Props: Passing Data Down

Arguments for components: passing strings and numbers, destructuring props, the read-only rule, one-way data flow, the children prop, and prop drilling.

Introduction

In the last lesson every `CourseCard` showed the same course, because the data was baked inside it. **Props** fix that: they are the **arguments of a component**. The parent passes values in, the component reads them and renders — the same card, now personalised for every student, every course, every product. *(Roman Urdu: Props component ke arguments hain — parent data bhejta hai, component usay dikhata hai.)*

Key points

- **Passing props looks like HTML attributes:** `<ProfileCard name="Ali" age={20} />`. Text goes in quotes; **numbers, booleans, arrays, objects and expressions go in curly braces** — `age="20"` is the *string* "20", `age={20}` is the *number* 20. This is the exam point from the video.
- **The component receives one object:** React collects all props into a single object and hands it to your function — `function ProfileCard(props)` — and you read `props.name`, `props.age`.
- **Destructuring is the clean way:** `function ProfileCard({ name, age })` unpacks the object at the door (the same destructuring from the JavaScript series), and the body uses `name` and `age` directly.

  (Roman Urdu: Props object ko function ke darwaze par hi khol lo — andar saaf naam milenge.)
- **Props are READ-ONLY:** a component must never change its own props. Props belong to the parent; the child may only read them. Trying to reassign a prop is a famous beginner error — React data flows **down**, never back up through props.
- **One-way flow keeps apps sane:** parent → child → grandchild. When a value is wrong on screen, you walk up the one road it travelled.
- **`children` is a special prop:** whatever you put *between* a component's opening and closing tags arrives as `props.children` — how wrapper components like `Card` hold any content.
- **Prop drilling:** passing a value through many layers that do not need it, just to reach a deep child — the pain that lesson 5.7 (useContext) solves.

Code examples

Example 1 — One card, three students

```
function ProfileCard(props) {
  return (
    <div className="card">
      <h3>{props.name}</h3>
      <p>Age: {props.age}</p>
      <p>Course: {props.course}</p>
    </div>
  );
}

function App() {
  return (
    <div>
      <ProfileCard name="Ali" age={20} course="React" />
      <ProfileCard name="Fatima" age={22} course="JavaScript" />
      <ProfileCard name="Usman" age={19} course="CSS" />
    </div>
  );
}
```

What it does: `App` uses the same component three times, passing different props each time. Note the two kinds of values: `name` and `course` are strings in quotes, while `age` uses curly braces so it stays a real number.

What you will see: three different profile cards — Ali (20, React), Fatima (22, JavaScript) and Usman (19, CSS) — all from one component.

Example 2 — Destructuring: before and after

```
// Before: props. everywhere
function PriceTag(props) {
  return <p>Rs {props.price} — {props.label}</p>;
}

// After: unpacked at the door
function PriceTag({ price, label }) {
  return <p>Rs {price} — {label}</p>;
}
```

What it does: Both components receive the same props object. The second one destructures it in the parameter list, so the body reads like plain variables. Same behaviour, cleaner code — this is the style you

will see in every real project.

What you will see: identical output from both versions, e.g. "Rs 250 — Chai" when used as `<PriceTag price={250} label="Chai" />`.

Example 3 — The children prop: a wrapper that holds anything

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

function App() {
  return (
    <Card title="Announcement">
      <p>Class is at 10 AM tomorrow.</p>
      <button>OK</button>
    </Card>
  );
}
```

What it does: `Card` does not know or care what goes inside it. The paragraph and button written between `<Card>` and `</Card>` arrive as the `children` prop and are placed inside the card body — the pattern behind sidebars, modals and layout wrappers.

What you will see: a card titled "Announcement" containing the paragraph and the OK button, framed by the Card's own styling.

Quick recap

- **Props are a component's arguments:** the parent passes them like attributes; strings in quotes, everything else in `{ }`.
- Read them via the props object or **destructure** them in the parameters; props are **read-only** and data flows one way — down.
- `children` carries whatever sits between a component's tags; passing data through many layers is **prop drilling**, solved later by Context.

Common mistakes

1. **Passing numbers as strings** — `age="20"` sends the text "20", and `age + 1` becomes "201". The fix: numbers (and booleans, arrays, objects) always go in curly braces: `age={20}`.

2. **Trying to change props inside the child** — `props.price = 300` breaks the read-only rule and the one-way flow. The fix: the child reads props; changing data is the parent's job (with state, next lesson).
3. **Forgetting that props arrive as ONE object** — writing `function Card(name, age)` never works; React calls your function with a single props object. The fix: one parameter — the object — then destructure it: `function Card({ name, age })`.

Practice

1. Build a `ProductCard` component with destructured props `name`, `price` and `inStock`. Show the name, the price as `Rs {price}`, and the text "Available" or "Sold out" from the boolean. Use it for three different products in `App`.
2. Make a `Panel` wrapper component that takes a `title` prop and `children`, styled as a bordered box. Put two different contents inside two Panels — a short list in one, a paragraph and a button in the other.

Practice quiz

1. What is the difference between age="20" and age={20} in JSX?

- There is none
- age="20" is the string "20", age={20} is the number 20
- age="20" is the number, age={20} is the string
- Both are errors

Answer: age="20" is the string "20", age={20} is the number 20

2. May a child component change its own props?

- Yes, props are read-write
- No, props are read-only and belong to the parent; data flows down, never back up through props
- Yes, but only numbers
- Only if it uses useState

Answer: No, props are read-only and belong to the parent; data flows down, never back up through props

3. What arrives as props.children?

- The component's name
- Whatever you put between the component's opening and closing tags
- The parent's state
- Nothing, children is not a real prop

Answer: Whatever you put between the component's opening and closing tags

---

Watch the Code Scope channel: https://www.youtube.com/@CodeScopePK — this lesson's own video joins the channel when it is recorded.

Read online: https://scopetreasure.com/notes/react-props · More free tools and notes at https://scopetreasure.com