seat afterward (cleanup function)
- But if you give them an **`async` trip that doesn't finish immediately** (a `Promise`)...
  - ❌ They don't know when to clean the seat
  - ❌ They can't handle unhandled promises

✨ Bonus: What if You NEED `await` in Cleanup?

If you're doing something async during cleanup, you must **wrap it safely**:

```
useEffect(() => {
  const fetchSomething = async () => { /* await here */ };
  fetchSomething();

  return () => {
    // 🚫 avoid: directly writing async
    (async () => {
      await doSomethingAsync(); // ☑️ safe pattern
    })();
  };
}, []);
```

🧠 TL;DR Summary

❓ Question	☑️ Answer
Can we use `async` directly?	❌ No
Why not?	It returns a Promise instead of cleanup
How to fix it?	Create & call an `async` function **inside** the effect
Can we use `await` in cleanup?	☑️ Yes, but use an IIFE: `(async () => {})()`

💡 React Class-Based Component – Multiple `state` Variables

> 🏫 Equivalent to multiple `useState()` in functional components, but done with style in class components!

🚀 Full Example: Class Component with Multiple States

```
import React from "react";

class MultiStateExample extends React.Component {
  constructor(props) {
    super(props);

    // 🧠 Initialize multiple state variables
    this.state = {
      count: 0,            // 🔢 Number state
      name: "Darshan",     // 👤 String state
      isLoggedIn: false,   // 🔐 Boolean state
    };
  }

  render() {
    return (
      <div>
        <h2>👤 Name: {this.state.name}</h2>
        <h3>🔢 Count: {this.state.count}</h3>
        <h4>🔐 Logged In: {this.state.isLoggedIn ? "✅ Yes" : "❌ No"}</h4>

        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          ➕ Increment Count
        </button>

        <button onClick={() => this.setState({ isLoggedIn: !this.state.isLoggedIn
})}>
          🔄 Toggle Login
        </button>
      </div>
    );
  }
}

export default MultiStateExample;
```

📌 Key Concepts Recap

🧠 Concept	✅ Code Example	💬 Explanation
Single State Object	`this.state = { count: 0, name: "..." }`	All states live inside **one object**
Update One Field Only	`this.setState({ count: newCount })`	React **auto-merges** this field into the state
No Need to Spread Manually	No need for `{ ...prevState }` like in `useState()`	Class state is smart, no spread required! 😎

🔁 Functional Component Equivalent

```
const [count, setCount] = useState(0);
const [name, setName] = useState("Darshan");
const [isLoggedIn, setIsLoggedIn] = useState(false);
```

🆚 In **function components**, each `useState()` call manages one value individually. In **class components**, everything lives inside `this.state` 🏠.

🎯 Quick Reminders

- ☑️ `setState()` only **updates the specific property**, no full overwrite needed.
- 🧩 Ideal for grouped values that logically belong together (like form data).
- 💧 Cleaner than using multiple `useState()` in simple scenarios.
