- Only called at the **top level** of **functional components**
- **Cannot be conditional**
- Allow functional components to be stateful

useState(initial value) -> cur_state, setStateFunc

Make sure to import it.

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

function Form() {
  const [name, setName] = useState("");  // State for name input
  const [email, setEmail] = useState(""); // State for email input

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log(`Name: ${name}, Email: ${email}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Enter your name"
      />
      <input
        type="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        placeholder="Enter your email"
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Example with Objects and Arrays:

```
const [user, setUser] = useState({ name: "", age: 0 });

const updateName = (newName) => {
  setUser(prevUser => ({  ...prevUser, name: newName }));
};

const [items, setItems] = useState([]);

const addItem = (item) => {
```

```
  setItems(prevItems => [...prevItems, item]);
};
```

useEffect(effectFunc, dependencyArr)

The `useEffect` hook is one of the most powerful and commonly used hooks in React. It allows you to perform **side effects** in functional components. Before hooks, this functionality was typically handled in class components using lifecycle methods like `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.

What Are Side Effects?

In React, a **side effect** refers to any operation that affects something outside the scope of the component, such as:

- Fetching data from an API
- Subscribing to an event
- Setting up a timer or interval
- Manually manipulating the DOM

Syntax:

```
useEffect(effectFunction, dependencyArray);
```

1. `effectFunction`: This is the function that contains the side effect logic. It may optionally return a cleanup function.
2. `dependencyArray`: This is an optional array of variables that the effect depends on. The effect re-runs only if one of these dependencies changes.

Key Points About `useEffect`

1. **Runs After Render:** The `useEffect` runs after the component renders (or re-renders). By default, it executes after every render, unless you specify dependencies.
2. **Cleanup Function:** If your side effect requires cleanup (e.g., unsubscribing from an event or clearing a timer), you can return a function from the `effectFunction`. React will call this cleanup function before the component unmounts or before the effect re-runs due to a dependency change.
3. **Controlled Execution:** The dependency array allows you to control when the effect runs.

How It Works

1. Run on Every Render

If no dependency array is provided, the effect runs after **every render**, including the initial render.

```
useEffect(() => {
  console.log("Effect runs after every render");
});
```

2. Run Only Once (on Mount)

If you pass an empty dependency array `[]`, the effect runs **only once** when the component mounts.

```
useEffect(() => {
  console.log("Effect runs once on mount");
}, []); // Empty array means no dependencies
```

3. Run When Dependencies Change

If you pass an array of dependencies, the effect runs **only when one of the dependencies changes**.

```
useEffect(() => {
  console.log("Effect runs when 'count' changes");
}, [count]); // Runs only when 'count' changes
```

4. Cleanup Function

To clean up after a side effect (e.g., remove an event listener, cancel a timer), return a function from the effect.

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

  return () => {
    clearInterval(interval); // Cleanup function to clear the interval
    console.log("Interval cleared");
  };
}, []); // Runs only on mount/unmount
```

Examples

Example 1: Fetching Data

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

function DataFetcher() {
  const [data, setData] = useState([]);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/posts")
      .then((response) => response.json())
      .then((json) => setData(json));

    // No cleanup required here
  }, []); // Fetch data only on initial render

  return (
    <ul>
      {data.map((item) => (
        <li key={item.id}>{item.title}</li>
      ))}
    </ul>
  );
}

export default DataFetcher;
```

Example 2: Event Listener with Cleanup

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

function MouseTracker() {
  const [position, setPosition] = useState({ x: 0, y: 0 });

  useEffect(() => {
    const handleMouseMove = (event) => {
      setPosition({ x: event.clientX, y: event.clientY });
    };

    window.addEventListener("mousemove", handleMouseMove);

    return () => {
      window.removeEventListener("mousemove", handleMouseMove); // Cleanup on unmoun
    };
  }, []); // Empty dependency array ensures the listener is added once

  return (
    <div>
      Mouse Position: {position.x}, {position.y}
    </div>
  );
}
```

```
export default MouseTracker;
```

Example 3: Re-run Effect Based on Dependency

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

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

  useEffect(() => {
    console.log(`Count is now ${count}`);
  }, [count]); // Effect runs every time 'count' changes

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default Counter;
```

When to Use Cleanup Functions

1. **Event Listeners:** If you add event listeners (e.g., `window.addEventListener`), you need to remove them when the component unmounts or when the dependencies change.
2. **Timers:** Always clear intervals or timeouts when they're no longer needed.
3. **Subscriptions:** If you subscribe to data streams or other external resources, unsubscribe during cleanup.

Rules of `useEffect`

1. **Call `useEffect` at the Top Level:**
   - Don't call it conditionally or inside loops. Always call it at the top level of your component.

```
if (condition) {
  useEffect(...); // ❌ Don't do this
}
```

Instead:

```
useEffect(() => {
  if (condition) {
    // Effect logic here
  }
});
```

2. **Specify Dependencies Carefully:**

   - The dependency array determines when the effect re-runs. Missing dependencies can lead to stale values or bugs.

```
useEffect(() => {
  console.log(value); // ❌ Might use stale value
}, []); // 'value' should be in the dependencies
```

Correct usage:

```
useEffect(() => {
  console.log(value);
}, [value]); // ✅ Dependency array includes 'value'
```

Comparison with Class Components

Lifecycle Method in Class Components	Equivalent in `useEffect`
`componentDidMount`	`useEffect(() => {...}, [])`
`componentDidUpdate`	`useEffect(() => {...}, [dependencies])`
`componentWillUnmount`	Cleanup function in `useEffect`

Common Use Cases

1. **Data Fetching**
2. **Subscribing to Events**
3. **Timers and Intervals**
4. **Updating the Document Title**

```
useEffect(() => {
  document.title = `Count: ${count}`;
```

```
}, [count]);
```

5. **Animations and Transitions**

Summary

- `useEffect` is a powerful tool for handling side effects in functional components.
- It combines the functionalities of multiple class lifecycle methods (`componentDidMount`, `componentDidUpdate`, `componentWillUnmount`).
- The dependency array is key to controlling when and how often your effect runs.
- Always ensure you clean up effects like event listeners, timers, or subscriptions to avoid memory leaks.

This hook is essential for building dynamic and interactive React applications!