ScopeTreasure Courses

REACT COURSES · LESSON 5.16 · CODE SCOPE SERIES

Lists & Keys

Turning arrays into UI: map() builds the list, keys keep every row's identity, the index-as-key trap shown live, and filtering a list as you type.

Introduction

Almost every real screen is a list: products, messages, students, posts. In React you do not loop and push elements by hand — you `map()` **the data array into JSX**, and React builds the list. Each row also carries a **key**, React's way of knowing exactly which row is which when the list changes. *(Roman Urdu: List banana matlab array par map chalana — aur har row ko aik pehchan, yani key, dena.)*

Key points

- `map()` **is the list machine:** `items.map(item => <Row key={item.id} {...item} />)` — data in, elements out. It is the same `map` from the JavaScript series, returning JSX instead of numbers.
- **A key is a row's identity card:** a string or number, **unique among siblings** and **stable across renders** — the item's own id from the data (`item.id`) is the honest source.
- **Keys let React be surgical:** with good keys, React can insert, remove, reorder and update just the changed rows. Without them it guesses by position and gets rows' contents and inputs tangled.
  (Roman Urdu: Key har row ka CNIC hai — React usi se pehchanta hai ke kaunsi row badli, kaunsi gayi.)
- **The index-as-key trap:** `key={index}` looks harmless on a static list, but insert or delete at the top and every row's identity shifts by one — typed text and checkboxes jump to the wrong rows (demoed live in the video).
- **Keys are not props:** you cannot read `props.key` inside the child, and the same value may repeat in *different* lists — uniqueness is required only among siblings in one list.
- **Filtering is rendering the filtered array:** a search box keeps the query in state, the list shown is `items.filter(...)` computed during render — no second "visible list" state to keep in sync.
- **Empty lists are a case, not an afterthought:** `items.length === 0` shows a friendly empty message (the conditional rendering from 5.15 joining the list).

Code examples

Example 1 — map() builds the list

```
const students = [
  { id: 1, name: "Ali", course: "React" },
  { id: 2, name: "Fatima", course: "JavaScript" },
  { id: 3, name: "Usman", course: "CSS" },
];

function StudentList() {
  return (
    <ul>
      {students.map((s) => (
        <li key={s.id}>
          {s.name} — {s.course}
        </li>
      ))}
    </ul>
  );
}
```

What it does: The array of data maps into an array of `<li>` elements, each carrying its student's `id` as the key. Add a fourth student to the data and the list grows by itself — the markup never changes.

What you will see: a three-row list — "Ali — React", "Fatima — JavaScript", "Usman — CSS" — and no loop was written anywhere.

Example 2 — The index-key trap, shown honestly

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

function TrapDemo() {
  const [names, setNames] = useState(["Ali", "Fatima"]);

  return (
    <div>
      <button onClick={() => setNames(["Zara", ...names])}>Add Zara at the top</button>
      {names.map((name, index) => (
        <p key={index}>
          {name} <input placeholder="Note for this row" />
        </p>
      ))}
    </div>
  );
}
```

What it does: Each row has an input, and the key is the array index. Type a note in Ali's row, then add Zara at the top: every index shifts, React matches rows by the shifted keys, and your typed note now sits beside the wrong name. With `key={stable id}` the note would have travelled with Ali.

What you will see: notes slide one row down and attach to the wrong students after the insert — the trap, visible. This is the one demo where the bug is the lesson.

Example 3 — A searchable list: filter, then map

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

const courses = [
  { id: 1, title: "React", level: "Advanced" },
  { id: 2, title: "JavaScript", level: "Core" },
  { id: 3, title: "CSS", level: "Core" },
  { id: 4, title: "Redux", level: "Advanced" },
];

function CourseSearch() {
  const [query, setQuery] = useState("");
  const visible = courses.filter((c) =>
    c.title.toLowerCase().includes(query.toLowerCase())
  );

  return (
    <div>
      <input value={query} onChange={(e) => setQuery(e.target.value)} placeholder="Search
courses" />
      {visible.length === 0 ? (
        <p>No course matches "{query}".</p>
      ) : (
        <ul>
          {visible.map((c) => (
            <li key={c.id}>{c.title} — {c.level}</li>
          ))}
        </ul>
      )}
    </div>
  );
}
```

What it does: The only state is the query. The shown list is *derived* during render: filter the full array by the query, then map what survives. No duplicated "visible items" state exists, so the list can never disagree with the search box.

What you will see: type "re" and only React and Redux remain; type "zzz" and the friendly "No course matches" line appears instead of a blank hole.

Quick recap

- Lists are `map()` **over data**, and every row carries a **key** — unique among siblings, stable across renders, taken from the data's own id.

- `key={index}` **breaks** the moment rows are inserted, removed or reordered — state and inputs attach to the wrong rows.
- Searchable lists are derived, not stored: `filter` during render, `map` the survivors, handle the empty case.

Common mistakes

1. **Missing keys** — React warns in the console and falls back to position matching, with the same tangles as index keys. The fix: every mapped element gets a key from its data.
2. **Random keys** — `key={Math.random()}` gives every row a new identity on every render, so React destroys and rebuilds the whole list each time (inputs lose focus, state resets). The fix: keys come from the data and never change for the same item.
3. **Keys on the wrong element** — putting the key on an inner `<li>` when the map returns a wrapper component or fragment. The fix: the key goes on the **outermost** element the map callback returns.

Practice

1. Build a `MarksList`: an array of students with marks; render rows sorted as-is, each with name and marks, keyed by a student id you add to the data. Add a button that sorts a *copy* of the array highest-first into state — and confirm the rows keep their identities (keys) through the reorder.
2. Rebuild the Example 2 trap as the fixed version: give each name an object `{ id, name }`, key by id, and repeat the experiment — type notes, insert at the top, and verify every note stayed with its owner.

Practice quiz

1. What makes a good key for a mapped list row?

- The row's index, always
- A string or number unique among siblings and stable across renders, taken from the data's own id
- A random number, fresh every render
- The row's position on screen

Answer: A string or number unique among siblings and stable across renders, taken from the data's own id

2. Why does key={index} break when rows are inserted or deleted at the top?

- It does not break, it is the recommended way
- Every row's identity shifts by one, so typed text and checkboxes jump to the wrong rows
- React refuses index keys with an error
- It only breaks for long lists

Answer: Every row's identity shifts by one, so typed text and checkboxes jump to the wrong rows

3. Can a child component read its own key as props.key?

- Yes, keys are ordinary props
- No, keys are not props; you cannot read props.key inside the child
- Yes, but only for string keys
- Only in class components

Answer: No, keys are not props; you cannot read props.key inside the child

---

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-lists-keys · More free tools and notes at https://scopetreasure.com