3 Optimizing React performance

This chapter covers

- Understanding React rendering
- Optimizing performance using memoization
- Controlling rendering with dependency arrays

Navigating the complex world of React development requires more than just coding skills; it also demands an in-depth understanding of how React components behave and render. Have you ever pondered why a component re-renders or perhaps why it refuses to do so? The answers are often hidden in the subtle interplay of component relationships, state changes, and React's internal rendering logic.

Dispelling common myths about React rendering is essential. The idea that component re-renders are triggered only by property changes, for example, is a misconception that could lead developers astray. Understanding these nuances is key to writing efficient and effective React code.

Two of the central topics when it comes to (mis)understanding how React renders are memoization and dependency arrays. *Memoization*—a cornerstone in React optimization—is a technique centered on rendering efficiently and avoiding unnecessary recalculations. Using React's `memo()`, `useMemo`, and `useCallback` hooks effectively can significantly enhance your application's performance.

At the core of React's reactive nature are *dependency arrays*, which are used in hooks like `useEffect` and `useMemo`. These arrays are crucial in determining how and when your components update, making them fundamental to your React toolkit. It's important to balance optimiza-

tion with practicality, however. Overoptimizing sometimes introduces complexity without tangible benefits, making it crucial to optimize judiciously.
This chapter delves into the intricacies of React's rendering process, offering insights into optimization strategies, the role of memoization, and the effective use of dependency arrays. Embark on this journey to deepen your understanding and mastery of React, ensuring that your applications are not just functional but also optimally efficient.

Note The source code for the examples in this chapter is available at **https://reactlikea.pro/ch03**.

3.1 Understanding React rendering

To optimize React rendering, we need to understand it first. A functional component will render for one of three reasons:

- The component has just been *mounted*. (It was not in the component tree before and is now.)
- The parent component just re-rendered.
- The component uses a hook that has flagged this component for re-render.

That's it. If none of these things happens, your component will not re-render, and that's a guarantee. If any one of the three happens, your component will re-render for sure. But React might batch rendering after several of these things happen. If a state value changes *and* the parent component re-renders, for example, the component might re-render once, or it might re-render twice. That process is controlled by React and depends on subtle timing details.

You should have a fairly good understanding of all these things by now with your basic knowledge of React, so I won't go into detail. But I will discuss some misconceptions about this list. In particular, I'll go over these two myths:

- *React re-renders a component if the properties change.* Technically, that statement is not true, and that technicality matters.

- *In React 18 using Strict Mode, React mounts every component twice.* The reality is slightly different.

Let's discuss these two myths because they are important.

3.1.1 Changing properties is irrelevant

There's a common belief in React circles that components re-render because their properties change, but that's not the case, as we can prove in two ways:

- We can create a component that has properties that change but doesn't re-render.
- We can create a component that is rendered many times with the same properties and that re-renders every time.

Both of these situations are easy to imagine. First, we need a component that clearly shows whether it re-renders. Let's create that component as shown in the following listing.

Listing 3.1 A component that highlights re-renders

```
import { useEffect } from "react";
import { useRef } from "react";
export function Rerenderable() {
  const isFirst = useRef(true);
  useEffect(() => {
    isFirst.current = false;    #1
  }, []);
  const style = #2
    { color: isFirst.current ? "red" : "blue" };    #2
  const text = isFirst.current  #2
    ? "First render" : "Not first render";    #2
  return <p style={style}>{text}</p>;
}
```

#1 Sets the isFirst ref to false in an effect that, as always, runs after the current render

#2 Results in a different render on first render versus subsequent renders

Next, let’s create two components that use this component to highlight the two points.

Listing 3.2 Triggering re-renders against the myth

```
import { useRef, useState } from "react";
import { Rerenderable } from "./Rerenderable";
function RerenderWithoutPropsChange() {
  const [, setCount] = useState(0);    #1
  return (
    <div>
      <button
        onClick={() => setCount((c) => c + 1)}>     #2
        Click to re-render
      </button>
      <Rerenderable />     #3
    </div>
  );
}
function NoRerenderWithPropsChange() {
  const count = useRef(0);     #4
  return (
    <div>
      <button onClick={() => count.current++}>     #5
        Click to re-render
      </button>
      <Rerenderable count={count.current} />     #6
    </div>
  );
}
export default function App() {
  return (
    <main>
      <h4>Re-renders without changing properties</h4>
      <RerenderWithoutPropsChange />
      <h4>No re-render with changing properties</h4>
      <NoRerenderWithPropsChange />
    </main>
  );
}
```

#1 Initializes state, but we need to update it, not read it

#2 Updates state to force a re-render

#3 Embeds the re-renderable component without any props

#4 Defines a ref, which we can update without triggering a re-render

#5 Updates the value inside the ref

#6 Includes the re-renderable component with a property extracted from inside the ref

EXAMPLE: RERENDER

This example is in the `ch03/rerender` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch03/rerender
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch03-rerender**.

The point of the application in listing 3.2 is that if the myth were true, the `Rerenderable` inside `RerenderWithoutPropsChange` would never re-render because the properties never change. Likewise, the `Rerenderable` component inside `NoRerenderWithPropsChange` should re-render even though the parent doesn't because the property passed to it updates. Neither of those things happens, as you can see in figure 3.1.

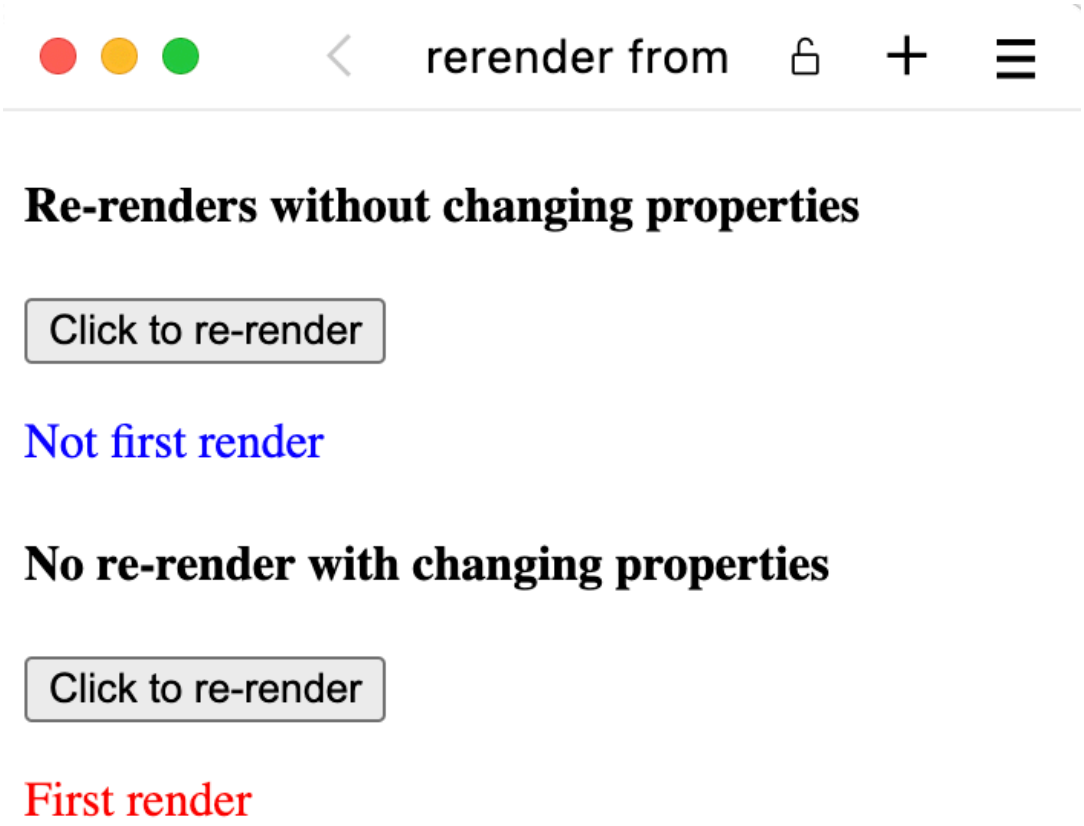

Figure 3.1 The result of clicking both buttons. If the myth were true, the top component would not re-render, whereas it clearly does, and the bottom component would re-render, whereas it clearly doesn't.

What does this mean? React components re-render when their parents do regardless of which properties they take, where those properties come from, or how those properties potentially update.

3.1.2 Repeated function calls in Strict Mode while in development

In Strict Mode during development using React 18 or later, React does several tricks to see whether your components are well designed, including running a bunch of functions multiple times. The reality can be tricky. It seems that components mount twice, but they don't; they render twice. Also, effects, initializers, and updaters are run twice (not every time, but sometimes). You don't gain anything directly from this feature, but it's a quirk of React that you need to be aware of. Let's cre-

ate a sample component that uses a bunch of these features to see how it runs in a normal run versus a Strict Mode development run.

Listing 3.3 A sample application with various effects

```
import { useEffect, useRef, useState } from "react";
let outsideCount = 0;
export default function App() {
  const ref = useRef(true);
  const [count, setCount] = useState(() => {
    console.log( #1
      "initializing count to", outsideCount   #1
    );  #1
    return outsideCount++;
  });
  console.log("rendering with ", count);    #1
  useEffect(() => {
    console.log(   #1
      "effect first time?", ref.current    #1
    );   #1
    ref.current = false;
    setCount((c) => {
      console.log(  #1
        "setting count from ", c, " to ", c + 1    #1
      );   #1
      return c + 1;
    });
    return () => console.log("cleaning up");   #1
  }, []);
  useEffect(() => {
    console.log("effect every time?", count);   #1
    return () => console.log("cleaning up every time");  #1
  }, [count]);
  return <h1>What is the count? {count}</h1>;
}
```

#1 Logs out what happens in various stages of the component life cycle

First, let's examine the output when we run this component under normal circumstances. For this purpose, we can run the component with React 17, run it without Strict Mode, or run it in a production environment. The easiest solution is to remove Strict Mode, so we will modify `main.jsx` as shown in the following listing.

Listing 3.4 Running without Strict Mode

```
import ReactDOM from "react-dom/client";
import App from "./App.jsx";
ReactDOM.createRoot(document.getElementById("root")).render(
  /*<React.StrictMode>*/    #1
  <App />   #1
  /*</React.StrictMode>*/   #1
);
```

#1 Removes Strict Mode by hiding these JSX nodes inside comment blocks

EXAMPLE: STRICT-MODE

This example is in the `ch03/strict-mode` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch03/strict-mode
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch03-strict-mode**.

So what's the output? It's kind of what we would expect:

```
initializing count to 0
rendering with  0
effect first time? true
setting count from  0  to  1
effect every time? 0
rendering with  1
cleaning up every time
effect every time? 1
```

Let's draw what happened in a (surprisingly complex) diagram (figure 3.2).

Now let's see how the output changes when we re-enable Strict Mode (by reverting `main.jsx` to the original):

```
initializing count to 0
rendering with  0
initializing count to 1
rendering with  1
effect first time? true
setting count from  1  to  2
effect every time? 1
cleaning up
cleaning up every time
effect first time? false
effect every time? 1
setting count from  2  to  3
rendering with  3
setting count from  2  to  3
rendering with  3
cleaning up every time
effect every time? 3
```

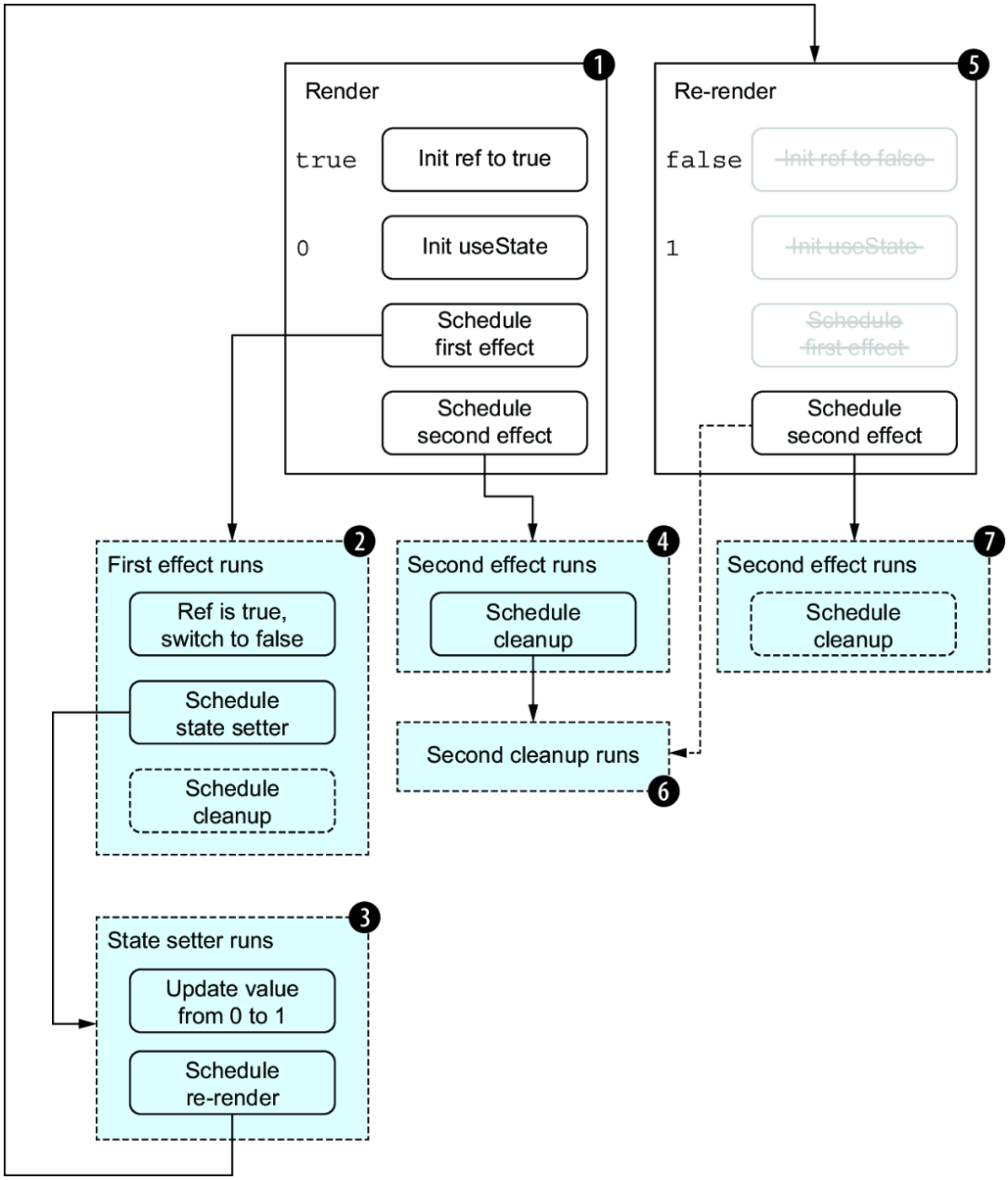

Figure 3.2 The flow of the component under normal circumstances. The large numbers are the order in which the various parts run. On the initial render (1), state is initialized and both effects are scheduled (steps 2 and 4), and when the first effect runs (2), it schedules a setter (3), which, in turn, schedules a re-render. On the re-render (5), the state is not reinitialized and the first effect doesn't run, but the second effect does run (7), making sure to clean up first (6). Note that two of the scheduled cleanups never run (the ones with a dashed border) because those effects are never rescheduled in this application's life cycle.

That code has a lot more lines of output! We see multiple initializers, and we see the first effect running multiple times but with different output. Strict Mode even cleans up the first effect, though we normally wouldn't expect that to happen before the component unmounts because it's scheduled with an empty dependency array. Let's try to dia-

gram what happens during Strict Mode in figure 3.3, but beware—it's getting complicated!

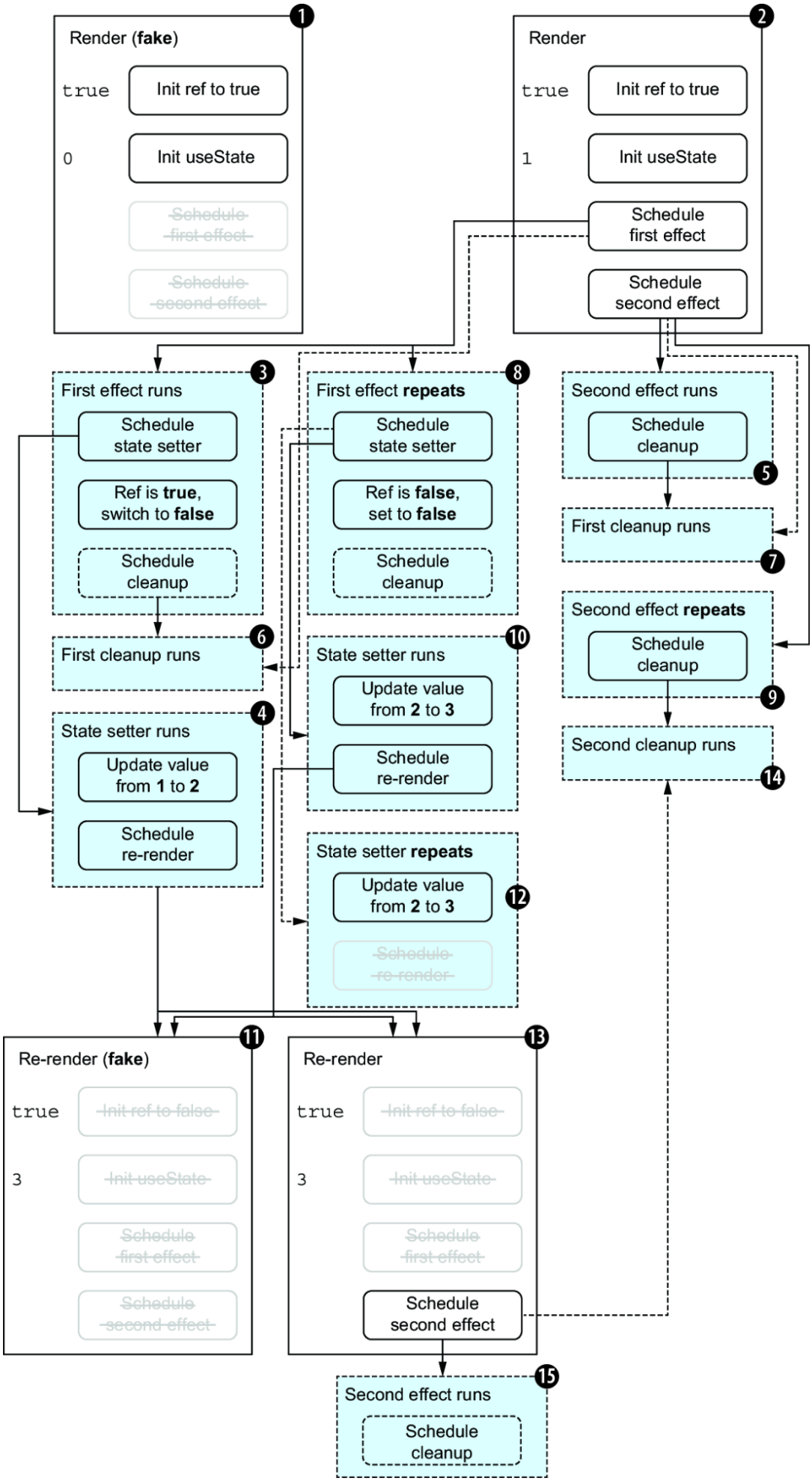

Figure 3.3 This time around, several things trigger twice, including renders (steps 1 and 11), initializers (runs in both steps 1 and 2),

effects (repeats in steps 8 and 9), and state setters (repeats in 12). The order is also a bit weird.

Figure 3.3 is the most complex diagram I have ever made. Getting it right took me a long time, and I already knew most of this stuff, so don't worry too much if you don't understand it completely.

React executes all those different types of functions in seemingly arbitrary order and with seemingly arbitrary repeats to allow you to catch potential bugs before they become production problems. React reserves the right to create smarter components in the future, when offstage components might suspend while they're not displayed (such as when they're outside the viewport or in an inactive browser window). Only when those components are redisplayed will they reanimate. You must allow all these functions to rerun so your components will reinitialize correctly.

Though that example seems to involve new worries, they're mostly edge-case situations anyway. Stick to the main takeaway in this section, and you should be safe from most problems. That main takeaway is *make sure that renders, initializers, effects, and state setters are pure functions.* By that, I mean it's crucial that these functions have no outside inputs or outside effects.

WHERE DID WE GO WRONG?

In our example in listing 3.3, the count ends up at 3 with Strict Mode even though it goes only to 1 in production. Why? Well, we're doing two things wrong:

- *The initializer has outside inputs and outside effects.* It reads from and updates a variable outside the scope of React. We should not do that. Truth be told, I did it only to break things.
- *The effect is not pure.* Sure, the state setter is pure, as it updates from the old value to that value + 1, but the effect is not pure, as it changes the state. To do things correctly, we should decrement the state in a cleanup function. To be fair, however, it doesn't make sense to increment a state value in an effect, so this example is made up. In reality, you'd increment based on a user action such as a pointer click, keyboard press, or form submission, and the events would trigger only once.

TIP Some known problems can arise from effects running twice. The React documentation has a great list of suggestions on how to circumvent these problems at **https://mng.bz/oeoZ**.

3.2 Optimizing performance by minimizing re-rendering

Several times, I have mentioned the importance of minimizing unnecessary renders. But how important is it to minimize re-rendering, and what tools can we use?

JavaScript tries to run at about 60 frames per second in most browsers, which is only 16 milliseconds (ms) per frame. Each time React renders one or more components, that counts as one frame. So if your entire React render takes more than 16 ms, your browser will start to treat your script as slow and start dropping frames. For some applications, this situation won't matter, but if you have a lot of animations and other moving elements, it will matter, and users will notice.

Secondarily, response time matters. Research shows that a user interface should update within 0.1 second for reaction to seem instantaneous to the user. If your UI is slower, your users will notice, which might result in their double-clicking buttons because the first click doesn't work, or they may simply leave in annoyance because the app feels slow.

There are many ways to optimize JavaScript applications in general and React applications in particular, and I'm not going to cover all of them—only the ones that are particularly relevant to React. One method that I've already discussed is minimizing state updates. We want to make sure that we update the component state only when the resulting component output must change.

Another common tool in the React toolbox is memoization, which I've already mentioned a few times. In this section, you're going to see a lot more examples.

NOTE With the introduction of React 19, an experimental feature known as the React Compiler aims to automatically handle memoization, potentially making manual techniques like `memo()`, `useMemo()`, and `useCallback()` obsolete. While this feature promises to simplify performance optimizations by automating them, it remains experimental and is not yet widely adopted, so understanding and applying traditional memoization methods is still relevant for current development practices.

WHAT IS MEMOIZATION?

Memoization involves remembering the last input and output of a *pure* function and, if the function is invoked with those same inputs the next time around, returning the already calculated output rather than calling the function again. Note that it makes sense only for a pure function whose return value depends solely on its inputs—never any outside information or randomness.

This process can sometimes be considered to be caching. But whereas caching often remembers many different values for many different inputs, memoization normally remembers only the latest call to a given function, checks whether the next call is equivalent, and then reuses the previous answer.

React has a `memo()` function that can memoize a component, but it doesn't work for noncomponent functions. If you want to memoize regular functions and not just React components, you should install a package such as `memoizee` (note the two *e*s):

```
import memoize from 'memoizee';
const rawAddition = (a,  b) => a + b;
const addition = memoize(rawAddition);
```

In this example, if we invoked the unmemoized function, `rawAddition()`, with the same values again and again, the calculation would be performed over and over. But if we invoked the memoized variant, `addition()`, with the same values again and again, the calculation would be performed only the first time, and the response would be cached. The cached response would be returned for the subsequent invocations as long as the inputs remained the same.

We can memoize bits of React applications in three ways:

- We can memoize an entire component.
- We can memoize a bit of JSX.
- We can memoize a property to be passed to a component.

I will discuss all these approaches with examples in the following subsections. After going through examples that require memoization, I'll discuss the hooks that we use to achieve this purpose in more technical detail.

3.2.1 Memoize a component

I've already mentioned something that you might find a bit weird: when a component renders, all child components also render, whether they have changed or not. This forced rendering of all children includes child components that are completely self-contained and don't take properties, they only render a static bit of JSX. Also, component-accepting properties will render even if they're given the same properties again.

We can use the `memo()` function from the React module to memoize an entire component. Then, if the component is invoked again with the same properties (or lack thereof), it will not render again but use the same response that was already calculated once.

In such a case, React optimizes the reconciliation of your component into the browser's document object model (DOM) and realizes that no new information has been created, so the DOM does not even need to be compared to the JSX. React knows that, because the JSX is not just similar but exactly the same, the DOM will already be correct. Such optimizations can save a lot of time!

Let's create a to-do list application that allows users to add to-dos. While the user is typing in the input field, we will update the internal state of the title of the to-do to be added. This approach is common for a controlled input field, but it causes a lot of renders. For our first attempt, we will implement the to-do application without memoization.

Listing 3.5 A to-do list without memoization

```
import { useState } from "react";
function Items({ items }) {     #1
  return (
    <>
      <h2>Todo items</h2>
      <ul>
        {items.map((todo) => (
          <li key={todo}>{todo}</li>
        ))}
      </ul>
    </>
  );
}
function Todo() {
  const [items, setItems] = useState(["Clean gutter", "Do dishes"]);
  const [newItem, setNewItem] = useState("");
  const onSubmit = (evt) => {
    setItems((items) => items.concat([newItem]));
    setNewItem("");
    evt.preventDefault();
  };
  const onChange =  #2
    (evt) => setNewItem(evt.target.value);    #2
  return (
    <main>
      <Items items={items} />     #3
      <form onSubmit={onSubmit}>
        <input value={newItem} onChange={onChange} />
        <button>Add</button>
      </form>
    </main>
  );
}
function App() {
  return <Todo />;
}
export default App;
```

#1 Our Items component renders the items it receives; it does not have any state itself.

#2 The Todo component does have state, and is updated every time the user types in the input field.
#3 Because the state updates on every key entered, the JSX returned in the to-do list is also regenerated every time, causing the Items component to render every time.

EXAMPLE: TODO-SIMPLE

This example is in the `ch03/todo-simple` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch03/todo-simple
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch03-todo-simple**.

If we spin this code up in the browser, we see that it works and is a decent attempt at a simple to-do application. But if we open the performance tools in our browser of choice to see what happens every time a user types in the input field, we see that the browser can spend up to 5 ms handling the keypress event. Now, 5 ms may not sound like a lot of time, but remember that we have only 16 ms per frame, and we're already spending about a third of that time handling a single input field. If other things are happening in the application, we'll quickly run behind. Let's try to memoize the `Items` component by using the `memo()` function imported from the React module.

Listing 3.6 A to-do list with memoization

```
import { memo, useState } from "react";     #1
const Items = memo( #2
  function Items({ items }) {
  return (
    <>
      <h2>Todo items</h2>
      <ul>
        {items.map((todo) => (
          <li key={todo}>{todo}</li>
        ))}
      </ul>
    </>
  );
});
function Todo() {
  const [items, setItems] = useState(["Clean gutter", "Do dishes"]);
  const [newItem, setNewItem] = useState("");
  const onSubmit = (evt) => {
    setItems((list) => list.concat([newItem]));
    setNewItem("");
    evt.preventDefault();
  };
  const onChange = (evt) => setNewItem(evt.target.value);
  return (
    <main>
      <Items items={items} />
      <form onSubmit={onSubmit}>
        <input value={newItem} onChange={onChange} />
        <button>Add</button>
      </form>
    </main>
  );
}
function App() {
  return <Todo />;
}
export default App;
```

#1 Remember to import memo().

#2 Apply it to the whole component.

EXAMPLE: TODO-MEMO

This example is in the `ch03/todo-memo` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch03/todo-memo
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch03-todo-memo**.

With this minor optimization, our render for every keypress drops to about 2 ms simply because we don't need to render the whole list every time, looping over every entry and creating a new JSX element for every to-do item that's already in the list. That change saves us a significant amount of time with a minimal amount of work. Extra good job, us!

Note In React 19, we have an improved way to handle forms: using actions, which is a new kind of combined effect and callback. Using an action for this form would have simplified the code quite a bit, but because we're using React 18, we'll stick with this slightly more verbose code.

3.2.2 Memoize part of a component

In the preceding section, the items were rendered in a different component, so we had the luxury option of memoizing the entire component. But we won't always have that option. Sometimes, the relevant part of the JSX covers multiple components. Suppose that we did not have an `Items` component but instead rendered the list items directly in the `Todo` component. What can we do?

We can do two things:

- Move the section of the component that is often unchanged to a new, separate component and memoize that component, which would take us directly back to listing 3.6.

- Use the `useMemo` hook to memoize the JSX directly in the parent component, as shown in the following listing.

Listing 3.7 A to-do list with memo hook

```
import { useMemo, useState } from "react";     #1
function Todo() {
  const [items, setItems] = useState(["Clean gutter", "Do dishes"]);
  const [newItem, setNewItem] = useState("");
  const onSubmit = (evt) => {
    setItems((items) => items.concat([newItem]));
    setNewItem("");
    evt.preventDefault();
  };
  const itemsRendered = useMemo(     #2
    () => (
      <>
        <h2>Todo items</h2>
        <ul>
          {items.map((todo) => (
            <li key={todo}>{todo}</li>
          ))}
        </ul>
      </>
    ),
    [items]    #3
  );
  const onChange = (evt) => setNewItem(evt.target.value);
  return (
    <main>
      {itemsRendered}
      <form onSubmit={onSubmit}>
        <input value={newItem} onChange={onChange} />
        <button>Add</button>
      </form>
    </main>
  );
}
function App() {
  return <Todo />;
}
export default App;
```

#1 Imports useMemo() rather than memo()

#2 Renders the item's JSX inside the useMemo hook

#3 Adds items as a dependency of the hook. If not, the list never updates even as you add items.

EXAMPLE: TODO-USEMEMO

This example is in the `ch03/todo-usememo` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch03/todo-usememo
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch03-todo-usememo**.

Once again, typing in the input field results in a render time of only 2 ms. If you remove the `useMemo` hook and render the items directly inline in the component, runtime jumps back up to 3-4 ms per event—less than before because we use fewer components but still more than 2 ms.

We could argue about which version of the to-do application has the cleanest code. I probably prefer the version that uses a memoized component (listing 3.6), but others might prefer the one in listing 3.7. The two different approaches allow us to choose either option as we see fit.

3.2.3 Memoize properties to memoized components

Let's go back to our to-do application in listing 3.6 but add a new requirement. We always want to display the to-do item `Complete` `todo` `list` at the top of the list, regardless of what is in the list of items.

Listing 3.8 A to-do list with a fixed item (excerpt)

```
...
function Todo() {
  ...
  return (
    <main>
      <Items
        items={["Complete todo list", ...items]} />     #1
      ...
    </main>
  );
}
...
```

#1 The only change from listing 3.6 is this line. Instead of passing items as the property, we create a new array with a fixed item at the start and then the rest of the items as before.

If you spin this code up in a browser and check the run time per key-press event, you see that it jumps back up to around 5 ms again. What happened?

The problem is that even if the state value is identical (even referentially identical) on every render while we're typing, we create a new array inline on every render, which has the extra item in the front. Then we pass that new array to the memoized component, but because the passed value isn't referentially identical every time, the component has to do a full render.

The good news is that we already know how to fix this problem! We need to create a value in the component that changes only when the state value changes. We have the `useMemo` hook for that purpose. Let's apply it as shown in the following listing.

Listing 3.9 A to-do list with a memoized fixed item (excerpt)

```
...
function Todo() {
  ...

  const allItems = useMemo(() =>     #1
    ["Complete todo list", ...items],  #1
    [items]);    #1
  return (  #1
    <main>
      <Items items={allItems} />    #2
      ...
    </main>
  );
}
...
```

#1 We memoize the inline-created array and save it in a variable, allItems. This hook depends on the items array, so only when that array changes do we need to update the allItems value.
#2 We pass the memoized property to the items component, which memoizes correctly and renders only when the list is updated with new content.

EXAMPLE: TODO-FIXED

This example is in the `ch03/todo-fixed` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch03/todo-fixed
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch03-todo-fixed**.

We did it! We fixed the render, and the performance is back down to about 2 ms per render while typing.

To make this application better, we should be able to delete items from the list when it's complete. Let's add a callback to our `items` compo-

nent that will be invoked on click with the relevant item to remove. We go back to a previous iteration of the to-do list to make things simpler. If we do this the naive way, we will have the same problem as before:

```
<Items
  items={items}
  onDelete={(item) => setItems(ls => ls.filter(i => i === item))}
/>
```

Because this function is defined inline in JavaScript, we essentially create a new function every time. You might think that it's the same function on each render because each function has an identical definition, but that's not how JavaScript works. Passing new properties on every render is a no-go when we're using memoization. We need our values to be referentially identical if memoization is supposed to kick in. We need to memoize this callback. What better way is there than the `useCallback` hook made for this purpose? See the following listing.

Listing 3.10 A to-do list with deletable items

```
import { memo, useCallback, useState } from "react";
const Items = memo(function Items({ items, onDelete }) {
  return (
    <>
      <h2>Todo items</h2>
      <ul>
        {items.map((todo) => (
          <li key={todo}>
            {todo}
            <button onClick={() => onDelete(todo)}>X</button>
          </li>
        ))}
      </ul>
    </>
  );
});
function Todo() {
  const [items, setItems] = useState(["Clean gutter", "Do dishes"]);
  const [newItem, setNewItem] = useState("");
  const onSubmit = (evt) => {
    setItems((items) => items.concat([newItem]));
    setNewItem("");
    evt.preventDefault();
  };
  const onChange = (evt) => setNewItem(evt.target.value);
  const onDelete = useCallback(    #1
    (item) => setItems((list) => list.filter((i) => i !== item)),
    []    #2
  );
  return (
    <main>
      <Items items={items} onDelete={onDelete} />
      <form onSubmit={onSubmit}>
        <input value={newItem} onChange={onChange} />
        <button>Add</button>
      </form>
    </main>
  );
}
function App() {
```

```
    return <Todo />;
  }
  export default App;
```

#1 We memoize the callback in a hook. We could also use the useMemo hook, but this approach is simpler.
#2 We pass an empty dependency array because our only dependency is a state setter, which is a value known to be stable.

EXAMPLE: TODO-DELETE

This example is in the `ch03/todo-delete` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch03/todo-delete
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch03-todo-delete**.

We're back where we wanted to be. Our component renders swiftly even while typing, because the "expensive" part of the component, which doesn't change during typing, is memoized properly.

Often, you will see that it is necessary to memoize properties when you start memoizing components. Property memoization is relevant for objects and arrays created inline, and even more so for functions, which is the primary reason for the existence of the `useCallback` hook and why it is used so often in React.

Memoization is not a job to leave to the end of a project when you notice that your application is a bit sluggish. Memoization is something you do while developing to ensure a smooth, optimal user experience. With the tools presented in this section, you should be able to apply this example to your own projects.

At the same time, premature optimization is also a problem. If you optimize things that are running well, you might end up with the opposite situation. You incur slight run-time penalties just by invoking opti-

mization functions, and if they don't make the application faster, they make it slower.
If you're a new developer, don't optimize prematurely. But if you know what you're doing, feel free to memoize when you predict that it's going to matter.

3.2.4 Memoization hooks in detail

In this section, I'll describe uses and best practices for two memoization hooks in detail. In short, you can memoize any value with `useMemo` and functions in particular with `useCallback`. But when should you do so, and how do you make sure that the value updates correctly if necessary?

MEMOIZE ANY VALUE WITH USEMEMO

This hook memoizes values between renders and can be used for two different purposes (or both at the same time):

- To prevent expensive recalculations
- To maintain referential equality

Although the first concept is fairly easy to grasp, the second is a lot more complex. I discuss why referential equality matters in section 3.3, where I talk more about dependency arrays; I've also already touched on the topic a few times in this chapter. In this section, I introduce the hook and show how to memoize values to prevent expensive calculations.

`useMemo` takes a function and a dependency array. If any value in the dependency array has changed since the component was last rendered, the function will be executed, and the return value of said function will be the return value of the call to `useMemo`. If no value in the dependency array has changed since the last render, the value returned in the last render will be returned again (figure 3.4).

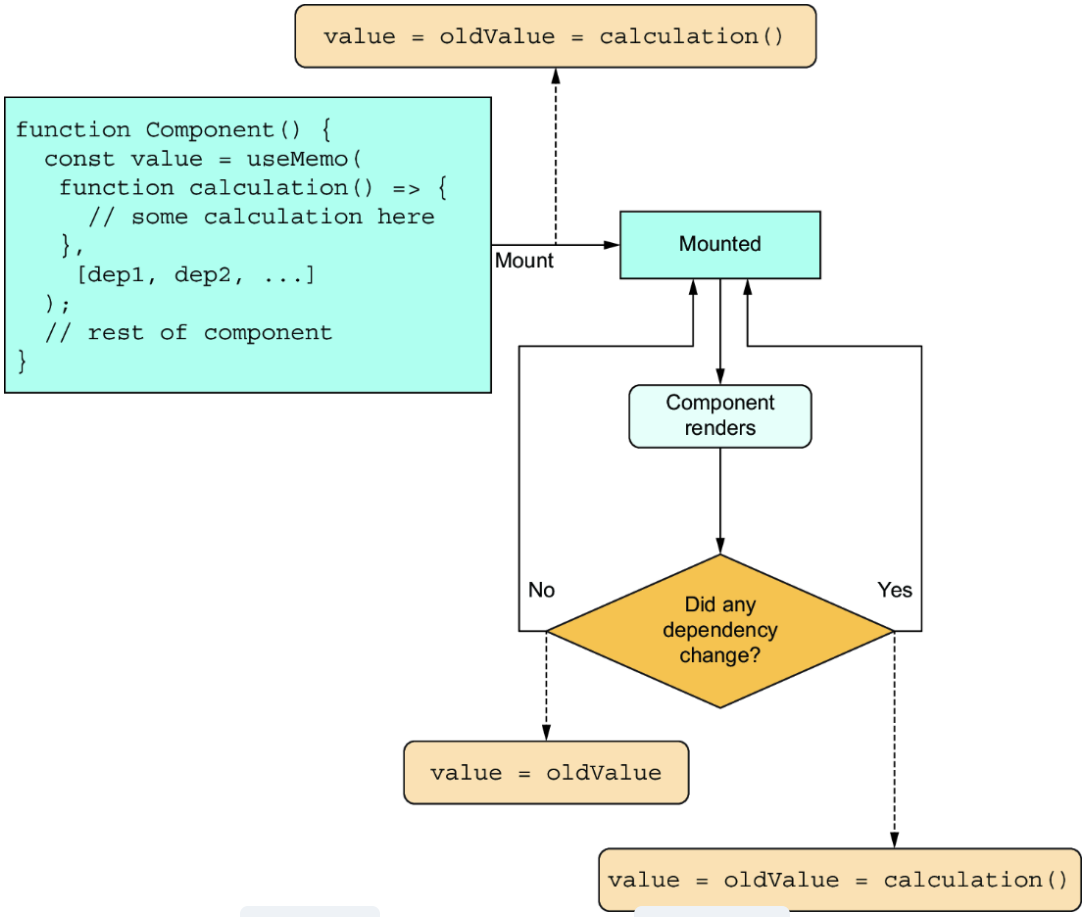

Figure 3.4 The `useMemo` flow. Note that `oldValue` doesn't exist in the component; it is an internal value accessible only to the React API and is not directly visible in the component.

Using this hook to avoid expensive recalculations is relatively simple to grasp. Suppose that we have a list of employees with some filters above the list. For this example, let's allow the user to see either all employees or only temporary workers.

We could perform this filtering of the employee list every time the component renders, but that process could be expensive, especially if the list contains 1,000 or more records. Instead, we'll use the `useMemo` hook to do the filtering only when either the source array or the filter changes value.

Listing 3.11 Memoized filtering

```
import { useMemo, useState } from "react";     #1
function Employees({ employeeList }) {
  const [showTempOnly, setShowTempOnly] = useState(false);
  const filteredList = useMemo(    #2
    () =>     #3
      employeeList.filter(({ isTemporary }) =>    #3
        showTempOnly ? isTemporary : true    #3
      ),   #3
    [employeeList, showTempOnly]     #4
  );
  return (
    <section>
      <h1>Employees</h1>
      <label>
        <input
          type="checkbox"
          onChange={() => setShowTempOnly((f) => !f)}
        />
        Show temp only?
      </label>
      <ul>
        {filteredList.map(({ id, name, salary, isTemporary }) => (
          <li key={id}>
            {name}: {salary} {isTemporary && "(temp)"}
          </li>
        ))}
      </ul>
    </section>
  );
}
function App() {
  const employeeList = [
    { name: "Bugs Bunny", salary: "$20,000", isTemporary: false },
    { name: "Daffy Duck", salary: "$15,000", isTemporary: false },
    { name: "Porky Pig", salary: "$17,000", isTemporary: true },
    ...
  ];
  return <Employees employeeList={employeeList} />;
}
export default App;
```

#1 We import the useMemo hook (along with useState) from the React package.

#2 We invoke useMemo with the two arguments it takes.

#3 The first argument is a function that performs filtering on the current array.

#4 The second argument is the dependencies of our calculation; it should be performed again only if either the list or the Boolean changes.

EXAMPLE: EMPLOYEES

This example is in the `ch03/employees` folder. You can use that example by running this command in the source folder:

```
$ npm run dev -w ch03/employees
```

Alternatively, you can go to this website to browse the code, see the example in action in your browser, or download the source code as a zip file: **https://reactlikea.pro/ch03-employees**.

You can see this application in action in figure 3.5 with the Boolean flag turned on and off.

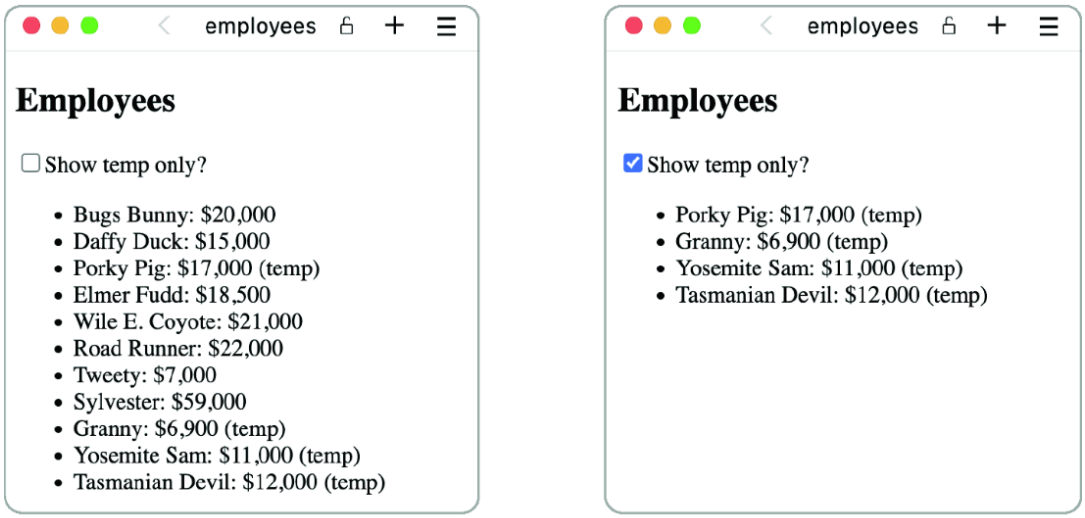

Figure 3.5 The list of employees with and without the temporary employee filter enabled

This example isn't a good one, however. Why? The only things that can change in this component are the list and the Boolean, so every time the component renders, one of those two (probably) changed. As a result, we need to do the calculation on every render because the dependencies always change. If we need to do the calculation on every render, using memoization is more expensive because of the overhead of

calling extra functions. If our component had a bunch of other properties and state values, however, it could change independently of the state and property values, and our optimization would start to help a lot.
Suppose that the component had some other state that did not refer to the filtering of the list but to the sorting of the list. In that case, our filtering calculation would not be performed simply because we sorted the list differently. The filtering would be (re)applied only when we updated the Boolean filter flag or when the source array changed, which is what we want in this case.

The second use case, and probably the more common use case for `useMemo`, involves referential equality. I'll discuss this topic in section 3.3.

MEMOIZE FUNCTIONS WITH USECALLBACK

`useCallback` is one of the least necessary hooks built into React, as it's a simple extension of `useMemo`. It does the same thing as `useMemo` if `useMemo` is used to memoize a function. `useCallback` could be defined in terms of `useMemo` simply like this:

```
function useCallback(fn, deps) {
  return useMemo(() => fn, deps);
}
```

This definition is even stated directly in the React documentation. So why do both hooks exist? `useCallback` makes memoized callbacks where referential equality is desired, and is never used to prevent expensive calculations. Callbacks are most often defined inline in the component body like so:

```
const handleClick = useCallback(
  (evt) => {
    // handle evt and do stuff
  },
  [],
);
```

If we want to define this same memoized function using `useMemo`, we would do it this way:

```
const handleClick = useMemo(
  () => (evt) => {     #1
    // handle evt and do stuff
  },
  [],
);
```

#1 This notation looks a little weird.

That double-arrow notation in the second line is easy to forget—and forgetting it changes the meaning of the code. Rather than assign a function to `handleClick` as desired, if we forget the double arrow, we assign the result of invoking said callback, which is most likely `undefined`, as we rarely return anything from event handlers. Even though this hook can do only a subset of what `useMemo` can do, we will be using `useCallback` more than `useMemo` in the remainder of this book because we memoize functions more often than other types of values.

3.3 Understanding dependency arrays

We have used dependency arrays a few times already to restrict when various hooks are triggered. We use dependency arrays for effect hooks. An empty array in an effect hook indicates that it runs only on mount, whereas a nonempty array indicates that the hook runs on mount and every time the mentioned dependencies update. We also use dependency arrays for memoization hooks; we can create stable values by using empty dependency arrays, for example.

But how do dependency arrays work in practice? How do you specify the right elements in a dependency array, and how do you make sure that the dependencies don't update too often?

First, let's reiterate which hooks use these dependency arrays to define when the hooks should take effect: `useEffect`, `useCallback`, `useMemo`, and `useLayoutEffect`. These four hooks, and only these four, use dependency arrays to conditionally trigger their effect and/or execution.

There are three general classes of dependency arrays. You don't specify an array at all, you specify an empty array, or you specify a nonempty array. These classes are exemplified as follows:

```
useEffect(() => { ... });  #1
useEffect(() => { ... }, []);   #2
useEffect(() => { ... }, [id]);  #3
```

#1 No array

#2 Empty array

#3 Nonempty array

If you don't specify a dependency array, the hook should be triggered on every render regardless of which values update in the hook (if any). If you have an empty array, your hook triggers only on mount and never again (except for cleanup functions, which trigger on unmount, but that's not your hook; that's a side effect of running your hook).

If you specify a nonempty array, your hook triggers when any of the values in the array change on a render. Any single change triggers the hook. React uses referential equality to determine whether a value has changed.

REFERENTIAL EQUALITY

In JavaScript (and many other languages), values come in two types: primitives and complex objects. JavaScript has seven primitive types (number, bigint, Boolean, string, symbol, `null`, and `undefined`) and one complex type (object). You may wonder, what about classes, regular expressions, arrays, and functions? They are also considered to be objects (though classes and functions in some sense are considered to be functions, which are a subtype of objects).

You can compare values to see whether they're strictly equal by using the triple-equals operator: `===`. The regular double-equals operator will do type conversion, so `"1" == 1`, but strict equality requires that the types be identical.

When you're comparing two primitive values, they are considered to be strictly equal if their values represent the same data, even if they are two different variables. You could compare `2 === 1+1`, for example, which would be true.

For complex types, however, strict equality means referential equality. Objects are considered to be identical only if they literally are the same object and updating one would also update the other. So `{} !== {}` and `[] !== []`, even though we're comparing two empty objects and arrays, respectively. They are not considered to be strictly equal, because they're not the same object (or array)—just similar data structures.

When we refer to React's using referential equality, we mean that React will use the strict equality operator to compare values and thus will consider objects or arrays to be identical only if they're references to the same object, and not merely two different values containing similar data.

3.3.1 What are dependencies?

Dependencies of a hook are a subset of all the variables and references that you use in the hook. A *dependency* is any local variable that exists

locally in the component scope but not any variable that also exists outside the component scope. Figure 3.6 shows a few examples.

```
const one = 1;
function Component({ two }) {
  const three = 3;
  useEffect(
    () => {
      const four = three + 1;
      const sum = one + two + three + four;
    },
    [two, three],
  );
  // Rest of component
}
```

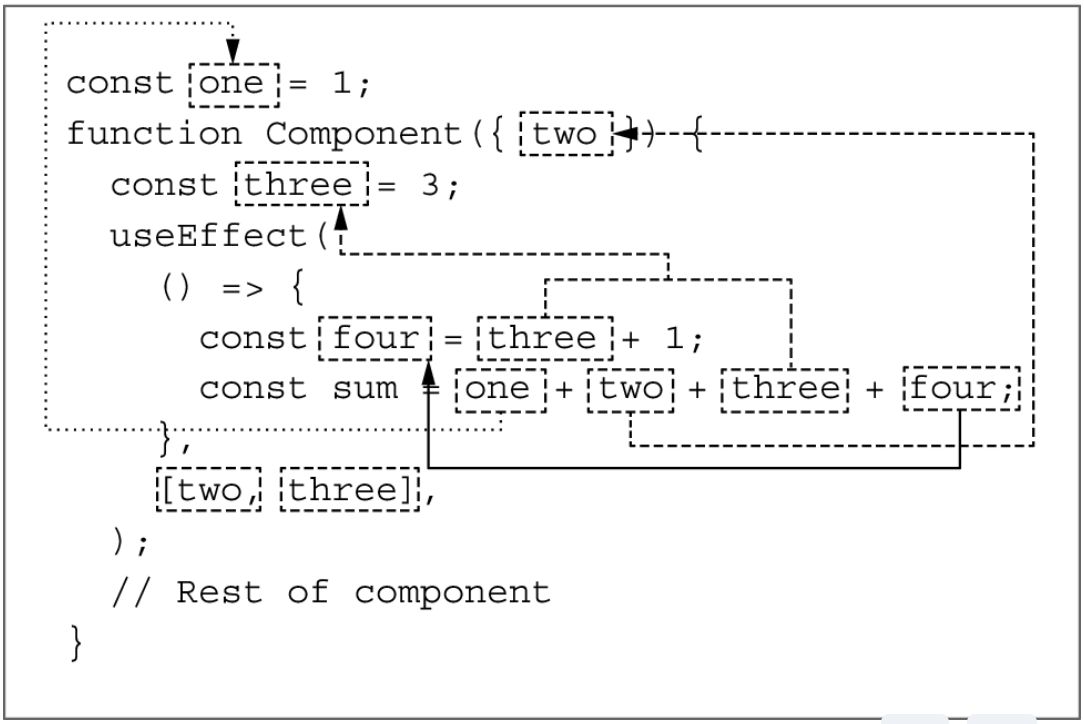

Figure 3.6 This hook uses four variables inside named `one`, `two`, `three`, and `four`. Variable `one` comes from outside the component, as noted by the dotted arrow. Variables `two` and `three` come from inside the component but outside the effect, as noted by the dashed arrows. Variable `four` comes from inside the effect itself. Only variables `two` and `three` are relevant to add as dependencies, as you see in the dashed boxes at the bottom.

Dependencies include any variable defined as a `const`, `let`, or `var` in the component, any functions defined inside the component, and any argument passed to the component (mostly properties, but potentially also forwarded references). Any function or variable defined outside the component or imported from other files is not relevant as a dependency for a hook. Finally, any variable defined inside the hook is not a dependency, as it does not exist in the outside component.

3.3.2 Run on every render by skipping the dependency array

Suppose that you want your effect to run on every render regardless of why a component re-renders. Maybe you want to track all the renders for tracking or statistical purposes. You could add a dependency array listing every single property and state value that exists, which would work if any of those values changed.

But remember that your component also re-renders because its parent component re-renders, and that parent-triggered render might not come with any change of a property or state value. So regardless of how many values you put in your dependency array, your effect will never run on every single possible render.

You have a simple solution: skip the dependency array. Don't supply any dependency array at all, and your effect will run on every render regardless of why the render happens:

```
function Component({ ... }) {
  useEffect(() => track('Component rendered'));
  ...
}
```

Note that we supply only a single argument to the `useEffect` function. We simply ignore passing a second argument.

You might ask, why run the effect in a hook at all? Why not just run the code inline, like so?

```
function Component({ ... }) {
  track('Component rendered');
  ...
}
```

Executing the tracking function inside an effect hook is recommended for optimization. The preceding `track` function might be a bit slow, and the responsiveness of this function call should not block your component from rendering. So by running the function in an effect, you decouple the execution of the effect from the rendering of the component.

MEMOIZATION WITHOUT DEPENDENCIES IS MEANINGLESS

Would you ever skip the dependency array for a memoization hook? If the memoization hook body runs for every render, the overhead of memoizing the value doesn't do anything. So you would never do that, as the code would be useless. If you use

```
const value = useMemo(() => someCalculation());
```

you might as well use this code, which is much more efficient and does the same thing:

```
const value = someCalculation();
```

NO DEPENDENCY ARRAY IS DIFFERENT FROM AN EMPTY ARRAY

Be aware that a missing dependency array is very different from an empty dependency array. The two values are polar opposites in the context of dependency arrays. A hook with an empty dependency array runs only once—on the initial mounting render of the component—and never again, whereas a hook without a dependency array runs on every single render of the component regardless of why it renders.

3.3.3 Skip stable variables from dependencies

If you have been extra attentive, you may realize that we sometimes cheat. We did not follow the best practice of always specifying all the variables used in a hook in the dependency array. We skipped that practice in listing 3.10. You get an extra gold star for noticing! I did point it out at the time, however, so maybe you get only a silver star. Here's the relevant section of the component in listing 3.10:

```
const [items, setItems] = useState(["Clean gutter", "Do dishes"]);
...
const onDelete = useCallback(
  (item) => setItems(     #1
    (list) => list.filter((i) => i !== item),
  ),
  []     #2
);
```

#1 Here, we use the variable setItems, which is clearly defined outside the effect yet inside the component.
#2 But we still specify an empty dependency array. That's not allowed, is it?

What's going on here? Are we allowed to skip listing variables as dependencies? Yes, if the variable is a stable variable. The concept of a stable variable is quite an oxymoron because it's a variable that doesn't vary. If the variable always has the same value for every render of your component, it's irrelevant to put it in the dependency array because we know that it never changes. That's partially the reason why values from outside our component don't go in the dependency array. If we define a constant outside our component or import a constant from another file, we know that it's always the same constant for every render of our component, so even if we depend on it, we don't need to consider it to be a value that can change.

In the same way, we can have variables inside our component that we know are stable—variables that never change. When it comes to functions and objects, stable values are extra important because even if a function has the same body every time, that fact does not mean that it *is* the same value.

When it comes to hooks, React defines and specifically lists some return values as stable. If you compare the returned values from certain hooks, you see that not just similar functions or objects but the same function or object is returned. We can ignore adding these values as dependencies to make our components and hooks easier to read and understand.

This is the case for the setter function returned by `useState`. The value returned can change for every render, but the setter function is always the same function reference, which is why we don't need to include it in dependency arrays.

This is also the case for the object returned by `useRef`. The object is always the same, but the value of the `current` property changes and is dynamic.

If you use a `useRef` reference or a `useState` setter inside an effect hook (or a memoized hook), you can specify it in the dependency array, but you don't have to. Both you and React know that both the reference and the setter are stable, so they never change and thus will never cause the execution of the hook to change. Specifying them as dependencies is optional. For consistency, I recommend that you either always or never include values known to be stable. (I never do.) Development teams often specify their choice in this matter in their coding standards.

You can make your own stable variables as well to make your components easier to read and understand, both for yourself as a developer and for the rest of the team. If you memoize a value in your component by using a hook, and you include an empty dependency array, the returned value is stable. A memoizing hook with an empty dependency array always returns the same value; thus, it can be considered to be stable.

Image this code for an incomplete component, in which we specify all dependencies even if they're known to be stable. The code becomes more verbose, sacrificing simplicity and understanding:

```
function Panel() {
  const [isOpen, setOpen] = useState(false);
  const toggleOpen = useCallback(
    () => setOpen(open => !open),    #1
    [setOpen],     #2
  );
  useEffect(
    () => {
      // Some effect here
      toggleOpen();    #3
    },
    [toggleOpen],    #4
  );
  ...
}
```

#1 We use a component-scoped variable . . .

#2 . . . so we specify it as a dependency.

#3 We use another component-scoped variable . . .

#4 . . . so we specify it, too, as a dependency.

Here, we specify `setOpen` as a dependency, even though (as just discussed) we can skip it because it is known to be stable; it never changes. If you are examining the code in the preceding component, however, it is not obvious that the effect runs only on mount because there is a dependency array. You have to track that dependency to check its origin, which might force you to track another list of dependencies.

If we skip the `setOpen` dependency from the `useCallback` hook, we will see that `toggleOpen` is now a stable value because it is defined in a memoizing hook with an empty dependency array. This value also will never change over the lifetime of the component, so we can skip the `toggleOpen` value from the dependency array of the effect hook. We can greatly simplify this component as follows:

```
function Panel() {
  const [isOpen, setOpen] = useState(false);
  const toggleOpen = useCallback(
    () => setOpen(open => !open),
    [],    #1
  );
  useEffect(() => toggleOpen(), []);     #1
  ...
}
```

#1 We can supply an empty dependency array for both hooks because we know that both use only values known to be stable.

This version is much easier to read and understand because you instantly know that both hooks run only a single time due to their empty lists of dependencies.

3.3.4 Get help maintaining dependency arrays

Maintaining dependency arrays can be quite troublesome. Say you're editing some effect and adding a reference to a property the component receives, but you forget to update the dependency array—the whole component starts acting weird.

In chapter 4, we discuss developer tooling. One of those tools, ESLint, has a great rule that will help you keep those dependency arrays updated. ESLint gives you an error directly in your editor if you forget to add something to the dependencies that should be there. This feature is enabled by default, so installing the package should get you going.

Summary

- React components re-render when their parent components re-render regardless of the properties they receive.
- Misconceptions in React include beliefs about property changes and double rendering in Strict Mode, which often don't reflect actual behavior.
- Memoization in React optimizes performance by caching the results of function calls and rendering, reducing unnecessary

recalculations.

- React's `memo()` function can be used to prevent unnecessary re-renders of components by comparing properties.
- The `useMemo` and `useCallback` hooks in React help optimize performance by memoizing complex calculations and functions.
- Dependencies in React hooks dictate when effects are rerun, which is crucial for optimizing performance and avoiding unnecessary updates.
- Strict equality, or referential equality, is key in React for determining whether hook dependencies have changed and re-calculation is necessary.
- React hooks such as `useEffect`, `useCallback`, `useMemo`, and `useLayoutEffect` use dependency arrays to manage hook updates.
- Omitting the dependency array in React hooks results in the hooks' running after every render, which can be useful for tracking renders or performance analysis.
- In React, stable variables such as `useState` setters and `useRef` objects don't need to be included in dependency arrays, as they don't change between renders.
- Premature optimization in React, such as unnecessary memoization, can lead to decreased performance due to the overhead of the optimization code.
- ESLint and other development tools are useful for maintaining accurate dependency arrays in React, ensuring that components behave as expected.