React Hooks

Master Modern React

One resource for everything you need
to know about React Hooks.

AISHWARYA PARAB

Introduction

Welcome to "React Hooks: Master Modern React". This eBook is designed to be your go-to resource for understanding and mastering React Hooks, one of the most powerful features introduced in React 16.8.

Whether you're preparing for interviews or looking to deepen your understanding of React, this book aims to provide clear, concise and practical insights into each hook, helping you become a more proficient React developer.

Who is This Book for?

This eBook is tailored for frontend developers who:

📍Have a basic understanding of React.
📍Want to leverage hooks to write cleaner and more efficient code.
📍Are preparing for frontend developer interviews.
📍Aim to advance their career by mastering modern React practices.

About the Author

My name is **Aishwarya Parab**, a frontend developer having industry experience in React. Currently, I am working as a **Senior Software Engineer** in India. I've also been recognised as a **Top Voice** in Frontend Development by LinkedIn. You can connect with me on LinkedIn here.

Through this book, I aim to share concise and practical knowledge on React hooks that can help you as your learning and revision guide.

Just this book is enough to clear interviews, build projects and ACTUALLY understand how hooks work in React.

How to Use This Book?

Each chapter is dedicated to a specific React Hook. The chapters are structured as follows:

1. **What the Hook Does:** A brief overview of the hook's purpose.
2. **Usage of the Hook:** How to implement the hook in your projects.
3. **Pitfalls/Caveats:** Common mistakes and how to avoid them.
4. **Examples:** Practical examples to illustrate the hook's usage in different scenarios.

The final chapter is dedicated to common **interview questions** on hooks. So, let's dive into the world of React Hooks and take your React skills to the next level!

Chapter 01: useState

What useState does?

The **useState** hook allows you to add state to functional components. Before hooks, state was only accessible in class components, but useState brought the power of state management to functional components.

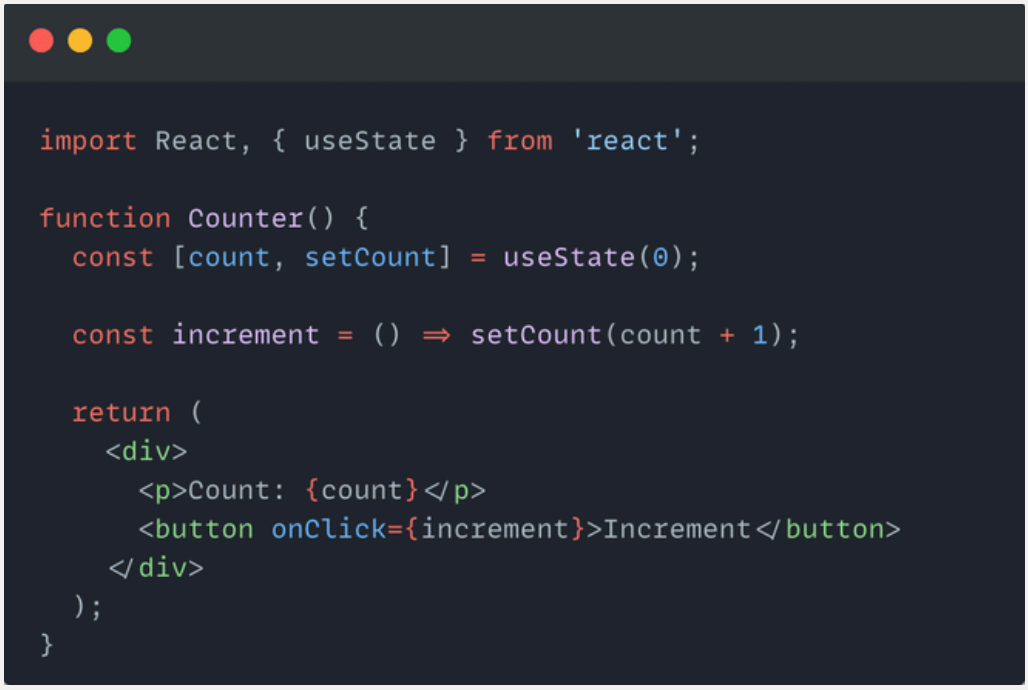
Usage

useState returns an array with exactly **two** values:

1. **The current state**. During the first render, it will match the **initialState** you have passed.
2. **The set function** that lets you update the state to a different value and trigger a re-render.

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

function MyComponent() {
  const [age, setAge] = useState(23);
  const [name, setName] = useState('Aishwarya');
  const [todos, setTodos] = useState(() => createTodos());
  // ...
```



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

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

  const increment = () => setCount(count + 1);

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

Using useState

Pitfalls

- **State Update Timing:** The set function only updates the state variable for the next render. If you read the state variable immediately after calling the set function, **you will still get the old value** that was on the screen before your call.

- **Skipping Re-renders:** If the new value you provide is identical to the current state (determined by an *Object.is* comparison), React will skip re-rendering the component and its children. This optimization ensures efficient rendering. Although React may still call your component before skipping the children, it shouldn't affect your code.

- **Batched State Updates:** React batches state updates and updates the screen after all event handlers have run and called their set functions. This prevents multiple re-renders during a single event.

```
function MyComponent() {
  const [age, setAge] = useState(23);
  const [name, setName] = useState('Aishwarya');

  function handleClick() {
    setName('Aish');
    console.log(name); // 🚩 Still "Aishwarya"!
  }
  ...
}
```

```
// age was previously 23
function handleClick() {
  setAge(age + 1); // setAge(23 + 1)
  setAge(age + 1); // setAge(23 + 1)
  setAge(age + 1); // setAge(23 + 1)
}

// Ultimately, age will be 24!
```

```
function handleClick() {
  setAge(a ⇒ a + 1); // setAge(23 ⇒ 24)
  setAge(a ⇒ a + 1); // setAge(24 ⇒ 25)
  setAge(a ⇒ a + 1); // setAge(25 ⇒ 26)
}

// ✅ New age: 26
```

```
// 🚩 Don't mutate an object in state like this:
form.firstName = 'Aishwarya';

// ✅ Replace state with a new object
setForm({
  ...form,
  firstName: 'Aishwarya'
});
```

Examples

Example 01: Counter

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

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

  const increment = () => setCount(count + 1);

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

Example 02: Toggle Button

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

function ToggleButton() {
  const [isOn, setIsOn] = useState(false);

  const toggle = () => setIsOn(!isOn);

  return (
    <button onClick={toggle}>
      {isOn ? 'ON' : 'OFF'}
    </button>
  );
}
```

Example 03: Form Input

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

function Form() {
  const [name, setName] = useState('');

  const handleChange = (event) => setName(event.target.value);

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`Hello, ${name}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" value={name} onChange={handleChange} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Chapter 02: useEffect

What useEffect does?

The **useEffect** hook lets you perform side effects in functional components. It is a close replacement for lifecycle methods like **componentDidMount**, **componentDidUpdate** and **componentWillUnmount** in class components.

Usage

useEffect returns undefined and takes in a **setup function** and optional **dependencies**:

1. Your setup function may also optionally return a cleanup function.
2. Your setup code runs when your component is added to the page (mounted).
3. After every re-render of your component where the dependencies have changed:
   - First, your **cleanup** code runs with the **old** props and state.
   - Then, your **setup** code runs with the **new** props and state.
4. Your cleanup code runs one final time after your component is removed from the page (unmounted).
5. If you specify the dependencies, your Effect runs after the initial render and after re-renders with changed dependencies.
6. An Effect with empty dependencies doesn't re-run when any of your component's props or state change. It will only run after the initial render.
7. If you pass no dependency array at all, your Effect runs after every single render (and re-render) of your component.

```
import { useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
  const [serverUrl, setServerUrl] = useState('https://localhost:1234');

  useEffect(() => {
    const connection = createConnection(serverUrl, roomId);
    connection.connect();
    return () => {
      connection.disconnect();
    };
  }, [serverUrl, roomId]);
  // ...
}
```

Using useEffect

Pitfalls

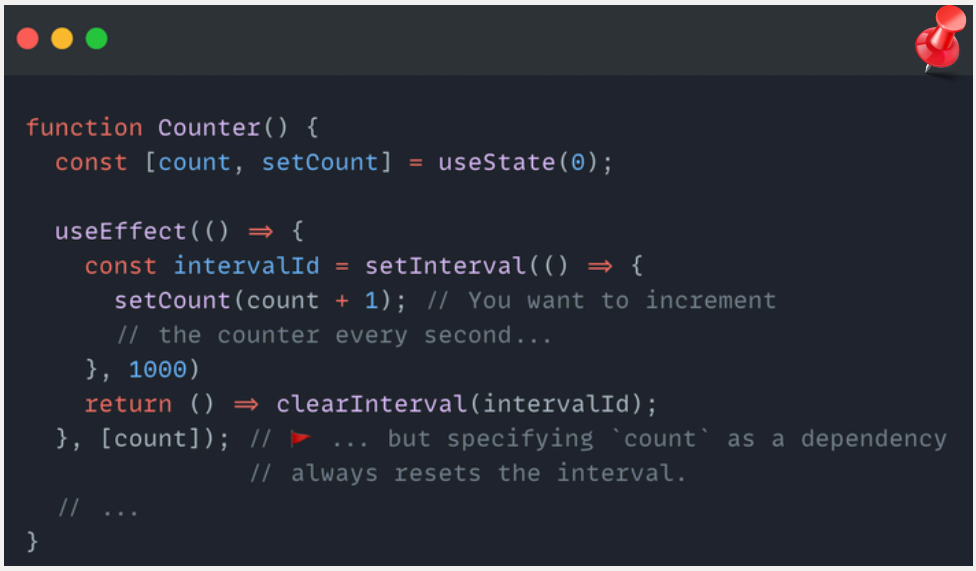
- **Dependency Array:** Always specify the dependencies in the array to avoid unnecessary re-renders. Forgetting to add dependencies can lead to bugs.

- **Cleanup Function:** If your effect returns a cleanup function, make sure it cleans up resources to prevent memory leaks.

- **Synchronous Effects:** Don't use useEffect for synchronous tasks that block rendering; instead, use it for side effects like data fetching or subscriptions.

- **Effects only run on the client:** They don’t run during server rendering.

- **UI Flickers:** If your Effect is doing something visual (for example, positioning a tooltip) and the delay is noticeable (for example, it flickers), replace useEffect with useLayoutEffect.



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

  useEffect(() => {
    const intervalId = setInterval(() => {
      setCount(count + 1); // You want to increment
      // the counter every second...
    }, 1000)
    return () => clearInterval(intervalId);
  }, [count]); // 🚩 ... but specifying `count` as a dependency
               // always resets the interval.
  // ...
}
```

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

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

  useEffect(() => {
    const intervalId = setInterval(() => {
      setCount(c => c + 1); // ✅ Pass a state updater
    }, 1000);
    return () => clearInterval(intervalId);
  }, []); // ✅ Now count is not a dependency

  return <h1>{count}</h1>;
}
```

```jsx
function MyComponent() {
  const [didMount, setDidMount] = useState(false);

  useEffect(() => {
    setDidMount(true);
  }, []);

  if (didMount) {
    // ... return client-only JSX ...
  }  else {
    // ... return initial JSX ...
  }
}
```

Examples

Example 01: Event Listener

```jsx
import React, { useEffect } from 'react';

function KeyPressLogger() {
  useEffect(() => {
    const handleKeyPress = (event) => {
      console.log(`Key pressed: ${event.key}`);
    };

    window.addEventListener('keypress', handleKeyPress);

    return () => {
      window.removeEventListener('keypress', handleKeyPress);
    };
  }, []);

  return <div>Press any key and check the console.</div>;
}
```

Example 02: Timer

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

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

  useEffect(() => {
    const timer = setInterval(() => {
      setCount((prevCount) => prevCount + 1);
    }, 1000);

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

  return <div>{count}</div>;
}
```

Examples

Example 03: Data Fetching

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

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

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch(`https://api.example.com/data?search=${query}`);
      const result = await response.json();
      setData(result);
    };

    fetchData();
  }, [query]); // The effect depends on `query`

  return (
    <div>
      {data ? (
        <ul>
          {data.map(item => (
            <li key={item.id}>{item.name}</li>
          ))}
        </ul>
      ) : (
        <p>Loading...</p>
      )}
    </div>
  );
}
```

useContext hook

NEXT

Chapter 03: useContext

What useContext does?

The **useContext** hook allows you to access the context value without needing to wrap your component in a **Consumer** component, as we would previously do. It makes context usage more straightforward and easier to manage in functional components.

Usage

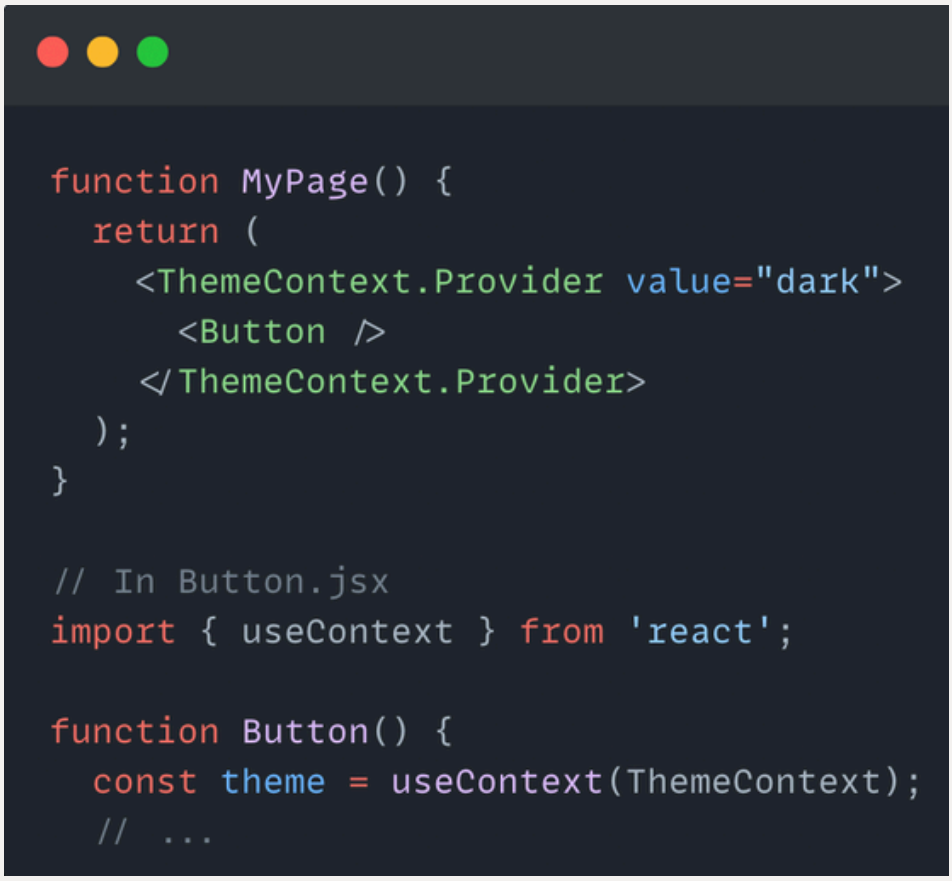
useContext returns the context value and takes in the context you created with **createContext()**.

```
import React, { useContext } from 'react';

const ThemeContext = React.createContext('light');

function ThemedComponent() {
  const theme = useContext(ThemeContext);

  return <div style={{
    background: theme === 'light' ? '#fff' : '#333',
  color: theme === 'light' ? '#000' : '#fff'
  }}>
    Current theme: {theme}
  </div>;
}
```



```jsx
function MyPage() {
  return (
    <ThemeContext.Provider value="dark">
      <Button />
    </ThemeContext.Provider>
  );
}

// In Button.jsx
import { useContext } from 'react';

function Button() {
  const theme = useContext(ThemeContext);
  // ...
```

useContext() always looks for the closest provider above the component that calls it.

It searches upwards and does not consider providers in the component from which you're calling useContext().

Pitfalls

- **Context Updates:** Be aware that useContext will re-render your component when the context value changes, which can lead to unnecessary re-renders if not managed properly.

- **Overuse of Context:** Avoid using context for every piece of state; it's best used for global state that needs to be accessed by many components.

- **Re-renders:** React automatically re-renders all the children that use a particular context starting from the provider that receives a different value. The previous and the next values are compared with the *Object.is* comparison. Skipping re-renders with memo does not prevent the children receiving fresh context values.

Examples

Example 01: Theme Toggle

```jsx
import React, { useContext, useState } from 'react';

const ThemeContext = React.createContext();

function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {
    setTheme((prevTheme) => (prevTheme === 'light' ? 'dark' : 'light'));
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

function ThemedComponent() {
  const { theme, toggleTheme } = useContext(ThemeContext);

  return (
    <div style={{ background: theme === 'light' ? '#fff' : '#333', color: theme === 'light' ? '#000' : '#fff' }}>
      Current theme: {theme}
      <button onClick={toggleTheme}>Toggle Theme</button>
    </div>
  );
}
```

Wrap App.js with ThemeProvider

Example 02: User Authentication

```jsx
import React, { useContext, useState } from 'react';

const AuthContext = React.createContext();

function AuthProvider({ children }) {
  const [user, setUser] = useState(null);

  const login = (userData) => setUser(userData);
  const logout = () => setUser(null);

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
}

function UserProfile() {
  const { user, logout } = useContext(AuthContext);

  return (
    <div>
      {user ? (
        <div>
          <p>Welcome, {user.name}</p>
          <button onClick={logout}>Logout</button>
        </div>
      ) : (
        <p>Please log in.</p>
      )}
    </div>
  );
}
```

Wrap App.js with AuthProvider

Example 03: Language Switcher

```jsx
import React, { useContext, useState } from 'react';

const LanguageContext = React.createContext();

function LanguageProvider({ children }) {
  const [language, setLanguage] = useState('en');

  const switchLanguage = (lang) => setLanguage(lang);

  return (
    <LanguageContext.Provider value={{ language, switchLanguage }}>
      {children}
    </LanguageContext.Provider>
  );
}

function LanguageSwitcher() {
  const { language, switchLanguage } = useContext(LanguageContext);

  return (
    <div>
      <p>Current language: {language}</p>
      <button onClick={() => switchLanguage('en')}>English</button>
      <button onClick={() => switchLanguage('es')}>Spanish</button>
    </div>
  );
}
```

Wrap App.js with LanguageProvider

useReducer hook

NEXT

Chapter 04: useReducer

What useReducer does?

The **useReducer** hook is an alternative to useState for managing complex state logic. It is especially useful when state transitions involve multiple sub-values. It mimics the Redux-like reducer pattern in functional components.

Usage

useReducer takes in three parameters:

1. **reducer:** The reducer function that specifies how the state gets updated. It should take the **state** and **action** as arguments, and should return the **next state**.
2. **initialArg:** The value from which the initial state is calculated. It can be a value of any type. How the initial state is calculated from it depends on the next **init** argument.
3. **optional init:** The initializer function that should return the initial state. If it's not specified, the initial state is set to **initialArg**. Otherwise, the initial state is set to the result of calling **init(initialArg)**.

useReducer returns an array with exactly **two** values:

1. **The current state**. During the first render, it's set to init(initialArg) or initialArg (if there's no init).
2. **The dispatch function** that lets you update the state to a different value and trigger a re-render.

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

function reducer(state, action) {
  // ...
}

function MyComponent() {
  const [state, dispatch] = useReducer(reducer, { age: 42 });
  // ...
```

The Dispatch Function

- The dispatch function returned by useReducer lets you update the state to a different value and trigger a re-render. You need to pass the action as the only argument to the dispatch function:

```
const [state, dispatch] = useReducer(reducer, { age: 42 });

function handleClick() {
  dispatch({ type: 'incremented_age' });
  // ...
```

- React will set the next state to the result of calling the reducer function you've provided with the **current state** and the **action** you've passed to dispatch.
- **action:** The action performed by the user. It can be a value of any type. By convention, an action is usually an object with a **type** property identifying it and, optionally, other properties with additional information.
- After dispatching, React will pass the **current state** and the **action** to your reducer function. Your reducer will calculate and return the **next state**. React will store that next state, render your component with it, and update the UI.

```jsx
import { useReducer } from 'react';

function reducer(state, action) {
  if (action.type === 'incremented_age') {
    return {
      age: state.age + 1
    };
  }
  throw Error('Unknown action.');
}

export default function Counter() {
  const [state, dispatch] = useReducer(reducer, { age: 42 });

  return (
    <>
      <button onClick={() => {
        dispatch({ type: 'incremented_age' })
      }}>
        Increment age
      </button>
      <p>Hello! You are {state.age}.</p>
    </>
  );
}
```

Pitfalls

- **State Update Timing:** The dispatch function only updates the state variable for the next render. If you read the state variable after calling the dispatch function, **you will still get the old value** that was on the screen before your call.

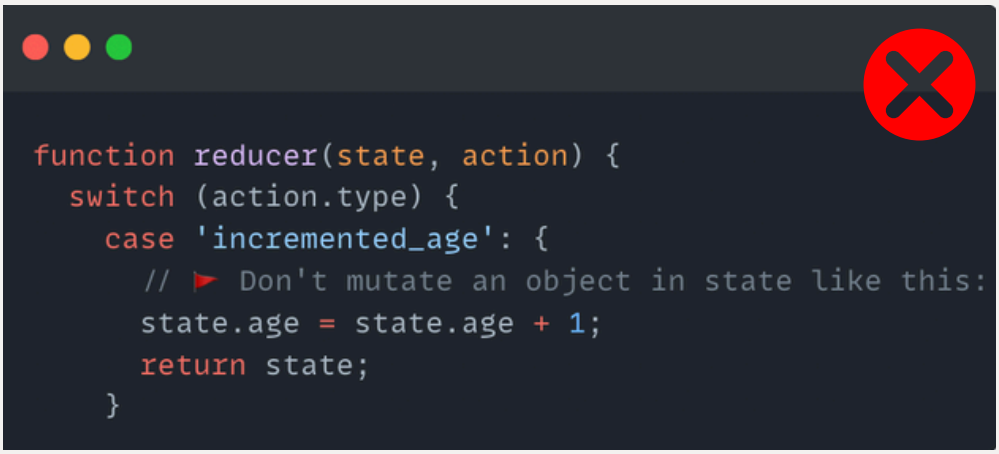
- **Skipping Re-renders:** If the new value you provide is identical to the current state, as determined by an *Object.is* comparison, React will skip re-rendering the component and its children.

- **Batched State Updates:** React batches state updates. It updates the screen after all the event handlers have run and have called their set functions. This prevents multiple re-renders during a single event.

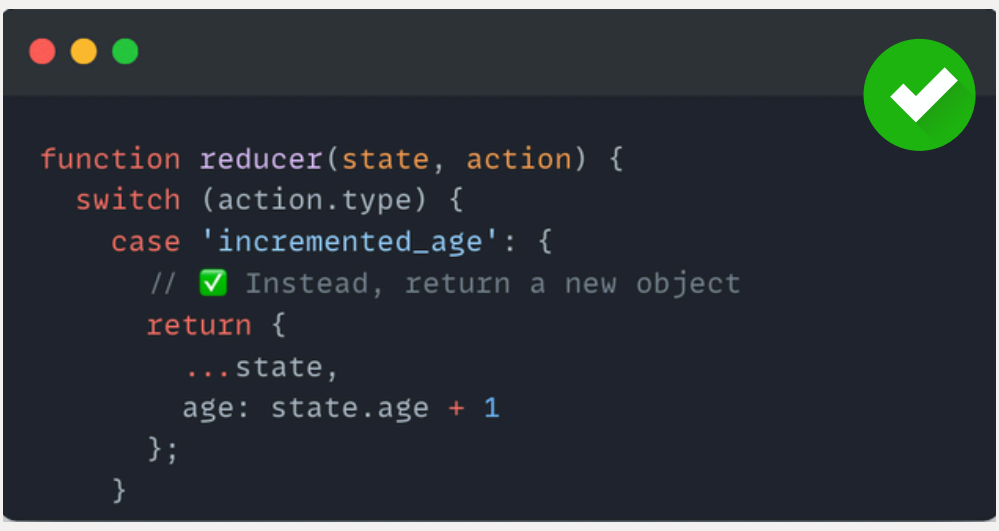
Reducer Function

```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      return {
        name: state.name,
        age: state.age + 1
      };
    }
    case 'changed_name': {
      return {
        name: action.nextName,
        age: state.age
      };
    }
  }
  throw Error('Unknown action: ' + action.type);
}
```

By convention, it is common to write it as a switch statement. For each case in the switch, calculate and return some next state.



```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      // ❌ Don't mutate an object in state like this:
      state.age = state.age + 1;
      return state;
    }
```



```
function reducer(state, action) {
  switch (action.type) {
    case 'incremented_age': {
      // ✅ Instead, return a new object
      return {
        ...state,
        age: state.age + 1
      };
    }
```

Examples

Example 01: Simple Counter

```jsx
import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>
    </div>
  );
}
```

Gear up!!
More examples
on the next page!

Example 02: Form State Management

```jsx
import React, { useReducer } from 'react';

const initialState = {
  username: '',
  email: ''
};

function reducer(state, action) {
  switch (action.type) {
    case 'setUsername':
      return { ...state, username: action.payload };
    case 'setEmail':
      return { ...state, email: action.payload };
    default:
      throw new Error();
  }
}

function Form() {
  const [state, dispatch] = useReducer(reducer, initialState);

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Form Data:', state);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={state.username}
        onChange={(e) => dispatch({ type: 'setUsername', payload: e.target.value })}
      />
      <input
        type="email"
        value={state.email}
        onChange={(e) => dispatch({ type: 'setEmail', payload: e.target.value })}
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Example 02: Form State Management

```jsx
import React, { useReducer } from 'react';

const initialState = {
  username: '',
  email: ''
};

function reducer(state, action) {
  switch (action.type) {
    case 'setUsername':
      return { ...state, username: action.payload };
    case 'setEmail':
      return { ...state, email: action.payload };
    default:
      throw new Error();
  }
}

function Form() {
  const [state, dispatch] = useReducer(reducer, initialState);

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Form Data:', state);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={state.username}
        onChange={(e) => dispatch({ type: 'setUsername', payload: e.target.value })}
      />
      <input
        type="email"
        value={state.email}
        onChange={(e) => dispatch({ type: 'setEmail', payload: e.target.value })}
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Example 03: Todo List

```jsx
import React, { useReducer, useState } from 'react';

const initialState = [];

function reducer(state, action) {
  switch (action.type) {
    case 'add':
      return [...state, { text: action.payload, completed: false }];
    case 'toggle':
      return state.map((todo, index) =>
        index === action.index ? { ...todo, completed: !todo.completed } : todo
      );
    case 'remove':
      return state.filter((_, index) => index !== action.index);
    default:
      throw new Error();
  }
}

function TodoApp() {
  const [state, dispatch] = useReducer(reducer, initialState);
  const [text, setText] = useState('');

  const handleAddTodo = () => {
    dispatch({ type: 'add', payload: text });
    setText('');
  };

  return (
    <div>
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
      />
      <button onClick={handleAddTodo}>Add Todo</button>
      <ul>
        {state.map((todo, index) => (
          <li key={index}>
            <span
              onClick={() => dispatch({ type: 'toggle', index })}
              style={{ textDecoration: todo.completed ? 'line-through' : 'none' }}
            >
              {todo.text}
            </span>
            <button onClick={() => dispatch({ type: 'remove', index })}>Remove</button>
          </li>
        ))}
      </ul>
    </div>
  );
}
```

Chapter 05: useMemo

What useMemo does?

The **useMemo** hook allows you to memoize expensive calculations, ensuring that they are only recomputed when necessary. This can improve performance by preventing unnecessary re-renders.

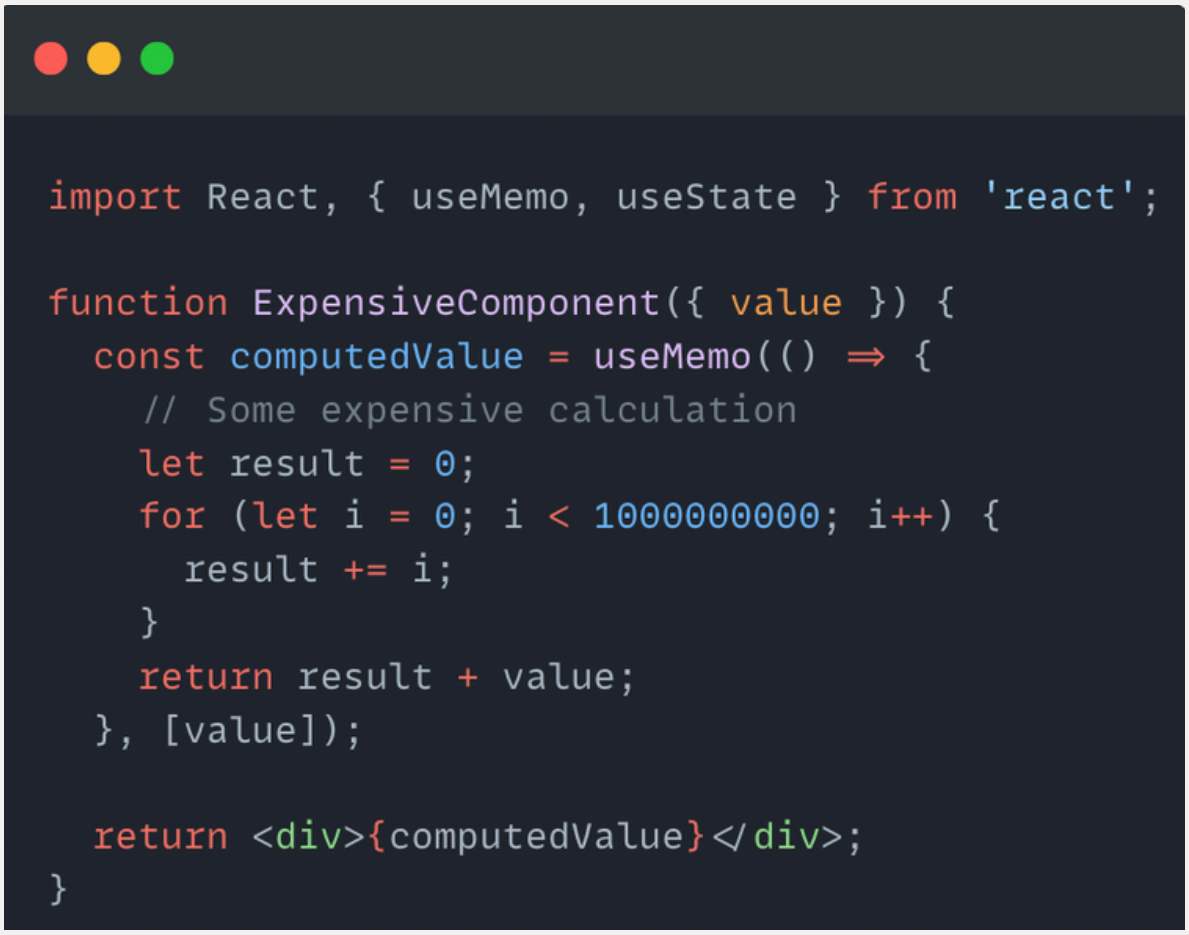
Usage

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

function TodoList({ todos, tab }) {
  const visibleTodos = useMemo(
    () ⇒ filterTodos(todos, tab),
    [todos, tab]
  );
  // ...
}
```

useMemo takes in **two** parameters:

1. **The function** calculating the value that you want to cache. It should take **no arguments** and should return a **value** of any type. React will call your function during the initial render. On next renders, React will return the same value again if the dependencies have **not changed** since the last render. Otherwise, it will call the function, return its result and store it so it can be reused later.
2. **dependencies:** React will compare each dependency with its previous value using the *Object.is* comparison.



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

function ExpensiveComponent({ value }) {
  const computedValue = useMemo(() => {
    // Some expensive calculation
    let result = 0;
    for (let i = 0; i < 1000000000; i++) {
      result += i;
    }
    return result + value;
  }, [value]);

  return <div>{computedValue}</div>;
}
```

Using useMemo

Pitfalls

- **Overuse:** Overusing useMemo can make your code harder to read and maintain. Use it only when you have performance issues.

- **Dependencies:** Make sure the dependency array is correct to avoid stale values or unnecessary recalculations. Not specifying the dependency array would make the function in useMemo() re-run every time the component renders. Thus, defeating the purpose of using it.

```jsx
function TodoList({ todos, tab }) {
  // 🔴 Recalculates every time: no dependency array
  const visibleTodos = useMemo(() => filterTodos(todos, tab));
  // ...
```

```jsx
function TodoList({ todos, tab }) {
  // ✅ Does not recalculate unnecessarily
  const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
  // ...
```

```jsx
function ReportList({ items }) {
  return (
    <article>
      {items.map(item => {
        // 🔴 You can't call useMemo in a loop like this:
        const data = useMemo(() => calculateReport(item), [item]);
        return (
          <figure key={item.id}>
            <Chart data={data} />
          </figure>
        );
      })}
    </article>
  );
}
```

Instead, extract a component for each item and memoize data for individual items.

```jsx
function ReportList({ items }) {
  return (
    <article>
      {items.map(item =>
        <Report key={item.id} item={item} />
      )}
    </article>
  );
}

function Report({ item }) {
  // ✅ Call useMemo at the top level:
  const data = useMemo(() => calculateReport(item), [item]);
  return (
    <figure>
      <Chart data={data} />
    </figure>
  );
}
```

Examples

Example 01: Expensive Calculation

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

function ExpensiveCalculationComponent() {
  const [value, setValue] = useState(0);

  const computedValue = useMemo(() => {
    // Expensive calculation
    let result = 0;
    for (let i = 0; i < 1000000000; i++) {
      result += i;
    }
    return result + value;
  }, [value]);

  return (
    <div>
      <p>Computed Value: {computedValue}</p>
      <button onClick={() => setValue(value + 1)}>Increment</button>
    </div>
  );
}
```

Example 02: Filtering a List

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

const data = [...Array(10000).keys()]; // Large list of data

function FilteredList() {
  const [filter, setFilter] = useState('');

  const filteredData = useMemo(() => {
    return data.filter(item => item.toString().includes(filter));
  }, [filter]);

  return (
    <div>
      <input
        type="text"
        value={filter}
        onChange={(e) => setFilter(e.target.value)}
      />
      <ul>
        {filteredData.map(item => (
          <li key={item}>{item}</li>
        ))}
      </ul>
    </div>
  );
}
```

Examples

Example 03: Caching API Results

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

function ApiDataComponent() {
  const [data, setData] = useState(null);
  const [query, setQuery] = useState('react');

  useEffect(() => {
    fetch(`https://api.example.com/search?q=${query}`)
      .then(response => response.json())
      .then(data => setData(data));
  }, [query]);

  const cachedData = useMemo(() => {
    return data ? data.results : [];
  }, [data]);

  return (
    <div>
      <input
        type="text"
        value={query}
        onChange={(e) => setQuery(e.target.value)}
      />
      <ul>
        {cachedData.map(item => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

useCallback hook

NEXT ➡

Chapter 06: useCallback

What useCallback does?

The **useCallback** hook returns a memoized version of a callback function that only changes if one of the dependencies has changed. Unlike useMemo, it does not call the function but **caches the function** itself.

Usage

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

export default function ProductPage({ productId, referrer, theme }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);
```

useCallback takes in **two** parameters:

- **The function** that you want to cache. It can take any arguments and return any values. React will return (not call) your function back to you during the initial render. On next renders, React will give you the same function again if the dependencies have **not changed** since the last render. Otherwise, it will give you the function that you have passed during the current render and store it in case it can be reused later.
- **dependencies:** React will compare each dependency with its previous value using the *Object.is* comparison.

useCallback v/s useMemo

The difference is in *what* they're letting you cache:

- useMemo caches the ***result*** of calling your function.
- useCallback caches *the* ***function*** *itself*.

```jsx
import { useMemo, useCallback } from 'react';

function ProductPage({ productId, referrer }) {
  const product = useData('/product/' + productId);

  const requirements = useMemo(() => { // Calls your function and caches its result
    return computeRequirements(product);
  }, [product]);

  const handleSubmit = useCallback((orderDetails) => { // Caches your function itself
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);

  return (
    <div className={theme}>
      <ShippingForm requirements={requirements} onSubmit={handleSubmit} />
    </div>
  );
}
```

useCallback v/s useMemo

Pitfalls

- **Unnecessary Usage:** Avoid using useCallback for functions that don't cause performance issues or are not passed as props.

- **Dependencies:** Make sure the dependency array is correct to avoid stale values or unnecessary recalculations. Not specifying the dependency array would make the function in useCallback() re-run every time the component renders. Thus, defeating the purpose of using it.

```
function ProductPage({ productId, referrer }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }); // 🔴 Returns a new function every time: no dependency array
  // ...
```

```
function ProductPage({ productId, referrer }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]); // ✅ Does not return a new function
                             // unnecessarily
  // ...
```

Examples

Example 01: Memoized Function

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

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

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

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

Example 02: Preventing unnecessary re-renders

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

const Child = React.memo(({ increment }) => {
  console.log('Child re-rendered');
  return <button onClick={increment}>Increment</button>;
});

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

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

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

By default, when a component re-renders, React re-renders all of its children recursively. But if the re-render is slow, you can skip re-rendering when its props are the same as on last render by wrapping it in memo.

Chapter 07: useRef

What useRef does?

The **useRef** hook provides a way to access and persist a mutable value that **does not cause a re-render when updated**. It is commonly used to reference DOM elements directly, store previous values or keep mutable objects.

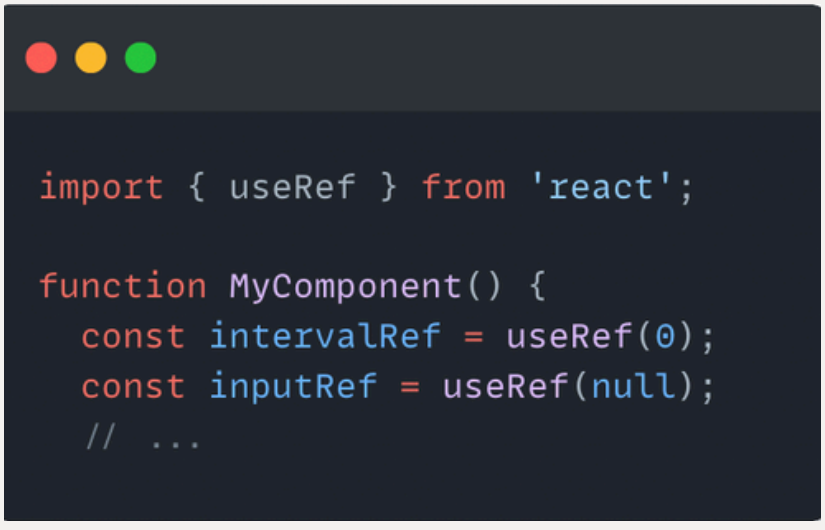
Usage

useRef takes in an initialValue:

- **initialValue:** The value you want the ref object's current property to be initially. It can be a value of any type. This argument is ignored after the initial render.

useRef returns an object with a **single** property:

- **current:** Initially, it's set to the **initialValue** you have passed. You can later set it to something else. If you pass the ref object to React as a **ref attribute** to a JSX node, React will set its current property.
- On the next renders, useRef will return the same object.

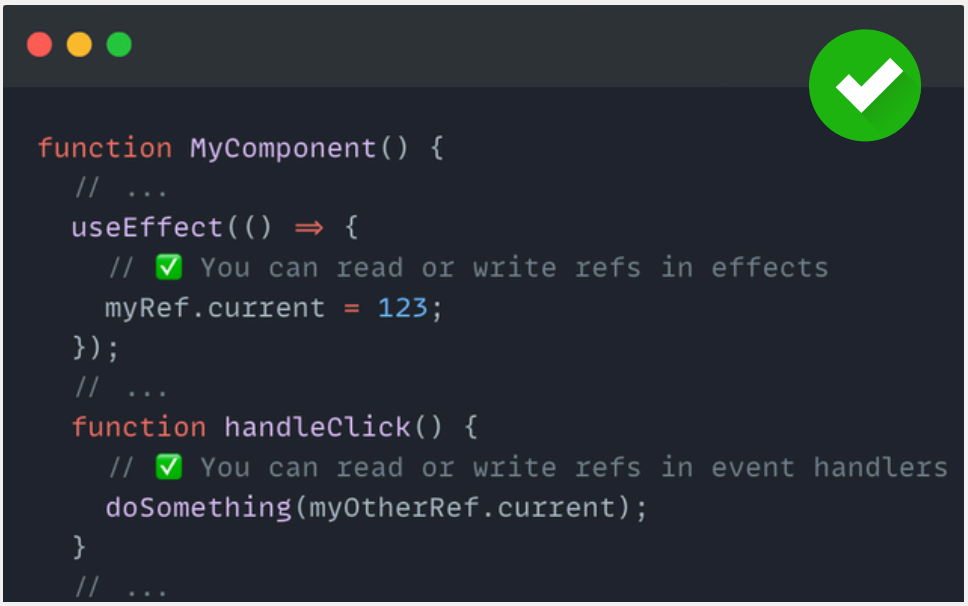


Pitfalls

- Unlike state, you can **mutate** the **ref.current** property. However, if it holds an object that is used for rendering, then you shouldn't mutate that object.

- When you change the ref.current property, React does not re-render your component. React is not aware of when you change it because a ref is a plain JavaScript object.

- Do not *write or read* ref.current during rendering, except for initialization. This makes your component's behaviour unpredictable.



Examples

Example 01: Accessing DOM Elements

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

function AccessDomElement() {
  const divRef = useRef(null);

  useEffect(() => {
    if (divRef.current) {
      divRef.current.style.backgroundColor = 'yellow';
    }
  }, []);

  return <div ref={divRef}>I am a div element</div>;
}
```

Example 02: Keeping a Mutable Value

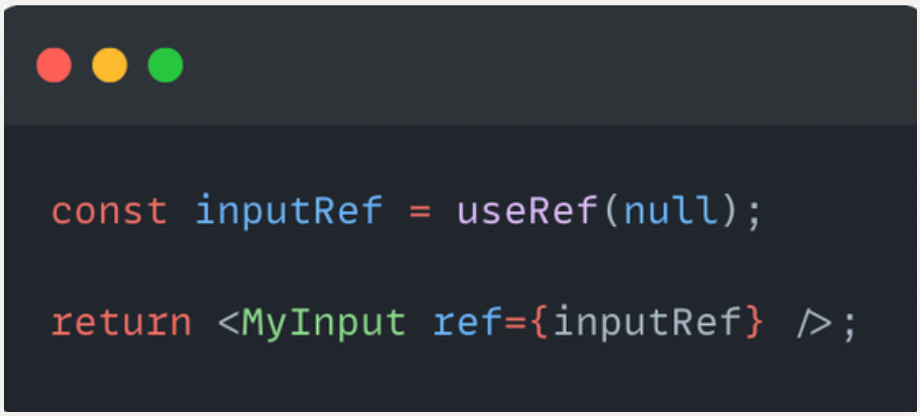
```jsx
import React, { useRef } from 'react';

function Timer() {
  const timerRef = useRef();

  const startTimer = () => {
    timerRef.current = setInterval(() => {
      console.log('Tick');
    }, 1000);
  };

  const stopTimer = () => {
    clearInterval(timerRef.current);
  };

  return (
    <div>
      <button onClick={startTimer}>Start Timer</button>
      <button onClick={stopTimer}>Stop Timer</button>
    </div>
  );
}
```



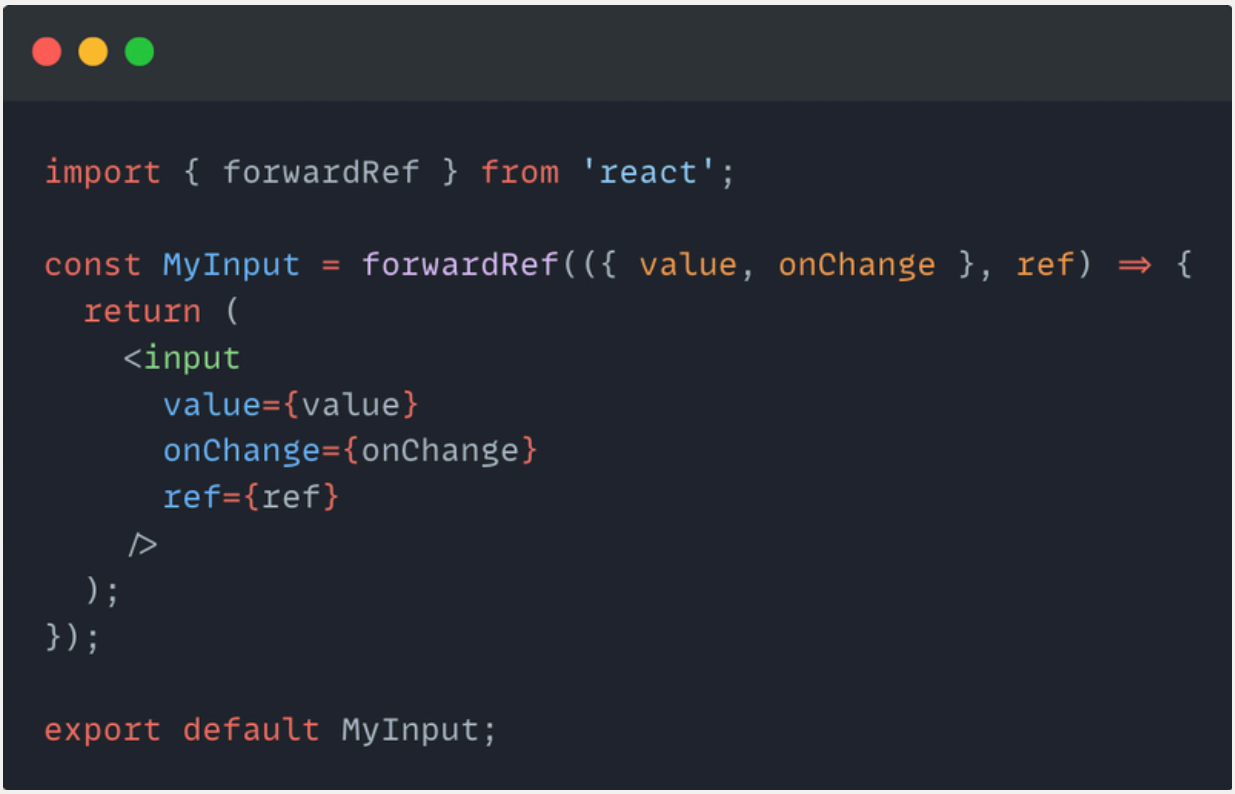
```
const inputRef = useRef(null);

return <MyInput ref={inputRef} />;
```

If you try to pass a ref to a child component, you might get an error.

By default, your own components don’t expose refs to the DOM nodes inside them.

To fix this, find the component that you want to get a ref to and then wrap it in forwardRef like this:



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

const MyInput = forwardRef(({ value, onChange }, ref) => {
  return (
    <input
      value={value}
      onChange={onChange}
      ref={ref}
    />
  );
});

export default MyInput;
```

Chapter 08: Interview Questions

1. What are React Hooks?
React Hooks are functions that let you use state and other React features without writing a class. They were introduced in React 16.8 to allow developers to use state and other React features in functional components.

2. What are the benefits of using hooks?
- Easier state management with hooks such as useState and useReducer.
- Reusability of logic with custom hooks that encapsulate and share logic between components.
- Simplified code structure as hooks allow you to organize logic inside a component into reusable, isolated functions, making the component easier to read and maintain.

3. What are the rules of hooks in React?
- Hooks should only be called at the top level of your React component.
- Hooks should only be called within React functional components or custom hooks. Not within loops, conditions or nested functions.

4. Explain the difference between useMemo() and useCallback()?

useMemo	useCallback
• Memoizes a value • Is used when you need to memoize the result of expensive calculations	• Memoizes a function • Is used when you need to memoize event handler functions to prevent unnecessary re-renders

5. What is a custom hook in React?

- A custom hook in React is a JavaScript function whose name starts with "use" and that can call other hooks. Custom hooks allow you to encapsulate and reuse stateful logic across multiple components, making your code more modular and easier to maintain.
- For example, creating a custom hook ‘useFetch’ for data fetching:

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

function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch(url);
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        const data = await response.json();
        setData(data);
      } catch (error) {
        setError(error);
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, [url]);

  return { data, loading, error };
}

export default useFetch;
```

- And using it across components:

```jsx
import React from 'react';
import useFetch from './useFetch';

function DataFetchingComponent() {
  const { data, loading, error } = useFetch('https://api.example.com/data');

  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error.message}</p>;

  return (
    <div>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}

export default DataFetchingComponent;
```

6. Can you use hooks in class components?

No, you cannot use Hooks in class components. Hooks are designed to work exclusively with functional components.

7. Explain the difference between useState() and useReducer()?

useState	useReducer
• Manages simple state in functional components • Directly updates state using the setter function • Best for simple state transitions (e.g., toggles, counters)	• Manages complex state logic and multiple sub-values • Updates state via actions dispatched to the reducer • Best for complex state transitions (e.g., forms, state objects with multiple properties)

8. How can you avoid stale closures in useEffect?

To avoid stale closures in useEffect, ensure that all variables used in the effect are included in the dependency array. This ensures the effect always has access to the latest values.

9. How can you use useEffect to mimic componentDidMount?

By passing an empty array as the second argument to **useEffect**, you can mimic **componentDidMount**. This makes the effect run only once after the initial render.

```
useEffect(() => {
  console.log('Component mounted');
}, []);
```

10. What will be the output of the following code?

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

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

  useEffect(() => {
    console.log('Effect:', count);
    return () => console.log('Cleanup:', count);
  }, [count]);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

Output:

Effect: 0
Cleanup: 0
Effect: 1
Cleanup: 1
Effect: 2
Cleanup: 2

Each time **count** changes, the effect logs the current count, and the cleanup function logs the previous count.

11. What happens if you omit dependencies in the dependency array of useEffect?

The effect will re-run on every render of the component. This can lead to performance issues, especially if the effect contains expensive operations like data fetching, complex calculations, or DOM manipulations.

12. What is the issue with the following code?

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

function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const timer = setInterval(() => {
      setSeconds(seconds + 1);
    }, 1000);
    return () => clearInterval(timer);
  }, []);

  return <div>Time: {seconds}</div>;
}
```

The issue is that **seconds** in the **setInterval** callback is always the initial value (0). This happens because the **useEffect** hook has an **empty** dependency array, so the setInterval callback captures the initial value of seconds.

To fix this, use a functional update:

```
setSeconds(prevSeconds => prevSeconds + 1);
```

13. How can you avoid an infinite loop with useEffect?
To avoid an infinite loop with useEffect, ensure that you manage dependencies correctly. For example, if useEffect has a dependency on a state variable that it updates, it can lead to an infinite loop. Make sure dependencies are well defined and avoid updating state within useEffect unless necessary.

14. How will the following code behave?

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

function Example() {
  const [value, setValue] = useState('initial');

  useEffect(() => {
    setValue('changed');
  }, []);

  return <div>{value}</div>;
}
```

The component will display "changed". The useEffect will run once after the initial render, setting value to "changed", which will then update the display.

15. How do you handle setting state based on previous state in functional components?
Use the functional form of the state updater function to handle state updates based on previous state values. Example:

```
setCount(prevCount => prevCount + 1);
```

16. How to resolve the common ‘Maximum Update Depth Error’?

- The "Maximum Update Depth Exceeded" error in React typically occurs when there is an infinite recursion or loop within the component's lifecycle or hooks.
- It can occur because of:
  - **Infinite Loop in useEffect:** If useEffect updates a state or prop that it depends on and that update causes useEffect to run again, this can create an infinite loop. This loop continues until the maximum call stack size is exceeded.
  - **Improper Dependency Array:** When dependencies in useEffect, useCallback or useMemo are not set correctly, it might lead to re-renders or recalculations that don't stop, causing a recursion or loop.
  - **Recursive State Updates:** State updates within a hook or a component that cause the component to re-render and trigger the same state update again can also result in an infinite loop.

Example

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

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

  useEffect(() => {
    setCount(count + 1); // This will cause an infinite loop
  }, [count]);

  return <div>{count}</div>;
}
```

Fix

```
useEffect(() => {
  // Effect logic
}, [dependency]); // Include all dependencies
```

17. How can you share state logic between components?

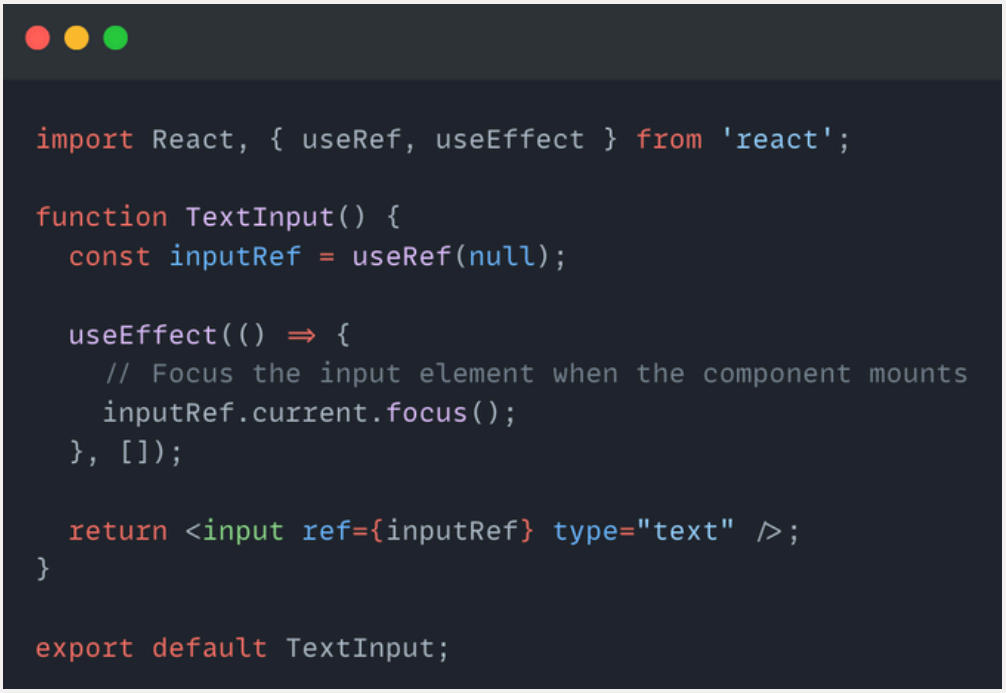
- **Lifting State Up:** When two or more components need to share the same state, you can move the state to their closest common ancestor and pass the state down as props.
- **Context API:** The Context API allows you to create a context and provide it to your component tree, making the state accessible to any component within the tree.
- **Custom Hooks:** Custom hooks allow you to encapsulate and reuse stateful logic. You can create a custom hook that manages state and provides it to multiple components.

18. State the differences between React.memo() and useMemo().

- React.memo() is a higher-order component (HOC) that we can use to wrap components that we do not want to re-render unless props within them change.
- useMemo() is a React hook that we can use to wrap functions within a component. We can use this to ensure that the values within that function are re-computed only when one of its dependencies change.

19. When would you use useRef?

- To access DOM elements and interact with them. For eg, focus an input element:



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

function TextInput() {
  const inputRef = useRef(null);

  useEffect(() => {
    // Focus the input element when the component mounts
    inputRef.current.focus();
  }, []);

  return <input ref={inputRef} type="text" />;
}

export default TextInput;
```

- To keep track of previous prop or state values:

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

function PreviousValueComponent() {
  const [count, setCount] = useState(0);
  const prevCountRef = useRef();

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

  return (
    <div>
      <p>Current count: {count}</p>
      <p>Previous count: {prevCountRef.current}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default PreviousValueComponent;
```

- To store values that need to be initialized once and persist across renders, avoiding re-initialization:

```jsx
import React, { useRef } from 'react';

function ExpensiveComponent() {
  const expensiveValueRef = useRef(computeExpensiveValue());

  return <div>Expensive Value: {expensiveValueRef.current}</div>;
}

function computeExpensiveValue() {
  // Expensive computation here
  return 42;
}

export default ExpensiveComponent;
```

20. How does the cleanup function in useEffect work?
The cleanup function runs before the component unmounts or before the effect re-runs. It is useful for cleaning up subscriptions, timers or other resources.

21. What are the benefits of useContext?

- useContext eliminates the need for a Consumer component, allowing direct access to context within functional components.
- Instead of passing props through multiple intermediary components, useContext allows direct access to the context, reducing code complexity.
- Custom Hooks can use useContext to create reusable logic that relies on context, promoting cleaner and more modular code.

22. How do functional components differ from class components?

- **State Management:** In class-based components, state is managed through the **this.state** object and updates are performed with **this.setState()**. In functional components with Hooks, state is managed with the **useState** hook, providing a simpler and more declarative way to update state.

- **Lifecycle Methods:** Class components use specific lifecycle methods, such as **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount**, to manage side effects. Hooks use **useEffect**, allowing you to define side effects with dependency arrays for precise control over when effects run.

- **Component Structure:** Class components require a more rigid structure with class syntax and render() methods. Functional components with Hooks are more concise, focusing on a functional approach without class syntax.

- **Reusability:** Custom Hooks enable you to extract common logic into reusable functions. This approach is more flexible than creating complex inheritance hierarchies or mixins with class components.

The End

Thank you for reading "React Hooks: Master Modern React". I hope this book has provided you with a clear and practical understanding of React Hooks.

By mastering these hooks, you can write more efficient and maintainable React code and be well-prepared for your next frontend developer interview.

Remember, practice is key to mastering React Hooks, so keep experimenting and building!

Happy coding!

 Connect with me on LinkedIn

 Watch me talk about tech on YouTube

 Read my blogs on Medium

 Get on a 1:1 call with me on TopMate

 Buy me a coffee here

React Hooks

Master Modern React

Why this eBook?

In "React Hooks: Master Modern React", Aishwarya demystifies React Hooks with clear, concise explanations and real-world examples.

This eBook is designed for frontend developers looking to deepen their understanding of React Hooks and ace their technical interviews.

Whether you're a beginner or an experienced developer, this guide provides the tools you need to write more efficient, maintainable React code.

What You'll Learn?

- useState
- useEffect
- useContext
- useReducer
- useMemo
- useCallback
- useRef
- Interview Questions on Hooks

AISHWARYA PARAB