LECTURE 27

Components, Props and Event Handling

How components talk to each other, what makes them re-render, and how React handles user interaction.

CSC336 Web Technologies

OVERVIEW

In This Lecture

1. Function components, composition, and reusability
2. Props, children, default props, and the "prop drilling" problem
3. What triggers a component to re-render
4. Handling events in React and understanding synthetic events
5. Controlled components and building forms

What Is a Function Component?

1. A JavaScript function that returns JSX describing a piece of UI
2. Component names start with a CAPITAL LETTER — this is how JSX tells a custom component apart from a plain HTML tag
3. Used in JSX just like an HTML tag, but capitalized: <Button />

A Function Component

```
function Button() {
  return <button>Click me</button>;
}

function App() {
  return (
    <div>
      <Button />
      <Button />
    </div>
  );
}
```

Composition: Small Pieces, Combined

1. Build complex UIs by combining smaller, focused components — not one giant component that does everything
2. The same idea as writing small, single-purpose functions in plain JavaScript
3. Each component should ideally do one thing well

COMPOSITION

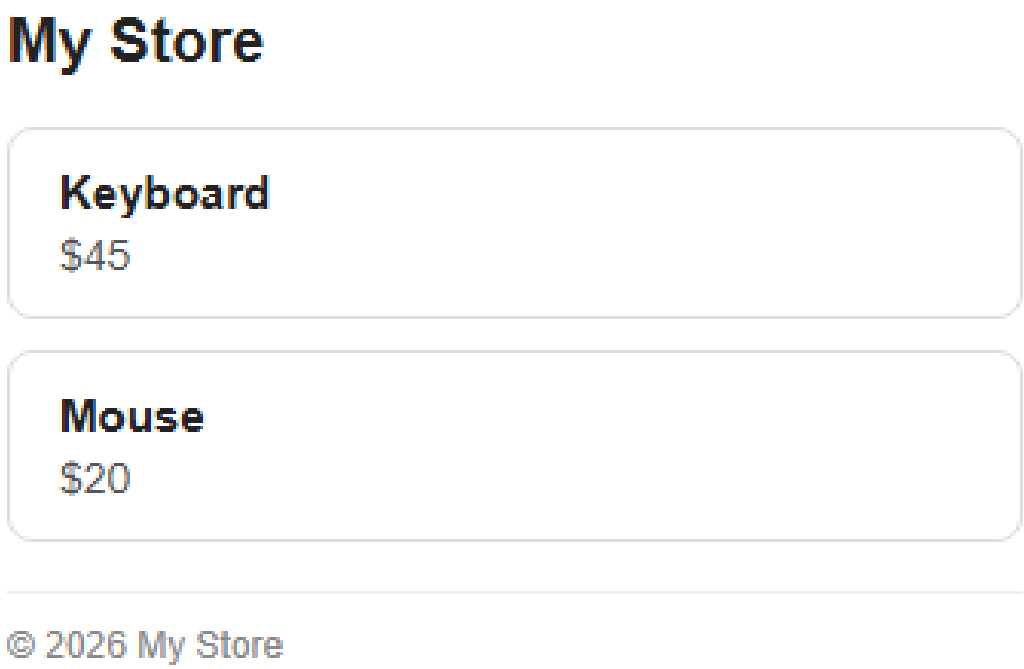
Header, ProductCard, Footer -> App

```
function ProductCard({ product }) {
  return (
    <div className="card">
      <h3>{product.name}</h3>
      <p>${product.price}</p>
    </div>
  );
}
function App() {
  return (
    <div>
      <Header />
      <ProductCard product={{ name: "Keyboard", price: 45 }} />
      <ProductCard product={{ name: "Mouse", price: 20 }} />
      <Footer />
    </div>
  );
}
```

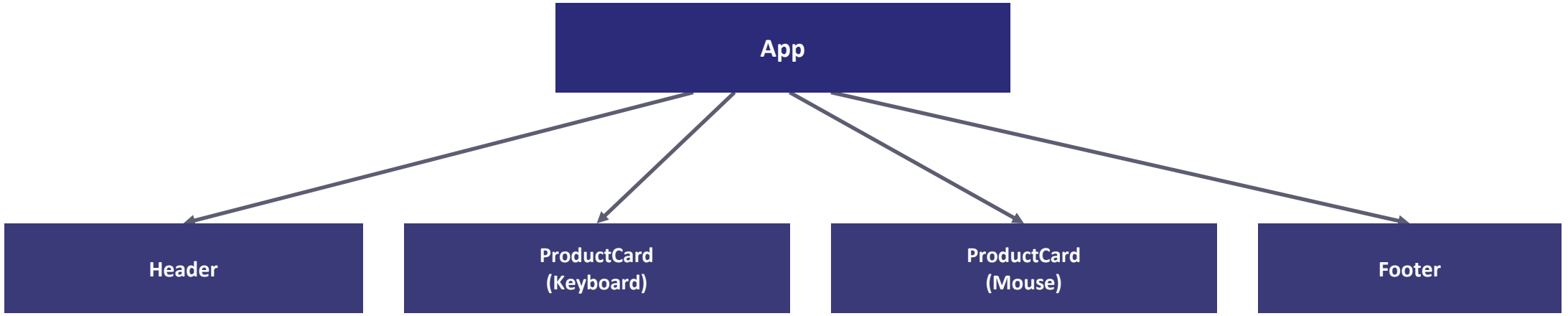
COMPOSITION

App, Composed and Rendered

App is composed of Header, two ProductCards (each fed different data), and Footer.



The Component Tree



Reusability: because ProductCard takes its data as a prop instead of hard-coding it, the same component renders any product.

Props: Passing Data Down

1. Props ("properties") let a parent component pass data down to a child
2. Set the same way you would set HTML attributes
3. Received by the component as a single object argument

Receiving and Destructuring Props

```
// Received as an object
function Greeting(props) {
  return <p>Hello, {props.name}!</p>;
}

// Destructured -- much more common
function Greeting({ name }) {
  return <p>Hello, {name}!</p>;
}

// Usage:
<Greeting name="Ayesha" />
```

Props Are Read-Only

WARNING

A component must never modify the props it receives. Props flow one way: parent to child. If a child needs to trigger a change, the parent must pass down a FUNCTION as a prop that the child calls.

Wrapping Arbitrary Content

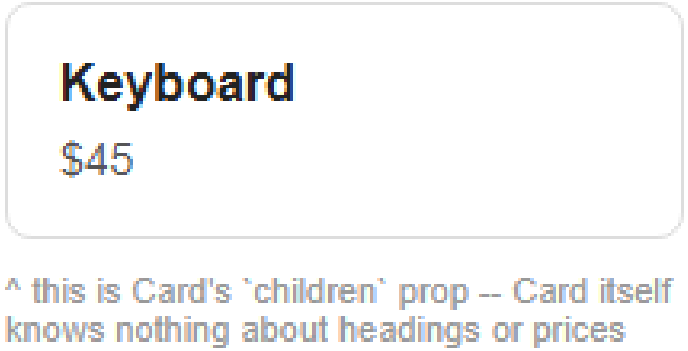
```
function Card({ children }) {
  return <div className="card">{children}</div>;
}

function App() {
  return (
    <Card>
      <h3>Keyboard</h3>
      <p>$45</p>
    </Card>
  );
}
```

THE CHILDREN PROP

children, Rendered

Every component automatically receives children: whatever was placed between its opening and closing JSX tags.



Fallback Values via Default Parameters

```
function Avatar({ src, alt = "User avatar", size = 40 }) {
  return <img src={src} alt={alt}
              width={size} height={size} />;
}

// <Avatar src="/me.png" /> -- size defaults to 40
```

The component still works sensibly if the caller forgets to pass a prop.

The Prop Drilling Problem

1. Happens when data must travel through several layers of components that don't actually use it themselves
2. App -> Page -> Sidebar -> UserBadge: only UserBadge actually reads user
3. Page and Sidebar just forward it along, purely so it can reach UserBadge

user Passed Through Two Layers That Don't Need It

```
function Page({ user }) {
  // Page doesn't use `user` itself
  return <Sidebar user={user} />;
}

function Sidebar({ user }) {
  // neither does Sidebar
  return <UserBadge user={user} />;
}

function UserBadge({ user }) {
  return <p>Logged in as {user.name}</p>;
}
```

The Fix Comes Next Lecture

NOTE

Prop drilling is exactly the problem the Context API (Lecture 28) is designed to solve -- it makes data available to a whole subtree without passing it through every level manually.

What Triggers a Re-render?

1. Its own state changes (via a state updater function -- Lecture 28)
2. Its parent re-renders, which by default re-renders all of its children too
3. The props it receives change (usually because a parent's state changed)
4. A Context value it consumes changes (Lecture 28)

Re-render Does Not Mean "DOM Changed"

NOTE

React first computes a new Virtual DOM tree and only updates the real DOM where the diff shows an actual difference (Lecture 26). Re-rendering only means React re-ran the function to see what it should currently render.

Events Look Familiar, With Differences

1. Plain HTML: onclick="handleClick()" -- a string, called immediately by the browser
2. JSX: event names are camelCase (onClick), and you pass a FUNCTION REFERENCE in curly braces
3. onClick={handleClick} runs on click; onClick={handleClick()} runs immediately while rendering

A Click Handler

```
function Button() {
  function handleClick() {
    alert("Button clicked!");
  }

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

Don't Call the Function Immediately

WARNING

Write onClick={handleClick}, NOT onClick={handleClick()}. The second form calls handleClick immediately while rendering, and passes whatever it RETURNS as the handler -- usually undefined, which does nothing on click.

Passing Arguments: Wrap in an Arrow Function

```
function ProductList({ products, onAddToCart }) {
  return (
    <ul>
      {products.map((product) => (
        <li key={product.id}>
          {product.name}
          <button onClick={
            () => onAddToCart(product.id)
          }>
            Add to cart
          </button>
        </li>
      ))}
    </ul>
  );
}
```

SyntheticEvent: A Consistent Wrapper

1. The event object your handler receives is NOT the browser's raw native event
2. React wraps it in a SyntheticEvent, giving a consistent API across all browsers
3. e.target, e.preventDefault(), e.stopPropagation() all work as expected
4. React attaches one listener at the app root and routes events internally, not one native listener per element

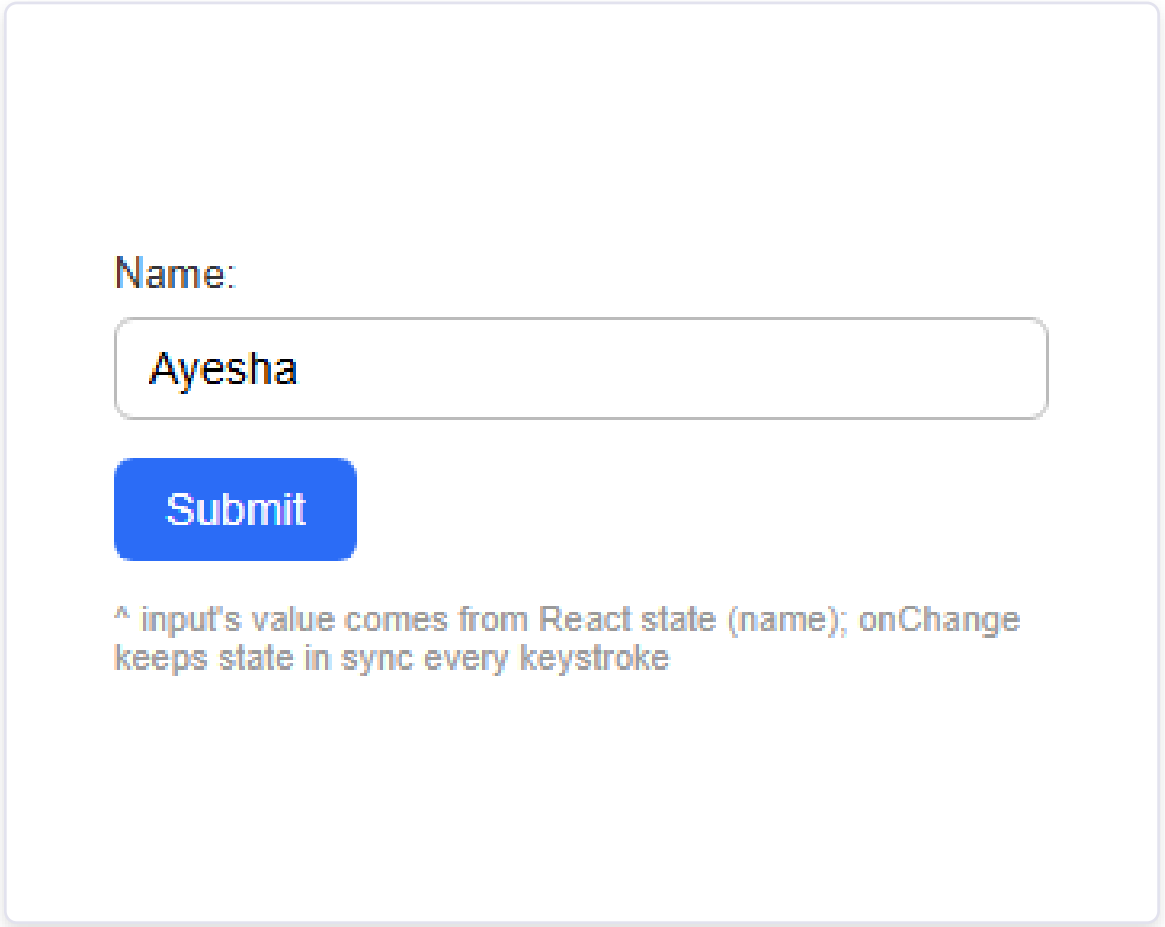
preventDefault() in Practice

```
function LinkButton() {
  function handleClick(e) {
    e.preventDefault(); // stop default navigation
    console.log("Link clicked, navigation prevented");
  }

  return (
    <a href="https://example.com" onClick={handleClick}>
      Click me
    </a>
  );
}
```

A Controlled Text Input

```
function NameForm() {
  const [name, setName] =
    useState("");
  return (
    <form>
      <label>Name:</label>
      <input
        value={name}
        onChange={(e) =>
          setName(e.target.value)}
      />
      <button type="submit">
        Submit
      </button>
    </form>
  );
}
```



What Makes an Input "Controlled"

1. Its value attribute is always set from React state -- state is the single source of truth
2. Its onChange handler updates that state on every keystroke, using e.target.value
3. State changes trigger a re-render, which recomputes value={name} -- a full, predictable loop

The Controlled-Input Loop

Every keystroke runs this full loop -- the displayed text always matches state, never the DOM's own memory of what was typed.

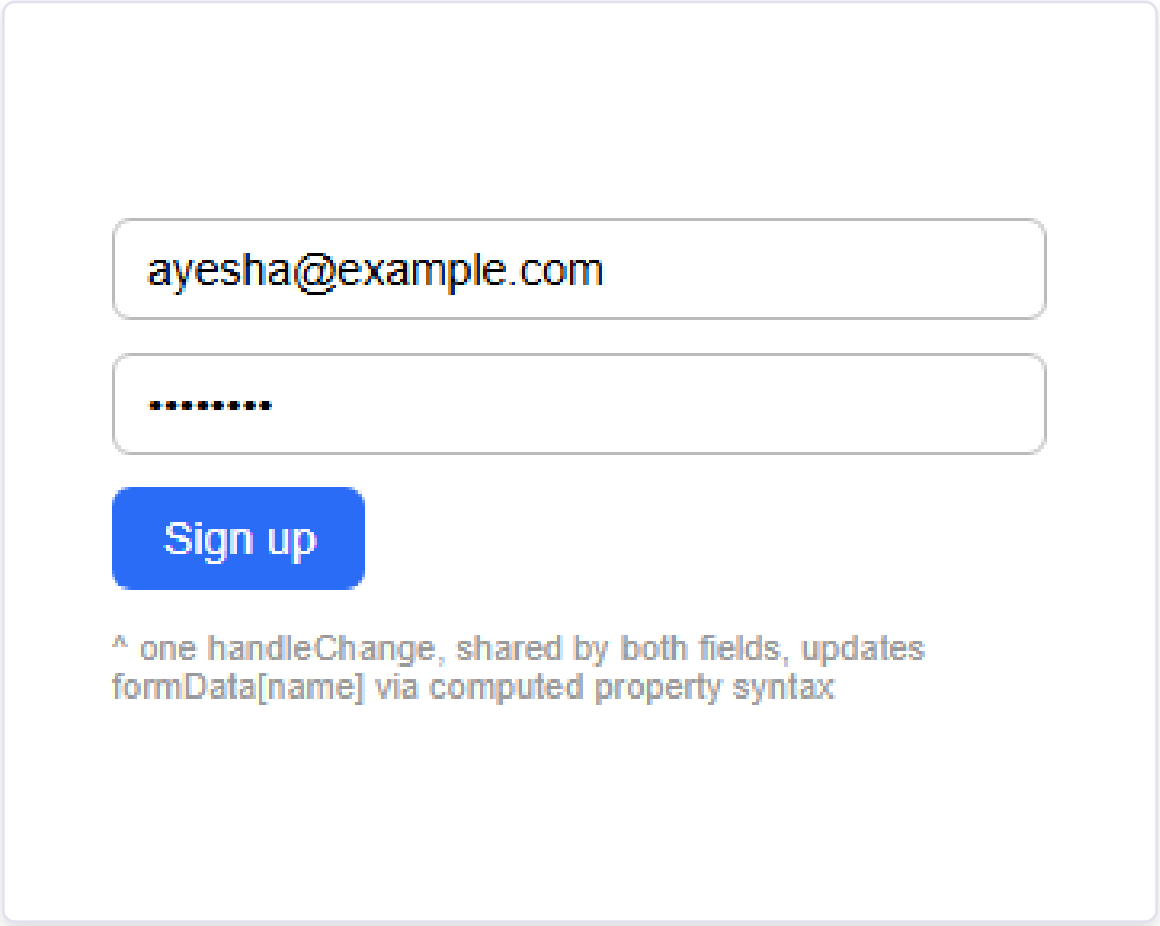
Uncontrolled Components Exist Too

TIP

React also supports uncontrolled components, where the DOM itself tracks an input's value and you read it with a ref only when needed. Controlled components are usually preferred: React state always reflects what's on screen, which simplifies validation and syncing multiple fields.

One Handler, Multiple Fields

```
function SignupForm() {
  const [data, setData] =
    useState({ email: "", password: "" });
  function onEdit(e) {
    const { name, value } = e.target;
    setData({ ...data, [name]: value });
  }
  return (
    <form>
      <input name="email"
        onChange={onEdit} />
      <input name="password"
        onChange={onEdit} />
      <button>Sign up</button>
    </form>
  );
}
```



KEY TAKEAWAYS

Lecture 27 in Six Points

1. Function components are just functions returning JSX; composition builds complex UIs from small, focused, reusable pieces.
2. Props pass data one-way from parent to child and must never be mutated; children lets a component wrap arbitrary nested content.
3. Prop drilling is the pain of passing data through components that don't need it -- solved by the Context API next lecture.
4. A component re-renders when its own state changes, its parent re-renders, or its props/consumed context change.
5. React events use camelCase names and SyntheticEvent objects; always pass a function reference to a handler, never a call.
6. Controlled components keep form values in React state via value and onChange, making state the single source of truth.