

# useCallback useMemo React Memo

what is useCallback and useMemo and memo in react ts with example

In React (TypeScript or JavaScript), `useCallback`, `useMemo`, and `memo` are optimization hooks and higher-order components that help avoid unnecessary re-renders and improve performance. Here's a quick overview of each:

## 1. `useCallback`

`useCallback` is a hook that returns a memoized version of a callback function. It's useful when you want to avoid recreating a function on every render, especially if the function is passed as a prop to child components.

**Example:**

tsx

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

type ButtonProps = {
  handleClick: () => void;
};

const Button: React.FC<ButtonProps> = React.memo(({ handleClick }) => {
  console.log('Button rendered');
  return <button onClick={handleClick}>Click me</button>;
});

const App: React.FC = () => {
  const [count, setCount] = useState(0);

  // Without `useCallback`, this function would be recreated on every render.
  const increment = useCallback(() => {
    setCount((prev) => prev + 1);
  }, []);

  return (
    <div>
      <p>Count: {count}</p>
      <Button handleClick={increment} />
    </div>
  );
};
```

```

    </div>
  );
};

export default App;

```

In this example:

- The `increment` function is memoized with `useCallback` so that it doesn't get recreated every time `App` renders.
- `Button` only re-renders if `increment` changes, reducing unnecessary renders.

## 2. `useMemo`

`useMemo` is a hook that memoizes the result of an expensive computation, preventing the computation from re-running on every render unless its dependencies change.

**Example:**

```

tsx

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

const App: React.FC = () => {
  const [count, setCount] = useState(0);
  const [text, setText] = useState('');

  // Expensive calculation that only recalculates when `count` changes
  const expensiveCalculation = useMemo(() => {
    console.log('Calculating...');
    return count * 1000;
  }, [count]);

  return (
    <div>
      <p>Count: {count}</p>
      <p>Expensive Calculation Result: {expensiveCalculation}</p>
      <button onClick={() => setCount((prev) => prev + 1)}>Increment</button>
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}

```

```

        placeholder="Type something..."
      />
    </div>
  );
};

export default App;

```

In this example:

- `expensiveCalculation` only recalculates if `count` changes. This is useful if the calculation is resource-intensive and doesn't need to be re-run on every render.

### 3. `memo`

`memo` is a higher-order component that memoizes a component itself. If the component's props haven't changed, `memo` will prevent it from re-rendering.

**Example:**

```

tsx

import React, { useState } from 'react';

type MessageProps = {
  text: string;
};

const Message: React.FC<MessageProps> = React.memo(({ text }) => {
  console.log('Message rendered');
  return <p>{text}</p>;
});

const App: React.FC = () => {
  const [count, setCount] = useState(0);
  const [message, setMessage] = useState('Hello');

  return (
    <div>
      <Message text={message} />
      <p>Count: {count}</p>
      <button onClick={() => setCount((prev) => prev + 1)}>Increment Count</button>
      <button onClick={() => setMessage('Hi')}>Change Message</button>
    </div>
  );
};

```

```

    </div>
  );
};

export default App;

```

In this example:

- The `Message` component is wrapped in `React.memo`, so it only re-renders when its `text` prop changes.
- Clicking the "Increment Count" button does not trigger a re-render for `Message` since `text` hasn't changed.

what is ref in react and how to use it

In React, you may be referring to `ref` rather than "ref." The `ref` (short for "reference") is used to access or modify DOM elements or React components directly. This is particularly useful for focusing on input fields, triggering animations, or integrating with third-party libraries that require direct DOM manipulation.

## Basic Usage of `ref` in React

React provides the `useRef` hook for functional components and `React.createRef` for class components. The `ref` object returned by `useRef` persists for the component's entire lifecycle, allowing you to access or modify the `current` property without causing a re-render.

### Example 1: Accessing a DOM Element with `useRef`

Here's an example of how to use `useRef` to focus an input field.

```

tsx

import React, { useRef } from 'react';

const App: React.FC = () => {
  const inputRef = useRef<HTMLInputElement>(null);

  const handleFocus = () => {
    // Access the DOM node and focus the input
    if (inputRef.current) {
      inputRef.current.focus();
    }
  }
}

```

```

};

return (
  <div>
    <input ref={inputRef} type="text" placeholder="Type here" />
    <button onClick={handleFocus}>Focus Input</button>
  </div>
);
};

export default App;

```

In this example:

- `useRef<HTMLInputElement>(null)` creates a reference to an `HTMLInputElement`.
- By assigning `inputRef` to the `ref` attribute of the `<input>`, we can access this element directly.
- When the button is clicked, `handleFocus` checks if `inputRef.current` is not `null` and then calls `focus()` on it.

## Example 2: Using `useRef` to Keep Track of Previous State

You can also use `useRef` to store a value that persists across renders without triggering a re-render. This is useful for tracking the previous state of a variable.

```

tsx

import React, { useRef, useState, useEffect } from 'react';

const App: React.FC = () => {
  const [count, setCount] = useState(0);
  const prevCountRef = useRef<number>(count);

  useEffect(() => {
    prevCountRef.current = count;
  }, [count]);

  return (
    <div>
      <p>Current Count: {count}</p>
      <p>Previous Count: {prevCountRef.current}</p>
    </div>
  );
};

```

```

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

export default App;

```

In this example:

- `prevCountRef` is updated to hold the previous value of `count` whenever `count` changes.
- The previous count value persists without causing re-renders, making `useRef` an efficient option for this purpose.

### Example 3: Using `ref` with Class Components

If you are working with class components, you use `React.createRef` to achieve similar behavior.

```

tsx

import React, { Component } from 'react';

class App extends Component {
  inputRef = React.createRef<HTMLInputElement>();

  handleFocus = () => {
    if (this.inputRef.current) {
      this.inputRef.current.focus();
    }
  };

  render() {
    return (
      <div>
        <input ref={this.inputRef} type="text" placeholder="Type here" />
        <button onClick={this.handleFocus}>Focus Input</button>
      </div>
    );
  }
}

```

```
export default App;
```

## Summary

- `useRef` in functional components and `React.createRef` in class components provide a way to access DOM elements or store mutable data without triggering re-renders.
- **Common use cases** include focusing input fields, triggering animations, storing values across renders, and integrating with non-React libraries.

Using `useRef` to Keep Track of Previous State give me exaple

Using `useRef` to keep track of previous state is a helpful way to store a previous value across renders without causing re-renders. Here's an example in TypeScript where we use `useRef` to track the previous count value in a counter component.

## Example: Track Previous State with `useRef`

tsx

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

const Counter: React.FC = () => {
  const [count, setCount] = useState(0);

  // useRef to store the previous count value
  const prevCountRef = useRef<number | null>(null);

  // Update the ref's current value to the current count after each render
  useEffect(() => {
    prevCountRef.current = count;
  }, [count]);

  return (
    <div>
      <h1>Current Count: {count}</h1>
      <h2>Previous Count: {prevCountRef.current ?? "N/A"}</h2>
      <button onClick={() => setCount((prev) => prev + 1)}>Increment</button>
      <button onClick={() => setCount((prev) => prev - 1)}>Decrement</button>
    </div>
  );
};
```

```
};  
  
export default Counter;
```

## Explanation:

1. **State ( `count` ):** The `count` state is used to keep track of the current count value.
2. **Ref ( `prevCountRef` ):** `prevCountRef` is created with `useRef` and initialized to `null`. This `ref` will store the previous count value across renders.
3. **Updating `prevCountRef` in `useEffect` :** The `useEffect` hook updates `prevCountRef.current` to the current count value after each render. This means that on the next render, `prevCountRef.current` will hold the value of `count` from the previous render.
4. **Displaying Previous and Current Count:** We display both the current and previous count values in the component.

When you click the "Increment" or "Decrement" button:

- `count` state is updated, causing the component to re-render.
- `prevCountRef.current` keeps the previous `count` value without triggering a re-render, allowing it to display the "Previous Count" value accurately.

how to make a breakpoint in the last example before the render the new count

To set a breakpoint before rendering the new `count` in the example, you can add a `debugger` statement in the `useEffect` or `setCount` function. This allows you to pause execution and inspect the state and ref values right before the component re-renders with the new `count`.

Here's how you can set a breakpoint right before the new `count` value is rendered:

### 1. Using `debugger` in `useEffect`

You can add a `debugger` statement inside the `useEffect` hook to pause execution each time the `count` changes, right before it updates the `prevCountRef` with the current value:

```
tsx
```



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

const Counter: React.FC = () => {
  const [count, setCount] = useState(0);
  const prevCountRef = useRef<number | null>(null);

  useEffect(() => {
    debugger; // Set breakpoint here to check the old count before updating
    prevCountRef
    prevCountRef.current = count;
  }, [count]);

  return (
    <div>
      <h1>Current Count: {count}</h1>
      <h2>Previous Count: {prevCountRef.current ?? "N/A"}</h2>
      <button onClick={() => setCount((prev) => prev + 1)}>Increment</button>
      <button onClick={() => setCount((prev) => prev - 1)}>Decrement</button>
    </div>
  );
};

export default Counter;
```

## Explanation:

- The `debugger` statement in the `useEffect` will trigger the breakpoint right before `prevCountRef.current` updates to the current `count`.
- When `count` changes, `useEffect` runs, hitting the breakpoint before the re-render with the updated count value. This lets you inspect `count` (current count) and `prevCountRef.current` (previous count) before the re-render.

## 2. Adding `debugger` in `setCount`

If you want to break before `count` updates, you could add the `debugger` inside the `setCount` callback:

```
tsx

<button
  onClick={() => {
```

```
    debugger; // Set breakpoint here to inspect before count updates
    setCount((prev) => prev + 1);
  }}
>
  Increment
</button>
```

In this case:

- The `debugger` pauses execution before `count` changes, allowing you to inspect the old `count` and `prevCountRef.current` before the state update triggers a re-render.