4
Using React Hooks

In this chapter, we will learn about React's common Hooks and how to use them with TypeScript. We will implement the knowledge of all these Hooks in a React component that allows a user to adjust a score for a person. We will start by exploring the effect Hook and begin to understand use cases where it is useful. We will then delve into two state Hooks, `useState` and `useReducer`, understanding when it is best to use each one. After that, we will cover the ref Hook and how it differs from the state Hook, and then the memo and callback Hooks, looking at how they can help performance.

So, we'll cover the following topics:

- Using the effect Hook
- Using state Hooks
- Using the ref Hook
- Using the memo Hook
- Using the callback Hook

Technical requirements

We will use the following technologies in this chapter:

- **Browser**: A modern browser such as Google Chrome
- **Node.js** and **npm**: You can install them from `https://nodejs.org/en/download/`
- **Visual Studio Code**: You can install it from `https://code.visualstudio.com/`

All the code snippets in this chapter can be found online at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter4`.

Using the effect Hook

In this section, we will learn about the effect Hook and where it is useful. We will then create a new React project and a component that makes use of the effect Hook.

Understanding the effect Hook parameters

The effect Hook is used for component side effects. A component side effect is something executed outside the scope of the component such as a web service request. The effect Hook is defined using the `useEffect` function from React. `useEffect` contains two parameters:

- A function that executes the effect; at a minimum, this function runs each time the component is rendered
- An optional array of dependencies that cause the effect function to rerun when changed

Here's an example of the `useEffect` Hook in a component:

```
function SomeComponent() {
  function someEffect() {
    console.log("Some effect");
  }
  useEffect(someEffect);
  return ...
}
```

The preceding effect Hook is passed an effect function called `someEffect`. No effect dependencies have been passed, so the effect function is executed each time the component renders.

Often, an anonymous arrow function is used for the effect function. Here's the same example but with an anonymous effect function instead:

```
function SomeComponent() {
  useEffect(() => {
    console.log("Some effect");
  });
  return ...
}
```

As you can see, this version of the code is a little shorter and arguably easier to read.

Here's another example of an effect:

```
function SomeOtherComponent({ search }) {
  useEffect(() => {
    console.log("An effect dependent on a search prop",
      search);
  }, [search]);
  Return ...;
}
```

This time the effect has a dependency on a `search` prop. So, the `search` prop is defined in an array in the effect Hook's second parameter. The effect function will run every time the value of `search` changes.

The rules of Hooks

There are some rules that all Hooks, including `useEffect`, must obey:

- A Hook can only be called at the top level of a function component. So, a Hook can't be called in a loop or in a nested function such as an event handler.
- A Hook can't be called conditionally.
- A Hook can only be used in function components and not class components.

The following example is a violation of the rules:

```
export function AnotherComponent() {
  function handleClick() {
    useEffect(() => {
      console.log("Some effect");
    });
  }
  return <button onClick={handleClick}>Cause effect</button>;
}
```

This is a violation because `useEffect` is called in a handler function rather than at the top level. A corrected version is as follows:

```
export function AnotherComponent() {
  const [clicked, setClicked] = useState(false);
  useEffect(() => {
```

```
    if (clicked) {
      console.log("Some effect");
    }
  }, [clicked]);
  function handleClick() {
    setClicked(true);
  }
  return <button onClick={handleClick}>Cause effect</button>;
}
```

`useEffect` has been lifted to the top level and now depends on the `clicked` state that is set in the handler function.

The following is another example that violates the rules of Hooks:

```
function YetAnotherComponent({ someProp }) {
  if (!someProp) {
    return null;
  }
  useEffect(() => {
    console.log("Some effect");
  });
  return ...
}
```

The violation is because `useEffect` is called conditionally. If `someProp` is falsy, `null` is returned from the component and `useEffect` is never called. So, the condition is that `useEffect` is only called when `someProp` is truthy.

A corrected version is as follows:

```
function YetAnotherComponent({someProp}) {
  useEffect(() => {
    if (someProp) {
      console.log("Some effect");
    }
  });
  if (!someProp) {
    return null
```

```
  }
  return ...
}
```

`useEffect` has been lifted above the condition. The condition has also been put inside the effect function so that its logic is only executed when `someProp` is truthy.

Effect cleanup

An effect can return a function that performs cleanup logic when the component is unmounted. Cleanup logic ensures nothing is left that could cause a memory leak. Let's consider the following example:

```
function ExampleComponent({onClickAnywhere}) {
  useEffect(() => {
    function handleClick() {
      onClickAnywhere();
    }
    document.addEventListener("click", handleClick);
  });
  return ...
}
```

The preceding effect function attaches an event handler to the `document` element. The event handler is never detatched though, so multiple event handlers will become attached to the `document` element as the effect is rerun. This problem is resolved by returning a `cleanup` function that detaches the event handler as follows:

```
function ExampleComponent({ onClickAnywhere }) {
  useEffect(() => {
    function handleClick() {
      onClickAnywhere();
    }
    document.addEventListener("click", listener);
    return function cleanup() {
      document.removeEventListener("click", listener);
    };
  });
  return ...;
}
```

Often, an anonymous arrow function is used for the cleanup function:

```
function ExampleComponent({ onClickAnywhere }) {
  useEffect(() => {
    function handleClick() {
      onClickAnywhere();
    }
    document.addEventListener("click", listener);
    return () => {
      document.removeEventListener("click", listener);
    };
  });
  return ...;
```

An anonymous arrow function is a little shorter than the named function in the previous example.

Next, we will explore a common use case for the effect Hook.

Creating the project

Let's start by creating a new project in Visual Studio Code using Create React App. We learned how to do this in *Chapter 3, Setting Up React and TypeScript* – the steps are as follows:

1. Open Visual Studio Code in a blank folder of your choice and run the following command:

   ```
   npx create-react-app app --template typescript
   ```

   Create React App will take a minute or so to create the project. The app is called `app` in the proceeding command, but feel free to change this.

2. Reopen Visual Studio Code in the `app` folder that has just been created (or whatever you called the app).

3. Install Prettier and its libraries to allow it to work with ESLint. Run the following command in the terminal to do this:

   ```
   npm i -D prettier eslint-config-prettier eslint-plugin-
   prettier
   ```

4. Enable Visual Studio Code to automatically format code as files are saved. To do this, create a `.vscode` folder in the project root and create a `settings.json` file containing the following:

   ```
   {
     "editor.formatOnSave": true,
   ```

```
    "editor.defaultFormatter": "esbenp.prettier-vscode"
  }
```

5. Update the ESLint configuration to allow Prettier to manage the styling rules. To do this, add the following highlighted line to the `eslintConfig` section in `package.json`:

```
  {
    ...,
    "eslintConfig": {
      "extends": [
        "react-app",
        "react-app/jest",
        "plugin:prettier/recommended"
      ]
    },
    ...
  }
```

6. Add the following Prettier configuration in a file called `.prettierrc.json`:

```
  {
    "printWidth": 100,
    "singleQuote": true,
    "semi": true,
    "tabWidth": 2,
    "trailingComma": "all",
    "endOfLine": "auto"
  }
```

7. Remove the following files from the `src` folder, because these aren't needed in this project:

   - `App.test.tsx`
   - `Logo.svg`

8. Open `index.tsx` and save the file without making any changes. This will remove any formatting issues.

9. Open `App.tsx` and replace the content with the following:

```
  import React from 'react';
  import './App.css';
```

```
function App() {
  return <div className="App"></div>;
}
export default App;
```

10. Start the app running in development mode by running `npm start` in the terminal. The app contains a blank page at the moment. Keep the app running as we explore the different Hooks in a React component.

That's the project created. Next, we will use the effect Hook.

Fetching data using the effect Hook

A common use of the effect Hook is fetching data. Carry out the following steps to implement an effect that fetches a person's name:

1. Create a function that will simulate a data request. To do this, create a file called `getPerson.ts` in the `src` folder and then add the following content to this file:

```
type Person = {
  name: string,
};
export function getPerson(): Promise<Person> {
  return new Promise((resolve) =>
    setTimeout(() => resolve({ name: "Bob" }), 1000)
  );
}
```

The function asynchronously returns an object, `{ name: "Bob" }`, after a second has elapsed.

Notice the type annotation for the return type, `Promise<Person>`. The `Promise` type represents a JavaScript `Promise`, which is something that will eventually be completed. The `Promise` type has a generic argument for the item type that is resolved in the promise, which is `Person` in this example. For more information on JavaScript promises, see the following link: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`.

2. Next, we will create a React component that will eventually display a person and a score. Create a file called `PersonScore.tsx` in the `src` folder and then add the following contents to the file:

```
import { useEffect } from 'react';
import { getPerson } from './getPerson';
```

```
export function PersonScore() {
  return null;
}
```

The `useEffect` Hook has been imported from React and the `getPerson` function we have just created has also been imported. At the moment, the component simply returns `null`.

3. Add the following effect above the return statement:

```
export function PersonScore() {
  useEffect(() => {
    getPerson().then((person) => console.log(person));
  }, []);
  return null;
}
```

The effect calls the `getPerson` function and outputs the returned person to the console. The effect is only executed after the component is initially rendered because an empty array has been specified as the effect dependencies in its second argument.

4. Open `App.tsx` and render the `PersonScore` component inside the `div` element:

```
import React from 'react';
import './App.css';
import { PersonScore } from './PersonScore';

function App() {
  return (
    <div className="App">
      <PersonScore />
    </div>
  );
}
export default App;
```

5. Go to the running app in the browser and go to the **Console** panel in the browser's DevTools. Notice that the `person` object appears in the console, which verifies that the effect that fetches the `person` data ran properly:

Figure 4.1 – The effect output

You may also notice that the effect function has been executed twice rather than once. This behavior is intentional and only happens in development mode with React Strict Mode. This will eventually allow a future React feature to preserve the state when sections of the UI are removed. See this blog post from the React team for more information on this behavior: `https://reactjs.org/blog/2022/03/29/react-v18.html#new-strict-mode-behaviors`.

6. Next, we will refactor how the effect function is called to expose an interesting problem. Open `PersonScore.tsx` and change the `useEffect` call to use the `async/await` syntax:

```
useEffect(async () => {
  const person = await getPerson();
  console.log(person);
}, []);
```

> **Note**
>
> The `async/await` syntax is an alternative way to write asynchronous code. Many developers prefer it because it reads like synchronous code. For more information on `async/await`, see the following link: `https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Promises#async_and_await`.

The preceding code is arguably more readable, but React raises an error. Look in the browser's console and you'll see the following error:

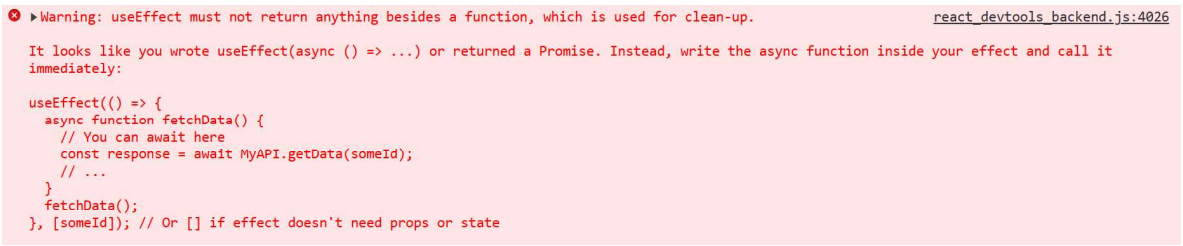

Figure 4.2 – Effect async error

The error is very informative – the `useEffect` Hook doesn't allow a function marked with `async` to be passed into it.

7. Next, update the code and use the approach suggested in the error message:

```
useEffect(() => {
  async function getThePerson() {
    const person = await getPerson();
    console.log(person);
  }
  getThePerson();
}, []);
```

A nested asynchronous function has been defined and immediately called in the effect function; this works nicely.

8. This implementation of the effect is arguably less readable than the initial version. So, switch back to that version before continuing to the next section. The code is available to copy from the following link: `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter4/Section1-Using-the-effect-hook/src/PersonScore.tsx`.

That completes our exploration of the effect Hook – here's a recap:

- The effect Hook is used to execute component side effects when a component is rendered or when certain props or states change.
- A common use case for the effect Hook is fetching data. Another use case is where DOM events need to be manually registered.
- Any required effect cleanup can be done in a function returned by the effect function.

Next, we will learn about the two state Hooks in React. Keep the app running as we move to the next section.

Using state Hooks

We have already learned about the `useState` Hook in previous chapters, but here we will look at it again and compare it against another state Hook we haven't covered yet, `useReducer`. We will expand the `PersonScore` component we created in the last section to explore these state Hooks.

Using useState

As a reminder, the `useState` Hook allows state to be defined in a variable. The syntax for `useState` is as follows:

```
const [state, setState] = useState(initialState);
```

We will enhance the `PersonScore` component we created in the last section to store the person's name in `state`. We will also have `state` for a score that is incremented, decremented, and reset using some buttons in the component. We will also add the `loading` state to the component, which will show a loading indicator when `true`.

Carry out the following steps:

1. Open `PersonScore.tsx` and add `useState` to the React import statement:

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

2. Add the following state definitions for `name`, `score`, and `loading` at the top of the component function, above the `useEffect` call:

```
export function PersonScore() {
  const [name, setName] = useState<string | undefined>();
  const [score, setScore] = useState(0);
  const [loading, setLoading] = useState(true);

  useEffect( ... );

  return null;
}
```

The `score` state is initialized to `0` and `loading` is initialized to `true`.

3. Change the effect function to set the `loading` and `name` state values after the person data has been fetched. This should replace the existing `console.log` statement:

```
useEffect(() => {
  getPerson().then((person) => {
    setLoading(false);
    setName(person.name);
  });
}, []);
```

After the person has been fetched, `loading` is set to `false`, and `name` is set to the person's name.

4. Next, add the following `if` statement in between the `useEffect` call and the return statement:

```
useEffect( ... );
if (loading) {
  return <div>Loading ...</div>;
}
return ...
```

This displays a loading indicator when the `loading` state is `true`.

5. Change the component's return statement from outputting nothing to outputting the following:

```
if (loading) {
  return <div>Loading ...</div>;
}
return (
  <div>
    <h3>
      {name}, {score}
    </h3>
    <button>Add</button>
    <button>Subtract</button>
    <button>Reset</button>
  </div>
);
```

The person's name and score are displayed in a header with **Add**, **Subtract**, and **Reset** buttons underneath (don't worry that the output is unstyled – we will learn how to style components in the next chapter):

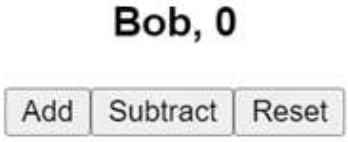

Figure 4.3 – The PersonScore component after data has been fetched

6. Update the **Add** button so that it increments the score when clicked:

```
<button onClick={() => setScore(score + 1)}>Add</button>
```

The button click event calls the score state setter to increment the state.

There is an alternative method of updating the state values based on their previous value. The alternative method uses a parameter in the state setter that gives the previous state value, so our example could look as follows:

```
setScore(previousScore => previousScore + 1)
```

This is arguably a little harder to read, so we'll stick to our initial method.

7. Add score state setters to the other buttons as follows:

```
<button onClick={() => setScore(score - 1)}>Subtract</
button>
<button onClick={() => setScore(0)}>Reset</button>
```

8. In the running app, click the different buttons. They should change the score as you would expect.

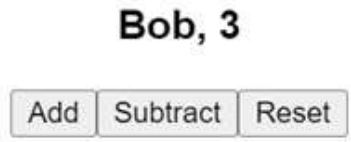

Figure 4.4 – The PersonScore component after the button is clicked

9. Before we finish this exercise, let's take some time to understand when the state values are actually set. Update the effect function to output the state values after they are set:

```
useEffect(() => {
  getPerson().then((person) => {
    setLoading(false);
    setName(person.name);
    console.log("State values", loading, name);
  });
}, []);
```

Perhaps we would expect `false` and `"Bob"` as the output to the console? However, `true` and `undefined` are the output to the console. This is because updating state values is not immediate – instead, they are batched and updated before the next render. So, it isn't until the next render that `loading` will be `false`, and `name` will be `"Bob"`.

We no longer need the `console.log` statement we added in this step, so remove it before continuing.

Next, we will learn about an alternative React Hook for using state.

Understanding useReducer

`useReducer` is an alternative method of managing state. It uses a **reducer** function for state changes, which takes in the current state value and returns the new state value.

Here is an example of a `useReducer` call:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

So, `useReducer` takes in a reducer function and the initial state value as parameters. It then returns a tuple containing the current state value and a function to **dispatch** state changes.

The dispatch function takes in an argument that describes the change. This object is called an **action**. An example `dispatch` call is as follows:

```
dispatch({ type: 'add', amount: 2 });
```

There is no defined structure for an action, but it is common practice for it to contain a property, such as `type`, to specify the type of change. Other properties in the action can vary depending on the type of change. Here's another example of a `dispatch` call:

```
dispatch({ type: 'loaded' });
```

This time, the action only needs the type to change the necessary state.

Turning our attention to the reducer function, it has parameters for the current state value and the action. Here's an example code snippet of a reducer:

```
function reducer(state: State, action: Action): State {
  switch (action.type) {
    case 'add':
      return { ...state, total: state.total + action.amount };
    case ...
      ...
    default:
      return state;
  }
}
```

The reducer function usually contains a `switch` statement based on the action type. Each switch branch makes the required changes to the state and returns the updated state. A new state object is created during the state change – the current state is never mutated. A mutating state would result in the component not re-rendering.

> **Note**
>
> In the preceding code snippet, inside the `"add"` branch the **spread syntax** is used on the `state` variable (`...state`). The spread syntax copies all the properties from the object after the three dots. In the preceding code snippet, all the properties are copied from the `state` variable into the new state object returned. The `total` property value will then be overwritten by `state.total + action.amount` because this is defined after the spread operation in the new object creation. For more information on the spread syntax, see the following link: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax`.

The types for `useReducer` can be explicitly defined in its generic parameter as follows:

```
const [state, dispatch] = useReducer<Reducer<State, Action>>(
  reducer,
  initialState
);
```

`Reducer` is a standard React type that has generic parameters for the type of state and the type of action.

So, `useReducer` is more complex than `useState` because state changes go through a reducer function that we must implement. This benefits complex state objects with related properties or when a state change depends on the previous state value.

Next, we will implement state using `useReducer`.

Using useReducer

We will refactor the `PersonScore` component we have been working on to use `useReducer` instead of `useState`. To do this, carry out the following steps. The code snippets used are available to copy from `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter4/Section2-Using-state-hooks/2-Using-useReducer/src/PersonScore.tsx`:

1. Open `PersonScore.tsx` and import `useReducer` instead of `useState` from React:

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

2. We will have the state in a single object, so define a type for the state beneath the import statements:

```
type State = {
  name: string | undefined;
  score: number;
```

```
  loading: boolean;
};
```

3. Next, let's also define types for all the action objects:

```
type Action =
  | {
      type: 'initialize';
      name: string;
    }
  | {
      type: 'increment';
    }
  | {
      type: 'decrement';
    }
  | {
      type: 'reset';
    };
```

These action objects represent all the ways in which state can change. The action object types are combined using a union type, allowing an action to be any of these.

4. Now, define the following reducer function underneath the type definitions:

```
function reducer(state: State, action: Action): State {
  switch (action.type) {
    case 'initialize':
      return { name: action.name, score: 0, loading:
false };
    case 'increment':
      return { ...state, score: state.score + 1 };
    case 'decrement':
      return { ...state, score: state.score - 1 };
    case 'reset':
      return { ...state, score: 0 };
    default:
      return state;
  }
}
```

The reducer function contains a `switch` statement that makes appropriate state changes for each type of action.

Notice the nice IntelliSense when referencing the `state` and `action` parameters:

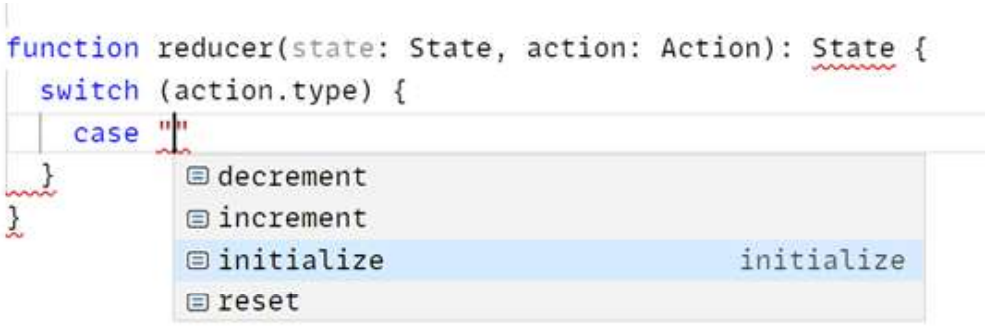

Figure 4.5 – IntelliSense inside the reducer function

5. Inside the `PersonScore` component, replace the `useState` calls with the following `useReducer` call:

```
const [{ name, score, loading }, dispatch] = useReducer(
  reducer,
  {
    name: undefined,
    score: 0,
    loading: true,
  }
);
```

The state has been initialized with an `undefined` name, a score of `0`, and `loading` set to `true`.

The current state value has been destructured into `name`, `score`, and `loading` variables. If you hover over these destructured state variables, you will see that their types have been inferred correctly.

6. We now need to amend the places in the component that update the state. Start with the effect function and dispatch an initialize action after the person has been returned:

```
useEffect(() => {
  getPerson().then(({ name }) =>
    dispatch({ type: 'initialize', name })
  );
}, []);
```

7. Lastly, dispatch the relevant actions in the button click handlers:

```
<button onClick={() => dispatch({ type: 'increment' })}>
  Add
</button>
<button onClick={() => dispatch({ type: 'decrement' })}>
  Subtract
</button>
<button onClick={() => dispatch({ type: 'reset' })}>
  Reset
</button>
```

8. If you try clicking the buttons in the running app, they will correctly update.

That completes our exploration of the `useReducer` Hook. It is more useful for complex state management situations than `useState`, for example, when the state is a complex object with related properties and state changes depend on previous state values. The `useState` Hook is more appropriate when the state is based on primitive values independent of any other state.

We will continue to expand the `PersonScore` component in the following sections. Next, we will learn how to move the focus to the **Add** button using the ref Hook.

Using the ref Hook

In this section, we will learn about the ref Hook and where it is useful. We will then walk through a common use case of the ref Hook by enhancing the `PersonScore` component we have been working on.

Understanding the ref Hook

The ref Hook is called `useRef` and it returns a variable whose value is persisted for the lifetime of a component. This means that the variable doesn't lose its value when a component re-renders.

The value returned from the ref Hook is often referred to as a **ref**. The ref can be changed without causing a re-render.

Here's the syntax for `useRef`:

```
const ref = useRef(initialValue);
```

An initial value can optionally be passed into `useRef`. The type of the ref can be explicitly defined in a generic argument for `useRef`:

```
const ref = useRef<Ref>(initialValue);
```

The generic argument is useful when no initial value is passed or is `null`. This is because TypeScript won't be able to infer the type correctly.

The value of the ref is accessed via its `current` property:

```
console.log("Current ref value", ref.current);
```

The value of the ref can be updated via its current property as well:

```
ref.current = newValue;
```

A common use of the `useRef` Hook is to access HTML elements imperatively. HTML elements have a `ref` attribute in JSX that can be assigned to a ref. The following is an example of this:

```
function MyComponent() {
  const inputRef = useRef<HTMLInputElement>(null);
  function doSomething() {
    console.log(
      "All the properties and methods of the input",
      inputRef.current
    );
  }
  return <input ref={inputRef} type="text" />;
}
```

The ref used here is called `inputRef` and is initially `null`. So, it is explicitly given a type of `HTMLInputElement`, which is a standard type for input elements. The ref is then assigned to the `ref` attribute on an input element in JSX. All the input's properties and methods are then accessible via the ref's `current` property.

Next, we will use the `useRef` Hook in the `PersonScore` component.

Using the ref Hook

We will enhance the `PersonScore` component we have been working on to use `useRef` to move the focus to the **Add** button. To do this, carry out the following steps. All the code snippets used are available at `https://github.com/PacktPublishing/Learn-React-with-`

`TypeScript-2nd-Edition/blob/main/Chapter4/Section3-Using-the-ref-hook/src/PersonScore.tsx`:

1. Open `PersonScore.tsx` and import `useRef` from React:

```
import { useEffect, useReducer, useRef } from 'react';
```

2. Create a ref for the **Add** button just below the `useReducer` statement:

```
const [ ... ] = useReducer( ... );

const addButtonRef = useRef<HTMLButtonElement>(null);

useEffect( ... )
```

The ref is named `addButtonRef` and is initially `null`. It is given the standard `HTMLButtonElement` type.

> **Note**
> All the standard HTML elements have corresponding TypeScript types for React. Right-click on the `HTMLButtonElement` type and choose **Go to Definition** to discover all these types. The React TypeScript types will open containing all the HTML element types.

3. Assign the ref to the `ref` attribute on the **Add** button JSX element:

```
<button
  ref={addButtonRef}
  onClick={() => dispatch({ type: 'increment' })}
>
  Add
</button>
```

4. Now that we have a reference to the **Add** button, we can invoke it's `focus` method to move the focus to it when the person's information has been fetched. Let's add another effect to do this below the existing effect that fetches the person:

```
useEffect(() => {
  getPerson().then(({ name }) =>
    dispatch({ type: 'initialize', name })
  );
}, []);
```

```
useEffect(() => {
  if (!loading) {
    addButtonRef.current?.focus();
  }
}, [loading]);

if (loading) {
  return <div>Loading ...</div>;
}
```

The effect is executed when the `loading` state is `true`, which will be after the person has been fetched.

Notice the `?` symbol after the `current` property on the ref. This is the **optional chaining** operator, and it allows the `focus` method to be invoked without having to check that `current` is not `null`. Visit the following link for more information about optional chaining: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining`.

We could have moved the focus to the **Add** button in the existing effect as follows:

```
useEffect(() => {
  getPerson().then(({ name }) => {
    dispatch({ type: 'initialize', name });
    addButtonRef.current?.focus();
  });
}, []);
```

However, this is mixing the concerns of fetching data, setting state, and setting focus to a button. Mixing concerns like this can make components hard to understand and change.

5. If you refresh the browser containing the running app, you will see a focus indicator on the **Add** button:

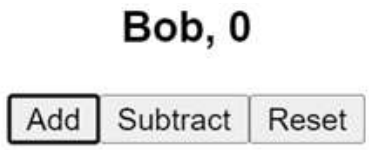

Figure 4.6 – The focused Add button

If you press the *Enter* key, you will see that the **Add** button is clicked and the score incremented. This proves that the **Add** button is focused.

That completes the enhancement and our exploration of the ref Hook.

To recap, the `useRef` Hook creates a mutatable value and doesn't cause a re-render when changed. It is commonly used to access HTML elements in React imperatively.

Next, we will learn about the memo Hook.

Using the memo Hook

In this section, we will learn about the memo Hook and where it is useful. We will then walk through an example in the `PersonScore` component we have been working on.

Understanding the memo Hook

The memo Hook creates a memoized value and is beneficial for values that have computationally expensive calculations. The Hook is called `useMemo` and the syntax is as follows:

```
const memoizedValue = useMemo(() => expensiveCalculation(),
[]);
```

A function that returns the value to memoize is passed into `useMemo` as the first argument. The function in this first argument should perform the expensive calculation.

The second argument passed to `useMemo` is an array of dependencies. So, if the `expensiveCalculation` function has dependencies `a` and `b`, the call will be as follows:

```
const memoizedValue = useMemo(
  () => expensiveCalculation(a, b),
  [a, b]
);
```

When any dependencies change, the function in the first argument is executed again to return a new value to memoize. In the previous example, a new version of `memoizedValue` is created every time `a` or `b` changes.

The type of the memoized value is inferred but can be explicitly defined in a generic parameter on `useMemo`. The following is an example of explicitly defining that the memoized value should have a `number` type:

```
 const memoizedValue = useMemo<number>(
  () => expensiveCalculation(),
  []
);
```

Next, we will experiment with `useMemo`.

Using the memo Hook

We will use the `PersonScore` component we have been working on to play with the `useMemo` Hook. To do so, carry out the following steps. The code snippets used are available at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter4/Section4-Using-the-memo-hook`:

1. Open `PersonScore.tsx` and import `useMemo` from React:

```
import {
  useEffect,
  useReducer,
  useRef,
  useMemo
} from 'react';
```

2. Add the following expensive function below the import statements:

```
function sillyExpensiveFunction() {
  console.log("Executing silly function");
  let sum = 0;
  for (let i = 0; i < 10000; i++) {
    sum += i;
  }
  return sum;
}
```

The function adds all the numbers between `0` and `10000` and will take a while to execute.

3. Add a call to the function in the `PersonScore` component beneath the effects:

```
useEffect( ... );

const expensiveCalculation = sillyExpensiveFunction();

if (loading) {
  return <div>Loading ...</div>;
}
```

4. Add the result of the function call to the JSX underneath `name` and `score`:

```
<h3>
  {name}, {score}
</h3>
<p>{expensiveCalculation}</p>
<button ... >
  Add
</button>
```

5. Refresh the browser containing the app and click the buttons. If you look in the console, you will see that the expensive function is executed every time the component is re-rendered after a button click.

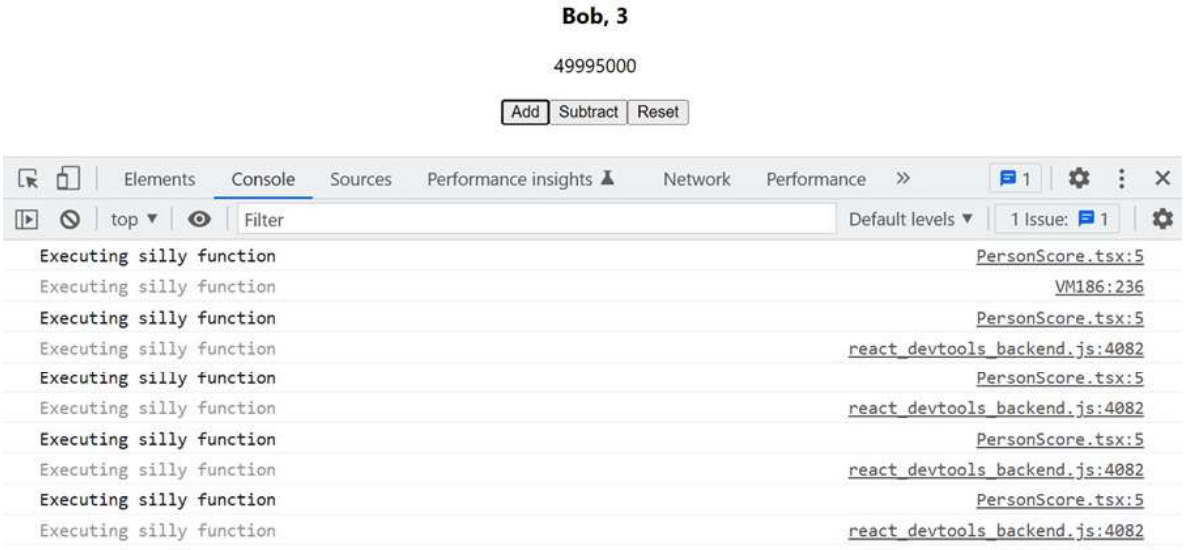

Figure 4.7 – The expensive function executed multiple times

Remember that a double render occurs in development mode and React's Strict Mode. So, once a button is clicked, you will see **Executing silly function** in the console twice.

An expensive function executing each time a component is re-rendered can lead to performance problems.

6. Rework the call to `sillyExpensiveFunction` as follows:

```
const expensiveCalculation = useMemo(
  () => sillyExpensiveFunction(),
  []
);
```

The `useMemo` Hook is used to memoize the value from the function call.

7. Refresh the browser containing the running app and click the buttons. If you look in the console, you will see that the expensive function isn't executed when the buttons are clicked because the memoized value is used instead.

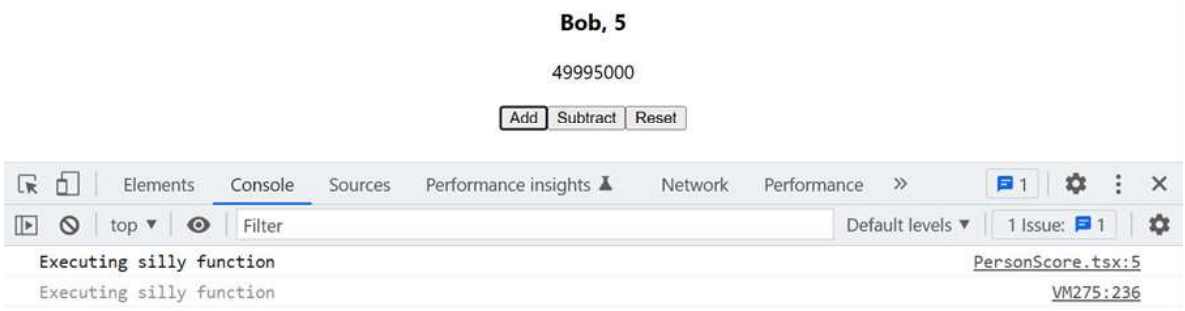

Figure 4.8 – The expensive function call memoized

That completes our exploration of the `useMemo` Hook. The takeaway from this section is that the `useMemo` Hook helps improve the performance of function calls by memoizing their results and using the memoized value when the function is re-executed.

Next, we will look at another Hook that can help performance.

Using the callback Hook

In this section, we will learn about the callback Hook and where it is useful. We will then use the Hook in the `PersonScore` component we have been working on.

Understanding the callback Hook

The callback Hook memoizes a function so that it isn't recreated on each render. The Hook is called `useCallback` and the syntax is as follows:

```
const memoizedCallback = useCallback(() => someFunction(), []);
```

A function that executes the function to memoize is passed into `useCallback` as the first argument. The second argument passed to `useCallback` is an array of dependencies. So, if the `someFunction` function has dependencies `a` and `b`, the call will be as follows:

```
const memoizedCallback = useCallback(
  () => someFunction(a, b),
  [a, b]
);
```

When any dependencies change, the function in the first argument is executed again to return a new function to memoize. In the previous example, a new version of `memoizedCallback` is created every time `a` or `b` changes.

The type of the memoized function is inferred but can be explicitly defined in a generic parameter on `useCallback`. Here is an example of explicitly defining that the memoized function has no parameters and returns `void`:

```
 const memoizedValue = useCallback<() => void>(
  () => someFunction (),
  []
);
```

A common use case for `useCallback` is to prevent unnecessary re-renders of child components. Before trying `useCallback`, we will take the time to understand when a component is re-rendered.

Understanding when a component is re-rendered

We already understand that a component re-renders when its state changes. Consider the following component:

```
export function SomeComponent() {
  const [someState, setSomeState] = useState('something');
  return (
    <div>
      <ChildComponent />
      <AnotherChildComponent something={someState} />
      <button
        onClick={() => setSomeState('Something else')}
      ></button>
    </div>
  );
}
```

When `someState` changes, `SomeComponent` will re-render – for example, when the button is clicked. In addition, `ChildComponent` and `AnotherChildComponent` will re-render when `someState` changes. This is because a component is re-rendered when its parent is re-rendered.

It may seem like this re-rendering behavior will cause performance problems – particularly when a component is rendered near the top of a large component tree. However, it rarely does cause performance issues. This is because the DOM will only be updated after a re-render if the virtual DOM

changes, and updating the DOM is the slow part of the process. In the preceding example, the DOM for `ChildComponent` won't be updated when `SomeComponent` is re-rendered if the definition of `ChildComponent` is as follows:

```
export function ChildComponent() {
  return <span>A child component</span>;
}
```

The DOM for `ChildComponent` won't be updated during a re-render because the virtual DOM will be unchanged.

While this re-rendering behavior generally doesn't cause performance problems, it can cause performance issues if a computationally expensive component is frequently re-rendered or a component with a slow side effect is frequently re-rendered. For example, we would want to avoid unnecessary re-renders in components with a side effect that fetches data.

There is a function called `memo` in React that can be used to prevent unnecessary re-renders. The memo function can be applied as follows to `ChildComponent` to prevent unnecessary re-renders:

```
export const ChildComponent = memo(() => {
  return <span>A child component</span>;
});
```

The `memo` function wraps the component and memoizes the result for a given set of props. The memoized function is then used during a re-render if the props are the same. Note that the preceding code snippet uses arrow function syntax so that the component can be a named export.

In summary, React's `memo` function can prevent the unnecessary re-rendering of slow components.

Next, we will use the `memo` function and the `useCallback` Hook to prevent unnecessary re-renders.

Using the callback Hook

We will now refactor the `PersonScore` component by extracting the **Reset** button into a separate component called `Reset`. This will lead to unnecessary re-rendering of the `Reset` component, which we will resolve using React's `memo` function and the `useCallback` Hook.

To do so, carry out the following steps. The code snippets used are available at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter4/Section5-Using-the-callback-hook`:

1. Start by creating a new file in the `src` folder for the reset button called `Reset.tsx` with the following content:

```
type Props = {
  onClick: () => void,
```

```
};
export function Reset({ onClick }: Props) {
  console.log("render Reset");
  return <button onClick={onClick}>Reset</button>;
}
```

The component takes in a click handler and displays the reset button. The component also outputs **render Reset** to the console so that we can clearly see when the component is re-rendered.

2. Open `PersonScore.tsx` and import the `Reset` component:

```
import { Reset } from './Reset';
```

3. Replace the existing reset button with the new `Reset` component as follows:

```
<div>
  ...
  <button onClick={() => dispatch({ type: 'decrement'
})}>
    Subtract
  </button>
  <Reset onClick={() => dispatch({ type: 'reset' })} />
</div>;
```

4. Go to the app running in the browser and open React's DevTools. Make sure the **Highlight updates when components render.** option is ticked in the **Components** panel's settings:

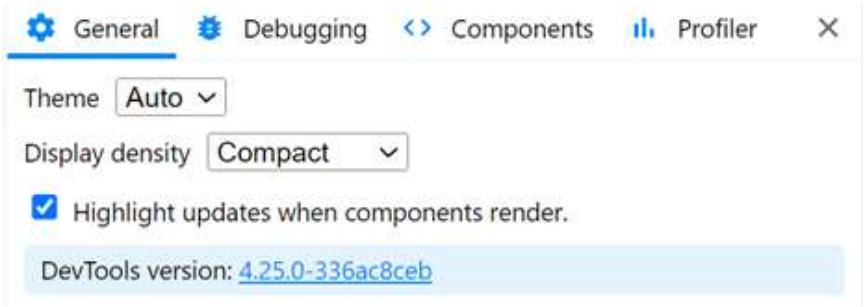

Figure 4.9 – The re-render highlight option

5. In the browser, the **Reset** button will work as it did before. Click this button as well as the **Add** and **Subtract** buttons. If you look at the console you'll notice that `Reset` is unnecessarily re-rendered. You will also see the re-render highlight around the **Reset** button.

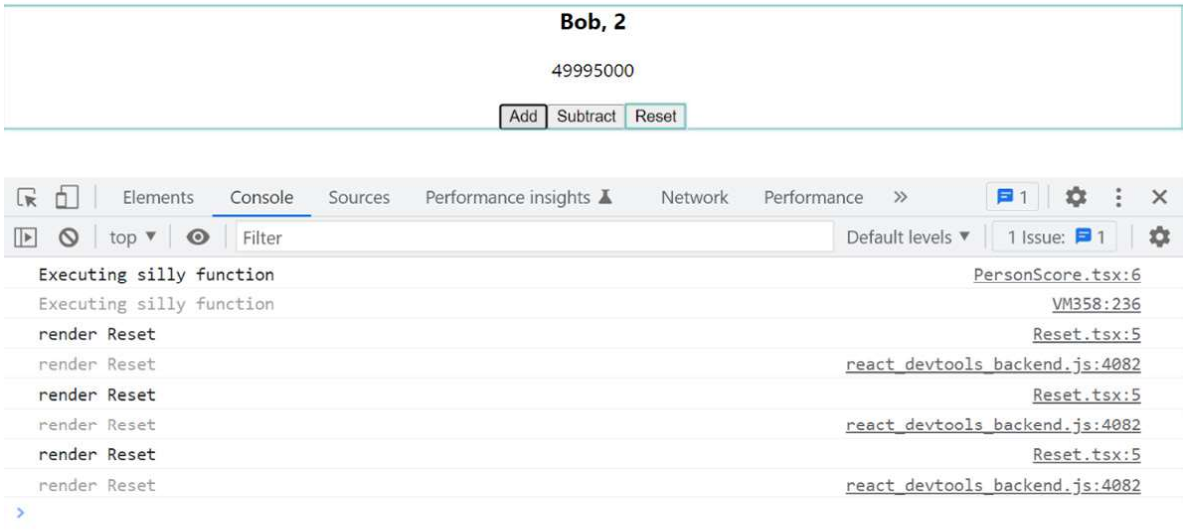

Figure 4.10 – The unnecessary re-renders of the Reset component

6. Use the browser's DevTools to inspect the DOM. To do this, right-click on the **Reset** button and choose **Inspect**. Click the buttons and observe the DOM elements. The DevTools in Chrome highlight elements when they are updated. You will see that only the `h3` element content was updated – none of the other elements are highlighted due to an update occurring.

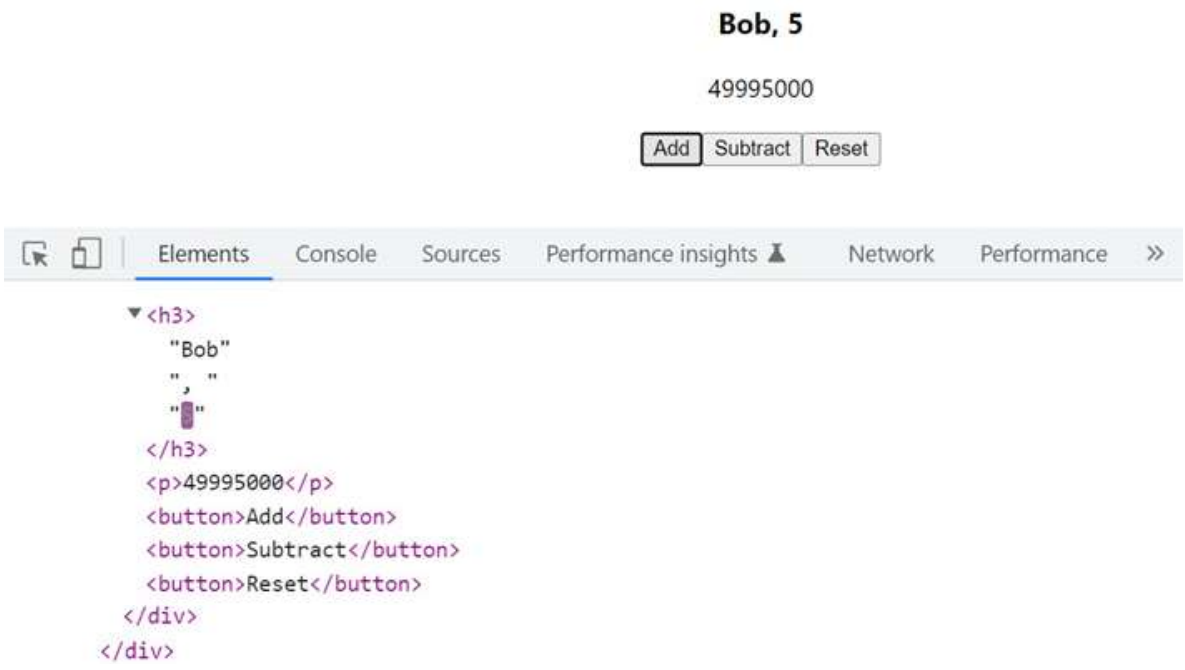

Figure 4.11 – The h3 element was updated after a re-render

Even though `Reset` is unnecessarily re-rendered, it doesn't result in a DOM update. In addition, `Reset` isn't computationally expensive and doesn't contain any side effects. So, the unnecessary render isn't really a performance problem. However, we will use this example to learn how to use React's `memo` function, and the `useCallback` Hook can prevent the unnecessary render.

7. We will now add React's `memo` function to try to prevent unnecessary re-renders. Open `React.tsx` and add a React import statement to import `memo` at the top of the file:

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

8. Now, wrap `memo` around the `Reset` component as follows:

```
export const Reset = memo(({ onClick }: Props) => {
  console.log("render Reset");
  return <button onClick={onClick}>Reset</button>;
});
```

9. In addition, add the following line beneath the `Reset` component definition so that it has a meaningful name in React's DevTools:

```
Reset.displayName = 'Reset';
```

10. In the browser, click the **Add**, **Subtract**, and **Reset** buttons. Then, look at the console and notice that `Reset` is *still* unnecessarily re-rendered.

11. We will use React's DevTools to start to understand why Reset is still unnecessarily re-rendered when its result is memoized. Open the **Profiler** panel and click the cog icon to open the settings. Go to the **Profiler** settings section and make sure **Record why each component rendered while profiling.** is ticked:

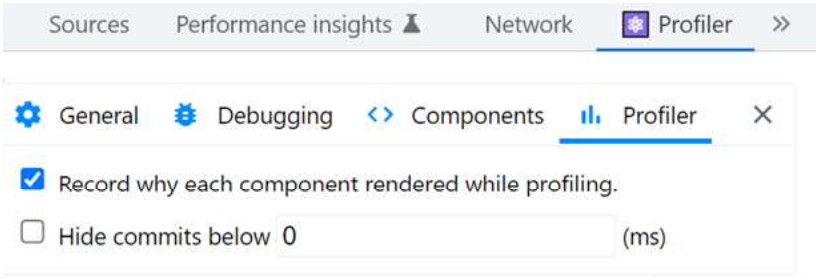

Figure 4.12 – Ensuring the Record why each component rendered while profiling. option is ticked

12. Click the blue circle icon to start profiling and then click the **Add** button in our app. Click the red circle icon to stop profiling.

13. In the flamegraph that appears, click the **Reset** bar. This gives useful information about the `Reset` component re-render:

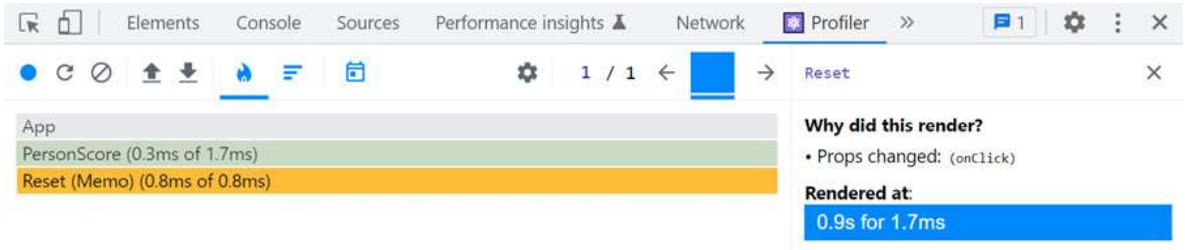

Figure 4.13 – Information about the Reset re-render

So, the unnecessary `Reset` render is happening because the `onClick` prop changes. The `onClick` handler contains the same code, but a new instance of the function is created on every render. This means `onClick` will have a different reference on each render. The changing `onClick` prop reference means that the memorized result from `Reset` isn't used and a re-render occurs instead.

14. We can use the `useCallback` Hook to memoize the `onClick` handler and prevent the re-render. Open `PersonScore.tsx` and start by refactoring the handler into a named function:

```
function handleReset() {
  dispatch({ type: 'reset' });
}

if (loading) {
  return <div>Loading ...</div>;
}

return (
  <div>
    ...
    <Reset onClick={handleReset} />
  </div>
);
```

15. Now, add `useCallback` to the React import statement:

```
import {
  useEffect,
  useReducer,
```

```
    useRef,
    useMemo,
    useCallback
  } from 'react';
```

16. Lastly, wrap `useCallback` around the click handler we just created:

```
const handleReset = useCallback(
  () => dispatch({ type: 'reset' }),
  []
);
```

17. Now, if you click the **Add**, **Subtract**, and **Reset** buttons, you will notice that `Reset` is no longer unnecessarily re-rendered.

    That completes our exploration of the `useCallback` Hook.

Here's a quick recap of everything we learned in this section:

- A component is re-rendered when its parent is re-rendered.
- React's `memo` function can be used to prevent unnecessary re-renders to child components.
- `useCallback` can be used to memoize functions. This can be used to create a stable reference for function props passed to child components to prevent unnecessary re-renders.
- React's `memo` function and `useCallback` should be used wisely – make sure they help performance before using them because they increase the complexity of the code.

Next, we will summarize the chapter.

Summary

In this chapter, we learned that all React Hooks must be called at the top level of a function component and can't be called conditionally.

The `useEffect` Hook can be used to execute component side effects when it is rendered. We learned how to use `useEffect` to fetch data, which is a common use case.

`useReducer` is an alternative to `useState` for using state, and we experienced using both approaches in our `PersonScore` example component. `useState` is excellent for primitive state values. `useReducer` is great for complex object state values, particularly when state changes depend on previous state values.

The `useRef` Hook creates a mutatable value and doesn't cause a re-render when changed. We used `useRef` to set focus to an HTML element after it was rendered, which is a common use case.

The `useMemo` and `useCallback` Hooks can be used to memoize values and functions, respectively, and can be used for performance optimization. The examples we used for these Hooks were a little contrived and using `useCallback` didn't improve performance, so remember to check that the use of these Hooks does improve performance.

So far in this book, the components we have created are unstyled. In the next chapter, we will learn several approaches for styling React components.

Questions

Answer the following questions to check what you have learned about React Hooks:

1. The following component renders some text for 5 seconds. This is problematic though – what is the problem?

```
export function TextVanish({ text }: Props) {
  if (!text) {
    return null;
  }
  const [textToRender, setTextToRender] = useState(text);
  useEffect(() => {
    setTimeout(() => setTextToRender(""), 5000);
  }, []);
  return <span>{textToRender}</span>;
}
```

2. The following code is a snippet from a React component that fetches some data and stores it in state. There are several problems with this code though – can you spot any of the problems?

```
const [data, setData] = useState([]);
useEffect(async () => {
  const data = await getData();
  setData(data);
});
```

3. How many times will the following component re-render in production mode when the button is clicked? Also, what will the button content be after one click?

```
export function Counter() {
  const [count, setCount] = useState(0);
  return (
    <button
```

```
      onClick={() => {
        setCount(count + 1);
        setCount(count + 1);
        setCount(count + 1);
      }}
    >
      {count}
    </button>
  );
}
```

4. How many times will the following component re-render in production mode when the button is clicked? Also, what will the button content be after one click?

```
export function CounterRef() {
  const count = useRef(0);
  return (
    <button
      onClick={() => {
        count.current = count.current + 1;
      }}
    >
      {count.current}
    </button>
  );
}
```

5. Consider the following `reducer` function:

```
type State = { steps: number };
type Action =
  | { type: 'forward'; steps: number }
  | { type: 'backwards'; steps: number };

function reducer(state: State, action: Action): State {
  switch (action.type) {
    case 'forward':
      return { ...state, steps: state.steps + action.
steps };
```

```
    case 'backwards':
      return { ...state, steps: state.steps - action.
        steps };
    default:
      return state;
  }
}
```

What will the type of the `action` parameter be narrowed down to in the `"backwards"` switch branch?

6. Consider the following `Counter` component:

```
export function Counter() {
  const [count, setCount] = useState(0);
  const memoCount = useMemo(() => count, []);
  return (
    <div>
      <button onClick={() => setCount(count + 1)}>
        {memoCount}
      </button>
    </div>
  );
}
```

What will the button content be after it is clicked once?

7. Consider the following `Counter` component:

```
export function Counter() {
  const [count, setCount] = useState(0);
  const handleClick = useCallback(() => {
    setCount(count + 1);
  }, []);
  return (
    <div>
      <button onClick={handleClick}>{count}</button>
    </div>
  );
}
```

What will the button content be after it is clicked twice?

Answers

1. The problem with the component is that both `useState` and `useEffect` are called conditionally (when the `text` prop is defined), and React doesn't allow its Hooks to be called conditionally. Placing the Hooks before the `if` statement resolves the problem:

```
export function TextVanish({ text }: Props) {
  const [textToRender, setTextToRender] = useState(text);
  useEffect(() => {
    setTimeout(() => setTextToRender(""), 5000);
  }, []);
  if (!text) {
    return null;
  }
  return <span>{textToRender}</span>;
}
```

2. The main problem with the code is that the effect function can't be marked as asynchronous with the `async` keyword. A solution is to revert to the older promise syntax:

```
const [data, setData] = useState([]);
useEffect(() => {
  getData().then((theData) => setData(theData));
});
```

The other major problem is that no dependencies are defined in the call to `useEffect`. This means the effect function will be executed on every render. The effect function sets some state, which causes a re-render. So, the component will keep re-rendering, and the effect function will keep executing indefinitely. An empty array passed into the second argument of `useEffect` will resolve the problem:

```
useEffect(() => {
  getData().then((theData) => setData(theData));
}, []);
```

Another problem is that the `data` state will have the `any[]` type, which isn't ideal. In this case, it is probably better to explicitly define the type of the state as follows:

```
const [data, setData] = useState<Data[]>([]);
```

The last problem is that the data state could be set after the component has been unmounted, which can lead to memory leaks. A solution is to set a flag when the component is unmounted and not set the state when the flag is set:

```
useEffect(() => {
  let cancel = false;
  getData().then((theData) => {
    if (!cancel) {
      setData(theData);
    }
  });
  return () => {
    cancel = true;
  };
}, []);
```

3. The button will only render once in production mode because state changes are batched.

   The state isn't changed until the next render, so clicking the button once will result in `count` being set to `1`, which means the button content will be `1`.

4. The button will not re-render when the button is clicked because changes to a ref don't cause a re-render.

   The `counter` ref will be incremented when the button is clicked. However, because a re-render doesn't occur, the button content will still be `0`.

5. TypeScript will narrow the type of the `action` parameter to `{ type: 'backwards'; steps: number }` in the `'backwards'` switch branch.

6. The button's content will always be `0` because the initial count of `0` is memoized and never updated.

7. The button content will be `1` after one click and will stay as `1` after subsequent clicks. So, after two clicks, it will be `1`.

   The key here is that the `handleClick` function is only created when the component is initially rendered because `useCallback` memoizes it. So, the `count` state will always be `0` within the memoized function. This means the `count` state will always be updated to `1`, which will appear in the button content.