

Yuko Takahashi



# React

## 100 Essential Knowledge for Beginners

—◆—

**This one book is OK!  
The Complete Guide  
to Becoming a React  
Programmer**

2024 Edition



# Index

[React Components](#)

[React Props](#)

[Understanding State in React](#)

[Virtual DOM in React](#)

[Understanding JSX in React](#)

[React Components: Class vs. Function](#)

[Understanding React Hooks](#)

[Conditional Rendering in React](#)

[Rendering Lists in React](#)

[Understanding Keys in React](#)

[React Class Component Lifecycle Methods](#)

[React Context API](#)

[Understanding Higher-order Components in React](#)

[Exploring React Router](#)

[Controlled Components in React](#)

[Uncontrolled Components in React](#)

[Using React Fragments](#)

[Using PropTypes in React](#)

[Using Refs in React](#)

[Using the Children Prop in React](#)

[Understanding React.memo](#)

[Utilizing Custom Hooks](#)

[Understanding Error Boundaries in React](#)

[Exploring Portals in React](#)

[Lazy Loading in React](#)

[Server-Side Rendering with React](#)

[Static Site Generation in Next.js](#)

[Event Handling in React](#)

[React Inline Styles](#)

[JSX Rendering Behavior](#)

[Using StrictMode in React](#)

[Understanding React DevTools](#)

[Setting Up React](#)

[Using useEffect Hook](#)

[Understanding useReducer in React](#)

[Using TypeScript with React](#)

[Dynamic Imports in React](#)

[React Reconciliation and Keys](#)

[Component Communication in React](#)

[Optimizing Performance with useCallback](#)

[Testing React Components](#)

[React Fast Refresh](#)

[Integrating Static Typing in React](#)

[Managing Global State with Redux](#)

[Managing Side Effects in Redux](#)

[Enhancing Accessibility in React](#)

[Using useMemo in React](#)

[Optimizing Bundle Sizes in React](#)

[Using Keys in List Rendering](#)

[State Management in Large Applications](#)

[Immutable Props in React](#)

[Shallow vs. Deep Copy in JavaScript](#)

[Using React Native for Mobile Development](#)

[Optimizing React for SEO](#)

[CSS-in-JS in React](#)

[Immutability in React State](#)

[Function Components with Hooks](#)

[Type Safety in React](#)

[Handling Forms in React](#)

[Enhancing Accessibility in Interactive UIs](#)

[Optimizing Render Times in React](#)

[Preventing Memory Leaks in React](#)

[Understanding Redux Toolkit](#)

[Utilizing forwardRef in React](#)

[Understanding useContext in React](#)

[Optimizing React Components](#)

[Using useLayoutEffect in React](#)

[Modular Architecture in React Projects](#)

[Error Handling in React](#)

[React Internationalization](#)

[Continuous Integration and Deployment in React](#)

[Managing Environments in React](#)

[Using GraphQL with React](#)

[Using Component Libraries in React](#)

[Clear Project Structure in React](#)

[Custom Middleware in Redux](#)

[Using Service Workers in React](#)

[React Keys in Lists](#)

[Optimizing Image Handling in React](#)

[Adopting Mobile-First Design in React](#)

[React Router Basics](#)

[React State Updates](#)

[Debouncing and Throttling in React](#)

[Preventing XSS in React](#)

[Animating in React](#)

[Data Immutability in React](#)

[Using useEffect Dependency Arrays](#)

[Compatibility with External Libraries](#)

[Understanding React's Concurrent Mode](#)

[Implementing Code Splitting in React](#)

[Testing Strategies in React](#)

[Optimizing React with Caching](#)

[Accessibility in React](#)

[Profiling React Performance](#)

[React List Virtualization](#)

[Graceful Data Handling in React](#)

[Memoization in React](#)

[CSS Organization in React](#)

[React Batched Updates](#)

[Using React Children Prop](#)

[Handling Asynchronous Operations in React](#)

[Optimizing React Bundle Size](#)

[CSS Modules in React](#)

[Functional Programming in React](#)

## Introduction



Welcome to a focused journey through the essentials of React, tailored specifically for those new to this powerful JavaScript library. Whether you're a seasoned programmer new to React or a veteran looking to refresh your knowledge on the latest features, this guide is crafted to streamline your learning process.

React is not just a tool, but a modern approach to building user interfaces with dynamic and interactive elements. This ebook distills the fundamental concepts you need to start building effective React applications. It avoids unnecessary complexities, focusing solely on the must-know principles and techniques.

As you progress through the pages, you will find the content rich in practical tips and core concepts, designed to be immediately applicable to your projects. We hope this guide proves invaluable in your development journey and assists you in quickly becoming proficient with React.

Should this resource prove helpful, we kindly ask you to leave a review or comment. Your feedback not only helps us improve, but also aids fellow engineers in similar situations discover this ebook. Let's grow and learn together in the ever-evolving world of software development.

# 1

## React Components

---

Components are the core building blocks of React applications, allowing you to divide the user interface (UI) into manageable, reusable pieces.

---

Here is an example of a simple React component that displays a greeting message:

### [Code]

```
import React from 'react';  
function Greeting() {  
  return <h1>Hello, world!</h1>;  
}  
export default Greeting;
```

### [Result]

The component would render "Hello, world!" inside an `<h1>` tag on the webpage.

In React, components are defined using JavaScript functions or classes. They return React elements describing what should appear on the screen. The Greeting function component above is simple: it returns a single `<h1>` element with the text "Hello, world!". This is an example of a functional component, which is favored for their simplicity and ease of use. Functional c



omponents can utilize hooks for managing state and other React features, which were introduced in React 16.8.

**[Trivia]**

Understanding component composition is key in React. Larger applications are built by composing many small components together, similar to building with LEGO bricks. This modularity not only makes development easier but also enhances code reusability and testing.

# 2

## React Props

---

Props are the mechanism by which components receive data from their parent, serving as read-only inputs that help configure their behavior or display.

---

Here's a simple example of a React component receiving props and using them to display data:

### [Code]

```
import React from 'react';
function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
}
export default Welcome;
```

### [Result]

If passed a prop name with the value "Alice", the component would render "Hello, Alice!" inside an `<h1>` tag.

Props (short for "properties") are how data gets passed around in a React application. Each component receives its own props object as a parameter, which can be used to read the prop

erties attached to it. Props are immutable within the component—meaning they cannot be changed by the component itself but can be replaced by new data from the parent component when it re-renders. This immutability helps prevent bugs and maintains data flow clarity across the application.

**[Trivia]**

A common React pattern is "lifting state up;" that is, sharing state data across multiple components by moving it up to their closest common ancestor. This technique allows components to remain pure (i.e., deterministic output based on props and state), which simplifies debugging and testing.

# 3

## Understanding State in React

---

In React, state refers to a component's local data storage that can be changed over time. Each component can have its own state.

---

Below is an example of a React class component using state. The component includes a button that increments a count stored in the state.

### [Code]

```
import React, { Component } from 'react';
class Counter extends Component {
  constructor(props) {
    super(props);
    // State initialization in the constructor
    this.state = { count: 0 };
  }
  incrementCount = () => {
    this.setState({ count: this.state.count + 1 });
  };
  render() {
    return (
      <div>
        <h1>Count: {this.state.count}</h1>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}
```

```
    </div>
  );
}
}
export default Counter;
```

### [Result]

Initially, the screen displays "Count: 0". Each click on the "Increment" button increases the count by 1.

State in React components is crucial for managing data that affects the render output. When `setState()` is called, React schedules an update to the component's state object and subsequently re-renders the component to reflect the new state. This method merges the object you provide into the current state, ensuring that only the components that rely on that state data re-render, which optimizes performance. The use of state enables React components to be dynamic and responsive to user interactions or other changes. This example demonstrates a simple use of state with a counter. It's important to initialize the state in the constructor of a class component, which sets up the initial state before any interaction occurs. This setup is vital for components to have accessible and modifiable state properties during their lifecycle.

### [Trivia]

React's `setState()` is asynchronous, which means it schedules changes to the component state and tells React to re-render the component and its children with updated state. This is a key concept in understanding how updates are managed in React applications.

# 4

## Virtual DOM in React

---

React uses a virtual DOM to optimize rendering by minimizing the number of updates to the actual DOM, which improves performance.

---

Below is a conceptual example explaining how React updates the real DOM using the virtual DOM.

### [Code]

```
// This is a conceptual example and not executable code
function updateComponent(virtualDOM) {
  const actualDOM = document.getElementById('app');
  const newDOM = renderToDOM(virtualDOM);
  if (newDOM !== actualDOM.innerHTML) {
    actualDOM.innerHTML = newDOM;
  }
}

function renderToDOM(virtualDOM) {
  // Simulate rendering process
  return `

# ${virtualDOM.props.title}</h1>`; } // Example of virtual DOM object const virtualDOM = { type: 'h1',


```

```
props: {  
  title: 'Hello, React!'  
}  
};  
  
// Example of how React might update the real DOM  
updateComponent(virtualDOM);
```

### **[Result]**

In a real React environment, the DOM would update to display "Hello, React!" if it's different from the current content.

React's virtual DOM is a lightweight copy of the actual DOM. It is used to test and see what changes need to be made in the real DOM. When changes occur in the component's state or props, React updates this virtual DOM first. Then, it compares the new virtual DOM with the previous snapshot of the virtual DOM. This process is called "diffing." Once React knows exactly which virtual DOM objects have changed, it updates only those parts in the real DOM, not the entire DOM. This selective update process significantly reduces the burden on the actual DOM and improves the performance of the application. This mechanism is essential for high-performance applications that need to handle complex updates and frequent re-rendering.

### **[Trivia]**

The virtual DOM not only improves performance but also adds a layer of abstraction that simplifies developer experience. React abstracts away the direct manipulation of the DOM, allowing developers to work at a higher conceptual level.

# 5

## Understanding JSX in React

---

JSX is a syntax extension for JavaScript that allows you to write HTML-like code inside JavaScript. It is often used with React to define the structure of user interfaces.

---

The following example demonstrates a simple React component using JSX.

### [Code]

```
import React from 'react';
function App() {
  return (
    <div>
      <h1>Hello, React!</h1>
      <p>Welcome to JSX.</p>
    </div>
  );
}
export default App;
```

### [Result]

The code would render a web page displaying the text "Hello, React!" in a header, followed by "Welcome to JSX." in a paragraph.



JSX allows developers to write HTML structures in the same file as JavaScript code, which simplifies the development process by avoiding the constant switching between HTML and JavaScript files. When using JSX, you can insert JavaScript expressions inside curly braces {}, which is handy for dynamic content. JSX is transformed into JavaScript calls that create React elements, which are then rendered to the DOM. To use JSX effectively: Always start component names with a capital letter. Return a single root element in the JSX expression. Use curly braces {} to integrate JavaScript expressions into JSX. Apply HTML attributes using camelCase notation, such as onClick for handling clicks.

#### **[Trivia]**

JSX is not a requirement for using React, but it is highly recommended for its readability and ease of integration with the UI logic. Babel compiles JSX into React.createElement() calls behind the scenes, which can be seen if you compile JSX in a Babel transpiler.

## Thank You for previewing this eBook

You can read the full version of this eBook in different formats:

- HTML (Free /Available to everyone)
- PDF / TXT (Available to V.I.P. members. Free Standard members can access up to 5 PDF/TXT eBooks per month each month)
- Epub & Mobipocket (Exclusive to V.I.P. members)

To download this full book, simply select the format you desire below

