React Fundamentals

A Complete 10-Chapter Course

Topics covered:

JSX & Vite Setup · Components & Props · useState · Event Handling · Conditional Rendering

Lists & Keys · Forms & Controlled Inputs · useEffect · Composition & children · Lifting State Up

Exercises: 30 hands-on component-building challenges with sample solutions

Format: A4 · Dark-theme code examples · Quick-reference tables

Table of Contents

CHAPTER 1 OF 10

What React Is, JSX, the Virtual DOM, Vite Setup

COURSE 1 · CH 1

What React Is, JSX, the Virtual DOM, and Vite Setup

Why React exists, the syntax it introduces, and getting a project running locally

Plain JavaScript can absolutely build interactive pages — the JS course covered exactly that, manipulating the DOM directly with `document.querySelector` and friends. **React** is a library that changes *how* you build that interactivity: instead of writing step-by-step instructions for changing the DOM, you describe what the UI should look like for a given set of data, and React figures out the DOM changes needed to make that true.

The Problem React Solves

As an interface grows, manually keeping the DOM in sync with your application's data gets messy fast — every place a value changes, you have to remember every DOM element that depends on it and update each one by hand. React lets you write **declarative** UI code instead: "this is what the page looks like when `count` is 5" rather than "go find the counter element and change its text to 5." When the data changes, React re-renders the description and patches only the parts of the real DOM that actually changed.

JSX — HTML-Like Syntax Inside JavaScript

```
function Greeting() {
  return <h1>Hello, world!</h1>;
}
```

That `<h1>Hello, world!</h1>` isn't a string and isn't real HTML — it's **JSX**, a syntax extension that lets you write markup directly inside JavaScript. A build tool (covered below) compiles it down to plain JavaScript function calls before it ever runs in the browser. JSX looks like HTML but follows JavaScript's rules underneath, which leads to a few differences worth knowing immediately:

```jsx
function UserCard() {
  const name = "Philip";
  return (
    <div className="card">
      <h2>{name}</h2>
      <p>Welcome back!</p>
    </div>
  );
}
```

- `className` instead of `class` — `class` is a reserved word in JavaScript, so JSX uses the DOM property name instead.
- **Curly braces** `{ }` drop back into plain JavaScript inside the markup — `{name}` embeds the variable's value.
- A component must return a **single root element** — wrap multiple sibling elements in one parent `<div>` (or a "fragment," `<>...</>`, when you don't want an extra DOM element).
- Every tag must be closed — `<img />`, not `<img>`, even for elements that are self-closing in plain HTML.

The Virtual DOM, Briefly

Each time a component's data changes, React builds a lightweight in-memory description of what the UI should look like — the **virtual DOM** — compares it against the previous version, and applies only the differences to the real, much slower-to-update browser DOM. You don't write any of this comparison logic yourself; it's the mechanism that makes the declarative style from earlier actually fast in practice. The chapters ahead focus entirely on the part you do write: describing what the UI should look like.

> **A COMPONENT IS JUST A FUNCTION**
> Every example so far is a regular JavaScript function that returns JSX. By convention, component names start with a capital letter (`Greeting`, `UserCard`) — React uses that capitalization to tell components apart from regular HTML tags like `<div>`.

Setting Up a Project with Vite

Vite is the current standard tool for starting a new React project — it sets up the JSX-compiling build step, a fast local dev server with live reload, and a production build command, all with one starting command:

```
# in a terminal, in the folder you want the project created in
npm create vite@latest my-app -- --template react

cd my-app
npm install
npm run dev
```

`npm run dev` starts a local server (usually at `http://localhost:5173`) that automatically refreshes the browser whenever a file is saved. A fresh Vite + React project includes a few starter files worth knowing immediately:

- `src/main.jsx` — the entry point; renders the top-level `<App />` component into the page's `<div id="root">`.
- `src/App.jsx` — the starter component you'll edit first; this is where the demo counter button lives by default.
- `index.html` — the one real HTML file in the whole project; everything else gets rendered into it by React.

> **CREATE REACT APP IS NO LONGER THE DEFAULT CHOICE**
>
> Older tutorials often start with `create-react-app`. It's been effectively superseded by Vite, which starts dev servers dramatically faster — new projects should use Vite unless there's a specific reason not to.

A Minimal Component, Start to Finish

```
// src/App.jsx
function App() {
  return (
    <div className="app">
      <h1>My First React App</h1>
      <p>If you can see this, JSX is rendering correctly.</p>
    </div>
  );
}

export default App;
```

`export default App;` makes this component importable from other files — `main.jsx` imports and renders it. Replacing the contents of `App.jsx` with the snippet above and saving is enough to see it appear instantly in the browser, thanks to Vite's live reload.

CONCEPT	WHAT IT MEANS HERE
JSX	HTML-like syntax that compiles to JavaScript function calls

CONCEPT	WHAT IT MEANS HERE
`Component`	A function that returns JSX describing a piece of UI
`Virtual DOM`	React's in-memory comparison step before touching the real DOM
`Vite`	The build tool/dev server used to run and bundle the project

Coding Challenges

CHALLENGE 1

Set up a new Vite + React project. Replace the contents of App.jsx with a component that renders your name in an h1 and one sentence about yourself in a p, wrapped in a single parent div.

View solution

CHALLENGE 2

Create a const variable holding your favorite number, and embed it inside a JSX expression using curly braces, alongside some surrounding text (e.g. "My favorite number is 7").

View solution

CHALLENGE 3

Write a second component function (e.g. Footer) that returns a small footer element with some text, then use it inside App's JSX alongside the existing content — App's return value should now include both pieces.

View solution

CHAPTER 1 QUICK REFERENCE

- **React** — a library for declaratively describing UI; React handles updating the real DOM
- **JSX** — HTML-like syntax inside JS; compiles to function calls, not parsed as a string
- **className**, not `class` — JSX attribute naming follows JS property names
- **{ }** drops back into plain JavaScript inside JSX markup
- A component returns exactly **one root element** (or a fragment `<>...</>`)
- **Vite** — current standard tool for starting/running a React project (`npm create vite@latest`)
- **Virtual DOM** — React's mechanism for updating only what changed, not the whole page

- **Next chapter:** components and props — passing data into a component from outside

CHAPTER 2 OF 10

Components and Props

COURSE 1 · CH 2

Components and Props

Making components reusable by passing data into them from outside

Chapter 1's `Footer` component was useful, but completely static — every instance of it renders the exact same text. **Props** (short for "properties") are how a parent component passes data into a child component, the same way a function takes arguments. This is what turns a one-off component into something genuinely reusable.

Passing Props from a Parent

```
function Greeting(props) {
  return <h2>Hello, {props.name}!</h2>;
}

function App() {
  return (
    <div>
      <Greeting name="Philip" />
      <Greeting name="Sam" />
    </div>
  );
}
```

Attributes written on a component tag — `name="Philip"` — become a single **props object** passed as the function's first parameter. `Greeting` reads `props.name` to render different text each time it's used, without needing to be rewritten. The two `<Greeting />` elements above render "Hello, Philip!" and "Hello, Sam!" from the exact same component definition.

Destructuring Props

```
// instead of writing props.name and props.age everywhere...
function UserCard({ name, age }) {
  return (
    <div>
      <h3>{name}</h3>
      <p>Age: {age}</p>
    </div>
  );
}
```

Writing `{ name, age }` directly in the parameter list **destructures** the props object, pulling out `name` and `age` as standalone variables. This is the convention used in almost all real React code — it reads more clearly than repeating `props.` in front of every value, and makes it obvious at a glance exactly which props a component expects.

Passing Different Types of Data

```
function ProductCard({ title, price, inStock }) {
  return (
    <div>
      <h3>{title}</h3>
      <p>${price}</p>
      <p>{inStock ? "In stock" : "Sold out"}</p>
    </div>
  );
}

// usage — curly braces pass real JS values, not strings
<ProductCard title="Keyboard" price={49.99} inStock={true} />
```

A string prop uses plain quotes (`title="Keyboard"`), but any other JavaScript value — numbers, booleans, arrays, objects, even other components — needs curly braces instead: `price={49.99}`, not `price="49.99"` (which would pass the string `"49.99"` rather than an actual number).

Default Prop Values

```
function Button({ label, color = "blue" }) {
  return <button style={{ background: color }}>{label}</button>;
}

// no color prop given — falls back to "blue"
<Button label="Save" />
```

Ordinary JavaScript default-parameter syntax works directly in a destructured props parameter: `color = "blue"` supplies a fallback whenever the caller doesn't pass that prop at all. Note the double curly braces in `style={{ background: color }}` — the outer pair is the JSX expression, the inner pair is a regular JS object literal for the `style` prop, which React expects as an object rather than a CSS string.

> **PROPS FLOW ONE WAY: PARENT → CHILD**
>
> A child component can read the props it's given, but it can never reach back up and change them directly — props are read-only from the child's perspective. If a child needs to cause a change in the parent (e.g. a button click updating something), the parent passes a function down as a prop, which the child calls. That pattern shows up properly once state is introduced in Chapter 3.

SYNTAX	PASSES...
`name="Sam"`	A string, "Sam"
`age={28}`	A number, 28
`active={true}`	A boolean, true
`onClick={handleClick}`	A function reference (not called yet)

Coding Challenges

CHALLENGE 1

Create a MovieCard component that destructures title and year props, rendering them as "Title (Year)". Use it three times in App with three different movies.

View solution

CHALLENGE 2

Create a Badge component that accepts a text prop and a color prop with a default value of "gray". Render it once with a custom color and once without one, confirming the default applies.

View solution

CHALLENGE 3

Create a StatusLabel component that accepts a boolean isOnline prop and renders "Online" or "Offline" using a ternary, with the text color also changing based on the same prop.

📄 View solution

CHAPTER 2 QUICK REFERENCE

- **Props** — data passed from a parent component into a child, like function arguments
- Component tag attributes (`name="Sam"`) become a single **props object**
- **Destructuring** — `function Comp({ a, b })` reads cleaner than `props.a` / `props.b`
- String props use quotes; everything else uses **curly braces** for real JS values
- **Default values** — `{ color = "blue" }`, same as default function parameters
- Props are **read-only** from the child's side — they flow one way, parent to child
- **Next chapter:** state with useState — giving a component data it can change itself

CHAPTER 3 OF 10

State with useState

COURSE 1 · CH 3

State with useState

Giving a component data it can change itself, and re-rendering automatically when it does

Props let a parent hand data down to a child, but they're read-only from the child's side. **State** is the other half of the picture — data a component owns and can change itself, in response to things like a click or a timer. Whenever a piece of state changes, React automatically re-renders that component (and re-runs the virtual DOM comparison from Chapter 1) to reflect the new value.

The useState Hook

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

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

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

`useState(0)` creates one piece of state, starting at `0`, and returns an array with exactly two things: the **current value** (`count`) and a **setter function** (`setCount`) used to change it. Calling `setCount(count + 1)` tells React "this component's state changed, please re-render it" — the component function runs again, this time with `count` holding the new value.

USESTATE IS A "HOOK" — AND HOOKS HAVE RULES

Functions starting with `use` are called **hooks** — special functions that "hook into" React features like state. They must be called directly inside a component function (or another hook), always at the top level — never inside an `if`, a loop, or a nested function. This guide will introduce several more hooks across the next few chapters; the same rule applies to all of them.

Why Not Just a Regular Variable?

```
// this does NOT work as expected
function BrokenCounter() {
  let count = 0;
  return <button onClick={() => count++}>{count}</button>;
}
```

A plain `let count = 0` would actually increment in memory each click, but React has no reason to re-render the component — nothing told it the UI needs updating, so the displayed number never changes on screen. `setCount` exists specifically to trigger that re-render; a regular variable reassignment is invisible to React.

Updating Based on the Previous Value

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

Passing a **function** to the setter (instead of a value directly) receives the most up-to-date previous value as its argument. This matters when several updates might happen close together — React guarantees each functional update sees the result of the one before it, whereas calling `setCount(count + 1)` twice in a row can both read the same stale `count` and only increment once instead of twice.

Multiple Independent State Variables

```
function ProfileForm() {
  const [name, setName] = useState("");
  const [isEditing, setIsEditing] = useState(false);

  return (
    <div>
      <p>Name: {name || "(not set)"}</p>
      <button onClick={() => setIsEditing(true)}>Edit</button>
    </div>
  );
}
```

A single component can call `useState` as many times as it needs — each call manages a completely independent piece of state, with its own value and its own setter. There's no requirement to bundle everything into one big state object the way some older patterns did; usually several small, clearly-named pieces of state read better than one combined one.

NEVER MUTATE STATE DIRECTLY

Calling something like `user.name = "New Name"` directly on a state object (instead of calling its setter with a new object) won't trigger a re-render, and risks subtle bugs even when it happens to work visually. Always treat state as **read-only** outside of its setter — for objects and arrays, that means creating a new copy with the change applied, rather than editing the existing one in place. This becomes especially important in Intermediate Chapter 3 (`useReducer`) and is worth getting into the habit of now.

PATTERN	WHEN TO USE IT
`setCount(5)`	Setting state to a known, fixed value
`setCount(count + 1)`	Simple update based on the current render's value
`setCount(prev => prev + 1)`	Safer update when multiple changes might happen in quick succession

Coding Challenges

CHALLENGE 1

Build a Counter component with +1 and -1 buttons, both updating the same count state, and a "Reset" button that sets count back to 0.

View solution

CHALLENGE 2

Build a LightSwitch component with a boolean isOn state (starting false) and a single button that toggles it, displaying "On" or "Off" based on the current value.

View solution

CHALLENGE 3

Build a component with two separate pieces of state: a numeric clickCount, and a string lastClicked storing the current time (use new Date().toLocaleTimeString()). One button updates both at once when clicked.

View solution

CHAPTER 3 QUICK REFERENCE

- **useState(initialValue)** — returns `[value, setValue]` for one piece of state

- Calling the setter triggers a **re-render** — a plain variable reassignment does not
- **Functional updates** — `setCount(prev => prev + 1)` — safer when updates happen close together
- A component can call `useState` **multiple times** for independent pieces of state
- **Never mutate state directly** — always go through its setter
- Hooks (functions starting with `use`) must be called at the top level of a component, never conditionally
- **Next chapter:** event handling — the onClick pattern used here, covered properly across more event types

CHAPTER 4 OF 10

Event Handling

COURSE 1 · CH 4

Event Handling

Responding to clicks, typing, and form submissions — the bridge between user actions and state changes

Chapter 3's `onClick={() => setCount(count + 1)}` was already an event handler, just introduced without much explanation. This chapter covers the pattern properly — naming handlers, the most common event types, reading information off the event itself, and the one JSX-specific gotcha that catches almost everyone coming from plain HTML.

Inline vs Named Handler Functions

```
function Greeting() {
  // inline — fine for short one-liners
  return <button onClick={() => alert("Hi!")}>Say hi</button>;
}

function SaveButton() {
  // named — better once there's real logic
  function handleClick() {
    console.log("Saving...");
  }

  return <button onClick={handleClick}>Save</button>;
}
```

Either style works the same way — what matters is passing a **function reference**, not the result of calling one. `onClick={handleClick}` is correct; `onClick={handleClick()}` would call `handleClick` immediately during render and pass its return value as the handler instead, which is almost never what's intended.

THE CLASSIC MISTAKE: CALLING THE FUNCTION INSTEAD OF PASSING IT

`onClick={handleClick()}` runs `handleClick` the instant the component renders — not when the button is clicked. If a handler needs to run with specific arguments, wrap it in an inline arrow function instead: `onClick={() => handleDelete(itemId)}`. The arrow function itself is the thing passed to `onClick`; it just happens to call `handleDelete` only once actually clicked.

The Event Object

```
function SearchBox() {
  function handleChange(event) {
    console.log(event.target.value);
  }

  return <input onChange={handleChange} />;
}
```

React automatically passes an **event object** as the handler's first argument, same as it would in plain JavaScript DOM event handling. `event.target` is the actual DOM element the event happened on — `event.target.value` is by far the most common thing read off it, giving the current text inside an input as the user types.

Handling Form Submission

```
function SignupForm() {
  function handleSubmit(event) {
    event.preventDefault();
    console.log("Form submitted");
  }

  return (
    <form onSubmit={handleSubmit}>
      <input type="email" />
      <button type="submit">Sign up</button>
    </form>
  );
}
```

`event.preventDefault()` stops the browser's default behavior for the event — for a form's `onSubmit`, that default is a full page reload, which would wipe out everything React has rendered. Calling it at the very start of the handler is the standard first line for almost every form submission handler you'll write.

Passing Extra Information to a Handler

```jsx
function TodoItem({ id, text, onDelete }) {
  return (
    <li>
      {text}
      <button onClick={() => onDelete(id)}>Delete</button>
    </li>
  );
}
```

Wrapping the call in an arrow function is also how a parent's function gets called *with information only the child knows* — here, `id` is a prop the child received, and `onDelete` is a function the parent passed down. The child doesn't need to know what `onDelete` does internally; it just calls it with the right ID when its own delete button is clicked. This "child calls a function the parent gave it" pattern is exactly how a child triggers a change in a parent's state, mentioned back in Chapter 2.

NAMING CONVENTION: HANDLEX

Most React code names event handler functions starting with `handle` — `handleClick`, `handleChange`, `handleSubmit` — while the prop itself (when passed down as a callback, like `onDelete` above) is named starting with `on`. Following this consistently makes it easy to tell at a glance which functions are event handlers just by their name.

PROP	FIRES WHEN...
`onClick`	An element is clicked
`onChange`	An input/textarea/select's value changes
`onSubmit`	A form is submitted (button click or Enter key)
`onMouseEnter / onMouseLeave`	The cursor enters/leaves an element
`onKeyDown`	A key is pressed while an element is focused

Coding Challenges

CHALLENGE 1

Build a component with a text input and a paragraph below it. As the user types, the paragraph should update live to show exactly what's in the input, using onChange and useState together.

View solution

CHALLENGE 2

Build a form with a single text input and a submit button. On submit, prevent the default page reload and log the entered value to the console, then clear the input.

View solution

CHALLENGE 3

Build a ColorButton component that accepts a color prop and an onSelect callback prop, calling onSelect(color) when clicked. Render three of them in App, each logging which color was picked.

View solution

CHAPTER 4 QUICK REFERENCE

- Always pass a **function reference** to an event prop — `onClick={handleClick}`, never `onClick={handleClick()}`
- Need to pass arguments? Wrap it: `onClick={() => handleDelete(id)}`
- React passes an **event object** automatically — `event.target.value` reads an input's current text
- **event.preventDefault()** — stops a form's default full-page-reload submit behavior
- Naming convention: handlers are named `handleX`, callback props are named `onX`
- A child calling a function prop is how it triggers a change up in its parent
- **Next chapter:** conditional rendering — showing different JSX based on state/props

CHAPTER 5 OF 10

Conditional Rendering

COURSE 1 · CH 5

Conditional Rendering

Showing different JSX depending on the current state or props — there's no special syntax, just JavaScript

JSX doesn't have its own `if` tag or special conditional markup — conditional rendering is just regular JavaScript deciding what to return or embed. The LightSwitch and StatusLabel components from earlier chapters were already doing this with a ternary; this chapter covers the full range of patterns and when each one reads best.

if/else — Returning Different JSX Entirely

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h2>Welcome back!</h2>;
  }
  return <h2>Please log in.</h2>;
}
```

A plain `if` statement works fine *above* the final `return`, or as separate early returns like this — it just can't be written directly inside the JSX itself, since `if` is a statement, not an expression that produces a value.

The Ternary — Inline, Within JSX

```
function Greeting({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn ? <h2>Welcome back!</h2> : <h2>Please log in.</h2>}
    </div>
  );
}
```

Because `condition ? a : b` is an **expression** — it produces a value — it can sit directly inside `{ }` in the middle of JSX, unlike `if`. This is the most common conditional pattern in React precisely because of that: it

fits inline without needing a separate return path.

&& — Rendering Something, or Nothing

```
function Inbox({ unreadCount }) {
  return (
    <div>
      <h2>Inbox</h2>
      {unreadCount > 0 && <p>You have {unreadCount} unread messages.</p>}
    </div>
  );
}
```

`condition && jsx` renders the JSX only when `condition` is true — when it's false, JavaScript's short-circuit evaluation stops before ever reaching the JSX, and nothing extra renders. This reads more cleanly than a ternary specifically for the "show this, or show nothing" case, where a full ternary's `: null` would be redundant clutter.

> **THE 0 && "TEXT" TRAP**
>
> `{unreadCount && <p>...</p>}` looks identical to the version above but has a subtle bug: when `unreadCount` is exactly `0`, `&&` evaluates to `0` itself (not `false`) — and React *does* render a bare `0` on the page, unlike `false` / `null` / `undefined`, which render nothing. Writing an explicit comparison (`unreadCount > 0 && ...`) avoids the issue entirely by guaranteeing the left side is a real boolean.

Rendering Nothing At All

```
function Banner({ message }) {
  if (!message) {
    return null;
  }
  return <div className="banner">{message}</div>;
}
```

A component returning `null` renders nothing at all into the DOM — no empty `<div>`, nothing. This is the standard way to write a component that sometimes shouldn't appear in the page at all, rather than appearing with empty content.

Several Possible States — A Mapped Lookup

```
function StatusBadge({ status }) {
  const labels = {
    pending: "⏳ Pending",
    shipped: "📦 Shipped",
    delivered: "✅ Delivered",
  };
  return <span>{labels[status] ?? "Unknown"}</span>;
}
```

Once there are more than two or three possible outcomes, a chain of `? :` ternaries gets hard to read fast. A plain object used as a lookup table — keyed by the possible values — scales much better, with `??` (the nullish coalescing operator) providing a fallback for any value not found in the table.

PATTERN	BEST FOR
`if / early return`	Entirely different JSX depending on a condition
`condition ? a : b`	Inline either/or choice within JSX
`condition && jsx`	Show something, or show nothing at all
`return null`	The whole component should render nothing this time
`Lookup object`	Three or more named possible states

Coding Challenges

CHALLENGE 1

Build a LoginButton component with isLoggedIn state (starting false). Use a ternary to show a "Log In" button when logged out and a "Log Out" button when logged in, each toggling the state when clicked.

View solution

CHALLENGE 2

Build a Cart component with an itemCount prop. Show "Your cart is empty" if itemCount is 0, otherwise show "You have N item(s) in your cart" using && correctly so 0 never renders by itself.

View solution

CHALLENGE 3

Build a TrafficLight component with a color prop ("red", "yellow", "green"). Use a lookup object to render the matching instruction text ("Stop", "Slow down", "Go") with a fallback of "Unknown signal" for any other value.

View solution

CHAPTER 5 QUICK REFERENCE

- No special JSX conditional syntax — it's all **regular JavaScript**
- **if/early return** — entirely different JSX for different cases
- **Ternary** — inline either/or, fits directly inside `{ }`
- **&&** — render something or nothing; guard with an explicit comparison to avoid the `0 && ...` trap
- **return null** — the component renders nothing into the DOM at all
- **Lookup object + ??** — cleaner than chained ternaries for 3+ named states
- **Next chapter:** lists and keys — rendering an array of data as repeated JSX

CHAPTER 6 OF 10

Lists and Keys

COURSE 1 · CH 6

Lists and Keys

Turning an array of data into repeated JSX, and why React needs a key to track each piece

Real applications display arrays of things constantly — a list of todos, search results, products in a cart. React has no special "repeat this markup" tag; instead, you use a regular JavaScript array method, `.map()`, to transform an array of data into an array of JSX elements.

Rendering an Array with .map()

```
function FruitList() {
  const fruits = ["Apple", "Banana", "Cherry"];

  return (
    <ul>
      {fruits.map((fruit) => (
        <li key={fruit}>{fruit}</li>
      ))}
    </ul>
  );
}
```

`fruits.map(fruit => <li>...</li>)` produces a new array, this time of `<li>` elements instead of strings — and React happily renders an array of elements embedded inside `{ }` exactly like it renders a single one. The curly braces around the whole `.map()` call work the same way they always have: dropping back into JavaScript inside JSX.

The key Prop — Why It's Required

Every element produced inside a `.map()` needs a **unique** `key` **prop**, as seen above. React uses keys to match each element in a new render against the same element from the previous render — without them, React can only guess by position, which causes real bugs when items are reordered, inserted, or removed (state belonging to one item can end up incorrectly attached to a different one after the list changes).

```
const todos = [
  { id: 1, text: "Buy milk" },
  { id: 2, text: "Walk the dog" },
];

function TodoList() {
  return (
    <ul>
      {todos.map((todo) => (
        <li key={todo.id}>{todo.text}</li>
      ))}
    </ul>
  );
}
```

A genuinely stable, unique identifier — like a database `id` — makes the best key. The plain string array example above used the fruit name itself only because the list had no duplicates and no id field available; a real dataset with an `id` field should always prefer it over anything else.

DON'T USE THE ARRAY INDEX AS A KEY (WHEN AVOIDABLE)

`fruits.map((fruit, index) => <li key={index}>...)` silences React's missing-key warning but doesn't actually solve the underlying problem — if the array's order changes, the index stays attached to a position rather than to the actual item, causing the exact same bugs keys exist to prevent. It's an acceptable last resort only for lists that are genuinely static and never reordered/filtered.

Rendering a Full Component Per Item

```
function TodoItem({ text }) {
  return <li>{text}</li>;
}

function TodoList() {
  return (
    <ul>
      {todos.map((todo) => (
        <TodoItem key={todo.id} text={todo.text} />
      ))}
    </ul>
  );
}
```

The same pattern extends naturally to rendering a custom component per item instead of a plain `<li>` — the `key` still goes directly on the outermost element returned from `.map()` (here, the `<TodoItem />` tag itself), never on something nested inside that component.

Filtering Before Rendering

```
function ActiveTodos({ todos }) {
  return (
    <ul>
      {todos
        .filter((todo) => !todo.done)
        .map((todo) => <li key={todo.id}>{todo.text}</li>)}
    </ul>
  );
}
```

Because `.map()` is just a regular array method, it chains naturally with `.filter()`, `.sort()`, or anything else from the array toolbox — filter down to the items that should actually be shown first, then map the remaining ones to JSX.

KEY IS INVISIBLE TO THE COMPONENT ITSELF

A component can't read its own `key` as a prop (`props.key` is always undefined) — it's reserved exclusively for React's internal bookkeeping. If a component also needs that same value for its own logic, pass it again under a different prop name, as `TodoItem`'s separate `text` prop did above.

KEY SOURCE	QUALITY
`Database id`	Best — stable and unique regardless of order changes
`A unique field (email, slug)`	Good, as long as it's genuinely unique in the dataset
`Array index`	Last resort only — breaks if the list is reordered/filtered

Coding Challenges

CHALLENGE 1

Given an array of at least 4 city name strings, render them as an unordered list using .map(), using each city name itself as the key.

View solution

CHALLENGE 2

Given an array of objects like { id, name, price }, render each one as a div showing "name — $price", using id as the key.

View solution

CHALLENGE 3

Given an array of objects like { id, name, inStock: true/false }, filter to only the in-stock items before mapping them to a list, so out-of-stock items never appear at all.

View solution

CHAPTER 6 QUICK REFERENCE

- **.map()** — the standard way to turn an array of data into an array of JSX elements
- Every mapped element needs a **unique key** prop, on the outermost returned element
- Prefer a **stable id** over the array index as a key whenever one is available
- `key` is reserved for React — a component can't read its own `props.key`
- `.filter()` chains naturally before `.map()` to exclude items from rendering entirely
- **Next chapter:** forms and controlled inputs — managing a form's fields properly with state

CHAPTER 7 OF 10

Forms and Controlled Inputs

COURSE 1 · CH 7

Forms and Controlled Inputs

Making state the single source of truth for a form's fields, instead of letting the DOM hold it

Challenge 1 from Chapter 4 already built one controlled input — an input whose displayed value comes from state, updated through `onChange`. This chapter generalizes that pattern across a whole form with several fields, plus the input types that don't use plain text (`checkbox`, `radio`, `select`).

Controlled vs Uncontrolled

```
// uncontrolled — the DOM itself holds the current value
<input type="text" />

// controlled — React state holds the current value
<input type="text" value={name} onChange={(e) => setName(e.target.value)} />
```

An input with no `value` prop is **uncontrolled** — the browser's own DOM tracks what's typed, and React only finds out by reading it later. A **controlled** input ties its `value` directly to a piece of state, with `onChange` updating that state on every keystroke — React becomes the single source of truth for what's in the field, not the DOM. Controlled inputs are the standard approach in React, since the current value is always available in state wherever it's needed (validation, submitting, displaying it elsewhere on the page) without ever having to reach into the DOM to ask.

One Handler for Several Fields

```
function SignupForm() {
  const [formData, setFormData] = useState({ name: "", email: "" });

  function handleChange(e) {
    const { name, value } = e.target;
    setFormData((prev) => ({ ...prev, [name]: value }));
  }

  return (
    <form>
      <input name="name" value={formData.name} onChange={handleChange} />
      <input name="email" value={formData.email} onChange={handleChange} />
    </form>
  );
}
```

Rather than one `useState` call per field, a form with several fields commonly uses a single state **object**, with one shared `handleChange` reading the field's own `name` attribute off `e.target` to know which key to update. `[name]: value` is a **computed property name** — plain JavaScript, not anything React-specific — letting one line update whichever key matches the input that fired the event. The `...prev` spread keeps every other field's value unchanged, updating only the one key that changed.

> **FORGETTING ...PREV LOSES THE OTHER FIELDS**
>
> `setFormData({ [name]: value })` (without spreading `prev` first) would replace the *entire* state object with just the one changed field — every other field's value would be wiped out on the very next keystroke. Always spread the previous state first when updating just one key of an object, exactly as covered for state objects generally back in Chapter 3.

Checkboxes and the checked Prop

```
function NewsletterSignup() {
  const [subscribed, setSubscribed] = useState(false);

  return (
    <label>
      <input
        type="checkbox"
        checked={subscribed}
        onChange={(e) => setSubscribed(e.target.checked)}
      />
      Subscribe to newsletter
    </label>
  );
}
```

A checkbox doesn't use `value` to control its display state — it uses `checked`, paired with `e.target.checked` (a boolean) rather than `e.target.value`. Wrapping the input in a `<label>` lets clicking the surrounding text also toggle the checkbox, a free accessibility win that costs nothing extra to add.

Select Dropdowns

```
function CountryPicker() {
  const [country, setCountry] = useState("ie");

  return (
    <select value={country} onChange={(e) => setCountry(e.target.value)}>
      <option value="ie">Ireland</option>
      <option value="hu">Hungary</option>
    </select>
  );
}
```

A `<select>` works the same as a text input — `value` and `onChange` on the `<select>` element itself (not the individual `<option>`s) — which is a small but genuine difference from plain HTML, where the `selected` attribute is normally set on the chosen `<option>` directly.

PUTTING IT TOGETHER WITH ONSUBMIT

The form's actual `onSubmit` handler (from Chapter 4) reads from the same state used to control the inputs — there's no need to separately collect values at submit time, since every keystroke has already kept state in sync. `handleSubmit` just calls `event.preventDefault()` and then does whatever needs to happen with the already-current `formData`.

ELEMENT	CONTROLLED VIA
`<input type="text">`	`value` + `e.target.value`
`<input type="checkbox">`	`checked` + `e.target.checked`
`<select>`	`value` on the select itself, same as text inputs
`<textarea>`	`value` + `e.target.value`, same as text inputs

Coding Challenges

CHALLENGE 1

Build a form with two text inputs (firstName, lastName) sharing one formData state object and one handleChange function, using the name/computed-property-name pattern. Display "Hello, [firstName] [lastName]" live below the

form.

View solution

CHALLENGE 2

Build a component with a checkbox controlled by boolean state, and a select dropdown with at least 3 options controlled by string state. Display both current values below the form.

View solution

CHALLENGE 3

Build a feedback form with a textarea (controlled) and a submit button. On submit, prevent the default behavior, log the message, and reset the textarea back to an empty string.

View solution

CHAPTER 7 QUICK REFERENCE

- **Controlled input** — `value` tied to state, `onChange` updates that state
- One shared `handleChange` + `name` attribute + computed property name (`[name]: value`) handles several fields at once
- Always **spread the previous state** (`...prev`) when updating just one key of a form-data object
- **Checkboxes** use `checked` / `e.target.checked`, not `value`
- **Select** and **textarea** are controlled the same way as text inputs
- A submit handler can read directly from state — no separate step needed to "collect" form values
- **Next chapter:** useEffect and side effects — running code in response to renders, not just events

CHAPTER 8 OF 10

useEffect and Side Effects

COURSE 1 · CH 8

useEffect and Side Effects

Running code in response to a render, rather than in response to a specific event

Everything so far has either rendered JSX or responded to a user-triggered event like a click. A **side effect** is anything that reaches outside the rendering process itself — fetching data, setting a timer, manually changing the document title, subscribing to something external. `useEffect` is the hook for running that kind of code at the right moment, tied to a component's render rather than to a click.

The Basic Shape

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

function PageTitle({ title }) {
  useEffect(() => {
    document.title = title;
  }, [title]);

  return null;
}
```

`useEffect` takes two arguments: a function containing the side effect itself, and a **dependency array** listing which values it depends on. React re-runs the effect function whenever any value in that array has changed since the last render — here, whenever `title` changes, the document's actual browser tab title updates to match.

The Three Forms of the Dependency Array

```
useEffect(() => { /* ... */ });          // no array — runs after every single render
useEffect(() => { /* ... */ }, []);        // empty array — runs once, after the first render only
useEffect(() => { /* ... */ }, [userId]);  // runs after the first render, and again whenever userId cha
```

These three shapes cover almost every use case. No array at all is rare in practice — re-running on literally every render is usually a sign something belongs in the dependency array instead. An empty array is the standard choice for "run this once when the component first appears" (fetching initial data, setting up a subscription). A populated array is for "run this again whenever *this specific thing* changes."

Fetching Data on Mount

```
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);

  useEffect(() => {
    fetch(`/api/users/${userId}`)
      .then((res) => res.json())
      .then((data) => setUser(data));
  }, [userId]);

  if (!user) return <p>Loading...</p>;
  return <h2>{user.name}</h2>;
}
```

This is the classic shape of a data-fetching effect: state to hold the result (starting as `null`, meaning "not loaded yet"), an effect that fetches and calls the setter once the data arrives, and a conditional render that shows a loading state until `user` stops being `null`. `[userId]` ensures the fetch re-runs if a different user's profile needs to be shown — without it, switching to a new `userId` prop wouldn't trigger a new fetch at all. (Intermediate Chapter 6 covers more complete error/loading handling for this exact pattern.)

Cleaning Up After an Effect

```
function Clock() {
  const [time, setTime] = useState(new Date());

  useEffect(() => {
    const id = setInterval(() => setTime(new Date()), 1000);
    return () => clearInterval(id);
  }, []);

  return <p>{time.toLocaleTimeString()}</p>;
}
```

Returning a function from inside the effect gives React a **cleanup function**, run right before the effect runs again, and also when the component is removed from the page entirely. `setInterval` here would otherwise keep ticking forever, even after `Clock` unmounts — `clearInterval(id)` in the cleanup function stops it at

exactly the right moment. Anything that "starts" something ongoing (a timer, an event listener, a subscription) almost always needs a matching cleanup function to "stop" it.

MISSING DEPENDENCIES CAUSE STALE OR INFINITE-LOOP BUGS

Leaving a value out of the dependency array that the effect actually uses means the effect can run with an outdated value, since React doesn't know to re-run it when that value changes. The opposite mistake — including *too much*, such as a value the effect itself updates — can cause the effect to re-run every single render in an infinite loop. Most editors flag this automatically via an ESLint rule (`exhaustive-deps`); when in doubt, list every state/prop value the effect function actually reads.

DEPENDENCY ARRAY	EFFECT RUNS...
`(none)`	After every render — rarely what's actually wanted
`[]`	Once, right after the first render
`[value]`	After the first render, and again whenever value changes

Coding Challenges

CHALLENGE 1

Build a component that uses useEffect with an empty dependency array to set document.title to "Welcome!" once, when the component first appears.

View solution

CHALLENGE 2

Build a Stopwatch component with a seconds count state that increases by 1 every second using setInterval inside useEffect, with a correct cleanup function that clears the interval.

View solution

CHALLENGE 3

Build a component that takes a query prop and uses useEffect with [query] as the dependency array to log "Searching for: [query]" to the console every time query changes (simulate changing it with a couple of useState-driven buttons in the parent).

View solution

CHAPTER 8 QUICK REFERENCE

- **useEffect(fn, deps)** — runs `fn` after a render, based on the dependency array
- **No array** — every render; **[]** — once on mount; **[value]** — on mount and whenever `value` changes
- Returning a function from the effect is a **cleanup function** — runs before the next effect, and on unmount
- Anything that "starts" something ongoing (timer, listener, subscription) needs a matching cleanup
- Missing dependencies → stale values; extra/wrong dependencies → possible infinite loops
- **Next chapter:** component composition and children — building flexible wrapper components

CHAPTER 9 OF 10

Component Composition and children

COURSE 1 · CH 9

Component Composition and children

Building flexible wrapper components that don't need to know what's inside them

Every component so far has rendered its own fixed JSX. **Composition** is about building components that wrap *other* JSX without needing to know what that content actually is — a generic "Card" or "Modal" that works regardless of what's placed inside it. React's mechanism for this is a special prop every component automatically receives: `children`.

The children Prop

```
function Card({ children }) {
  return <div className="card">{children}</div>;
}

// usage
<Card>
  <h2>Hello!</h2>
  <p>This content was passed in from outside.</p>
</Card>
```

Anything written *between* a component's opening and closing tags — instead of using the self-closing `/>` form seen in every example up to now — becomes available inside that component as `props.children`. `Card` doesn't know or care that it's rendering an `<h2>` and a `<p>` specifically; it just places whatever `children` turned out to be inside its own wrapper `<div>`.

Why This Matters — Reusability Without Duplication

```
<Card>
  <h2>Profile</h2>
  <p>Name: Philip</p>
</Card>

<Card>
  <img src="/photo.jpg" />
  <p>A completely different kind of content</p>
</Card>
```

The same `Card` component now wraps two entirely different pieces of content, with zero changes needed to `Card` itself. Compare this to the alternative — a `ProfileCard` and an `ImageCard`, each duplicating the same wrapper markup with only the inner content differing. Composition keeps the wrapper logic written exactly once.

Combining children with Regular Props

```
function Panel({ title, children }) {
  return (
    <div className="panel">
      <h3>{title}</h3>
      <div className="panel-body">{children}</div>
    </div>
  );
}

// usage
<Panel title="Settings">
  <p>Your preferences go here.</p>
</Panel>
```

`children` works alongside any other prop, not instead of them — `title` is a regular named prop controlling one specific, known piece of the layout (a heading), while `children` handles the arbitrary, unknown part (the body content). This combination — a few named "slot" props for the structured bits, plus `children` for everything else — covers the vast majority of real wrapper components.

Passing More Than One "Slot" of JSX

```jsx
function SplitLayout({ left, right }) {
  return (
    <div className="split">
      <div className="left">{left}</div>
      <div className="right">{right}</div>
    </div>
  );
}

// usage — JSX passed as ordinary props, not just as children
<SplitLayout
  left={<p>Sidebar content</p>}
  right={<p>Main content</p>}
/>
```

`children` only covers the single block of content between a component's tags — when a layout genuinely needs two or more independent regions, JSX can be passed as the value of *any* prop, exactly like a string or number. `left={<p>...</p>}` works because JSX is just a JavaScript value under the hood (from Chapter 1), and any JavaScript value can be passed as a prop.

COMPOSITION OVER CONFIGURATION

A common instinct coming from other tools is to add more and more props to control every detail of a component's internals (`showIcon`, `iconPosition`, `iconColor`...). Composition often leads to a cleaner result instead — give the wrapper a flexible `children` slot, and let the caller render exactly the icon/markup it wants directly inside, rather than trying to anticipate every combination through props.

PATTERN	USE IT WHEN...
`children`	One flexible region of arbitrary content
`Named JSX prop (left, header, etc.)`	Two or more independent regions in a fixed layout
`Regular prop (title, color, etc.)`	A single specific, known value — not arbitrary JSX

Coding Challenges

CHALLENGE 1

Build a Box component that renders its children inside a div with a border style, then use it three times in App wrapping three completely different pieces of content (a heading, a list, a paragraph).

View solution

CHALLENGE 2

Build an Alert component accepting a type prop ("info" or "error") and children, rendering a colored box (blue for info, red for error) containing whatever children were passed.

View solution

CHALLENGE 3

Build a TwoColumn component accepting left and right props (each containing JSX), rendering them side-by-side in two divs. Use it in App passing different JSX content to each slot.

View solution

CHAPTER 9 QUICK REFERENCE

- **children** — automatic prop holding whatever's written between a component's opening/closing tags
- Composition lets one wrapper component (Card, Panel) work with any content, written once
- `children` combines naturally with regular named props (`title`, `type`, etc.)
- JSX can be passed as the value of **any prop**, not just via children — useful for multi-region layouts
- Prefer flexible composition over an ever-growing list of configuration props
- **Next chapter:** lifting state up — sharing one piece of state between sibling components

CHAPTER 10 OF 10

Lifting State Up

COURSE 1 · CH 10

Lifting State Up

Sharing one piece of state between sibling components by moving it to their common parent

Every state example so far lived inside the one component that used it. The moment *two sibling components* need access to the same piece of data — and one of them needs to change it — that approach breaks down, since neither sibling can reach into the other's state directly. The fix is always the same: move the state up to their nearest common parent, then pass it back down to both children as props.

The Problem, Concretely

```
// these two components can't share temperature state as written —
// they're siblings, with no direct connection to each other
function TemperatureInput() {
  const [temp, setTemp] = useState("");
  // ...
}

function TemperatureDisplay() {
  // no way to read TemperatureInput's temp from here
}
```

`TemperatureInput` and `TemperatureDisplay` are siblings — neither is the other's parent or child, so neither can read or change the other's state directly. If `TemperatureDisplay` needs to show whatever value is currently typed into `TemperatureInput`, the `temp` state needs to live somewhere both of them can reach.

The Fix — Move State to the Shared Parent

```jsx
function TemperatureInput({ temp, onTempChange }) {
  return (
    <input
      value={temp}
      onChange={(e) => onTempChange(e.target.value)}
    />
  );
}

function TemperatureDisplay({ temp }) {
  return <p>Current: {temp || "--"}°</p>;
}

function TemperaturePanel() {
  const [temp, setTemp] = useState("");

  return (
    <div>
      <TemperatureInput temp={temp} onTempChange={setTemp} />
      <TemperatureDisplay temp={temp} />
    </div>
  );
}
```

`temp` now lives in `TemperaturePanel` — the closest component that's an ancestor of *both* siblings. Both children receive it as a prop, but only `TemperatureInput` is also given `setTemp` (renamed `onTempChange` for the callback-prop naming convention from Chapter 4) so it can request a change. Neither child holds its own copy of the state — there's exactly one source of truth, in the parent, and both children stay in sync with it automatically.

THIS IS THE SAME PATTERN FROM CHAPTER 4, JUST WITH TWO CHILDREN INSTEAD OF ONE

"A child calls a function prop the parent gave it" is exactly how `TodoItem`'s delete button worked back in Chapter 4 — lifting state up is that same idea applied to keeping two (or more) components synchronized with each other, rather than just notifying a parent that something happened.

Deciding Where State Should Live

A simple rule of thumb: state should live in the **lowest common ancestor** of every component that needs to read or change it — no higher, no lower. Putting it too high (e.g. in the very top-level `App`) when only two deeply nested components actually need it means passing it down through several layers of components that don't care about it at all, just to reach the ones that do.

PROP DRILLING — A REAL LIMITATION OF THIS PATTERN

Passing a prop through three, four, or more layers of components — each one just forwarding it along to its own children, never using it directly — is called **prop drilling**. It works, but becomes genuinely awkward to maintain as an app grows. Intermediate Chapter 2 introduces the **Context API** specifically to solve this, letting deeply nested components read a value directly without every component in between having to forward it manually.

SITUATION	WHERE STATE SHOULD LIVE
`Only one component uses it`	Inside that component itself
`Two sibling components need it`	Their common parent
`Many deeply nested components need it`	Common parent + Context (Intermediate Ch 2), to avoid prop drilling

Coding Challenges

CHALLENGE 1

Build a NameInput (a controlled text input) and a NameGreeting (renders "Hello, [name]!") as separate sibling components, both receiving a shared name value lifted up to a common Parent component.

View solution

CHALLENGE 2

Build two sibling Tab buttons and a content area, all controlled by one activeTab state lifted to their shared parent — clicking either button updates which tab's content the content area shows.

View solution

CHALLENGE 3

Build a ColorSwatches component (a row of clickable color buttons) and a PreviewBox component (a div whose background reflects the chosen color) as siblings, with the selectedColor state lifted to their shared parent.

View solution

CHAPTER 10 QUICK REFERENCE

- Sibling components can't read or change each other's state directly
- **Lift state up** — move it to the nearest shared ancestor, pass it (and a setter/callback) down as props
- Put state in the **lowest common ancestor** that actually needs it — no higher, no lower

- **Prop drilling** — passing a prop through several uninterested layers just to reach a deeply nested consumer
- Context (Intermediate Chapter 2) solves prop drilling for deeply nested sharing
- **That's React Fundamentals complete** — Intermediate picks up with useRef next