How Do the useCallback Hook and React.memo Work?

freecodecamp.org/learn/front-end-development-libraries-v9/lecture-working-with-data-fetching-and-memoization-in-react/how-do-the-usecallback-hook-and-react-memo-work

In the last lesson, you learned about memoization and how the `useMemo` hook works.

In this lesson, you'll learn how the `useCallback` hook and `React.memo` work.

In the last lesson, we also mentioned that `useCallback` is for memoizing function references.

For `React.memo`, it lets you memoize a component to prevent it from unnecessary re-renders when its prop has not changed.

Here's the basic syntax of the `useCallback` hook:

```
const handleClick = useCallback(() => {
  // code goes here
}, [dependency]);
```

And here's the basic syntax of `React.memo`:

```
const MemoizedComponent = React.memo(({ prop }) => {
  return (
    <>
      {/* Presentation */}
    </>
  )
});
```

Let's look at an example of the `useCallback` hook:

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

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

  const handleClick = () => {
    setCount((prevCount) => prevCount + 1);
  };

  useEffect(() => {
    console.log("useEffect runs");
  }, [handleClick]);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
}

export default Counter;
```

In the component, the effect runs any time `handleClick` changes because the `handleClick` function is being recreated on every render.

To fix this, you need to tell React to treat the `handleClick` function as the same thing across renders by memoizing it with the `useCallback` hook, so it doesn't get recreated:

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

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

  // Memoize the handleClick function with useCallback
  const handleClick = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, []);

  useEffect(() => {
    console.log("useEffect runs");
  }, [handleClick]);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
}

export default Counter;
```

Now the `handleClick` function is not being recreated on every render.

To show you how the `React.memo` (or `memo`) higher-order function works and the `useCallback` hook work in tandem, here's a `Counter` component with a `handleClick` function that needs `useCallback` but is currently not using it:

```
import { useState, useEffect, useCallback } from "react";
import CounterChild from "./CounterChild";

function Counter() {
  const [count, setCount] = useState(0);
  const [timer, setTimer] = useState(new Date().toLocaleTimeString());

  const handleClick = () => {
    setCount(count + 1);
  };

  useEffect(() => {
    const interval = setInterval(() => {
      setTimer(new Date().toLocaleTimeString());
    }, 1000);

    return () => clearInterval(interval);
  }, []);

  return (
    <div>
      <h1>Time: {timer}</h1>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
      <CounterChild onClick={handleClick} />
    </div>
  );
}

export default Counter;
```

This function also has a timer in state that updates every second. This makes the component re-render every time the `timer` changes, making the `handleClick` function get recreated on every render.

That's why the `handleClick` needs to be memoized with `useCallback`.

Here's the `CounterChild` component:

```
const CounterChild = ({ onClick }) => {
  console.log("CounterChild component rendered");
  return <button onClick={onClick}>Increment from Child</button>;
};

export default CounterChild;
```

This `CounterChild` component takes an `onClick` prop, giving you the ability to also increment the counter from it.

Since the `CounterChild` component is a child of the `Counter` component, it will also render any time the `Counter` re-renders due to the changing timer. So, the `CounterChild` also needs to be memoized.

Without memoization, because as the component re-renders due to the timer updating every second, the `CounterChild` component is also re-rendered.

To prevent this, you need to memoize the `CounterChild` component with `React.memo`:

```
import React from "react";

const CounterChild = React.memo(({ onClick }) => {
  console.log("CounterChild component rendered");
  return <button onClick={onClick}>Increment from Child</button>;
});

export default CounterChild;
```

Things do not work optimally yet even after memoizing the `CounterChild` with `React.memo`.

This happens because the `handleClick` function is being recreated on every render, so it also needs to be memoized with `useCallback`, in order to tell React that you need the function to stay the same across renders:

```
const handleClick = useCallback(() => {
  setCount((prevCount) => prevCount + 1);
}, [count]);
```

Now, the component only re-renders when the `count` state changes.

Questions

What does the `useCallback` hook memoize?

Computed values.

Function references.

Component renders.

State updates.

Which of these is the correct syntax for the `useCallback` hook?

```
const handleClick = useCallback(() => { /* code
*/ });
```

```
const handleClick = useCallback(() => { /* code */ },
dependency);
```

```
const handleClick = useCallback(() => { /* code */ },
[dependency]);
```

```
const handleClick = useCallback(() => { dependency },
{});
```

What does `React.memo` memoize?

Function references.

Component state.

Component renders based on unchanged props.

DOM elements.