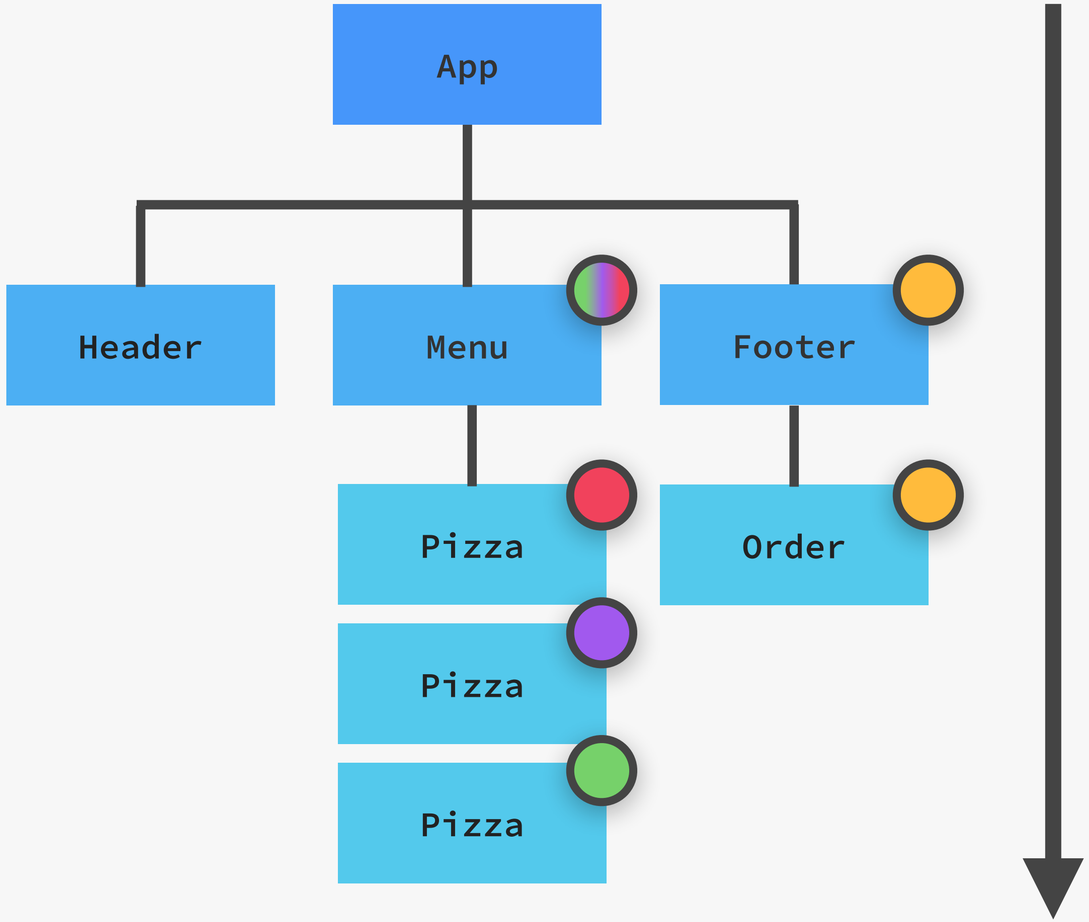
REVIEWING **PROPS**

PROPS

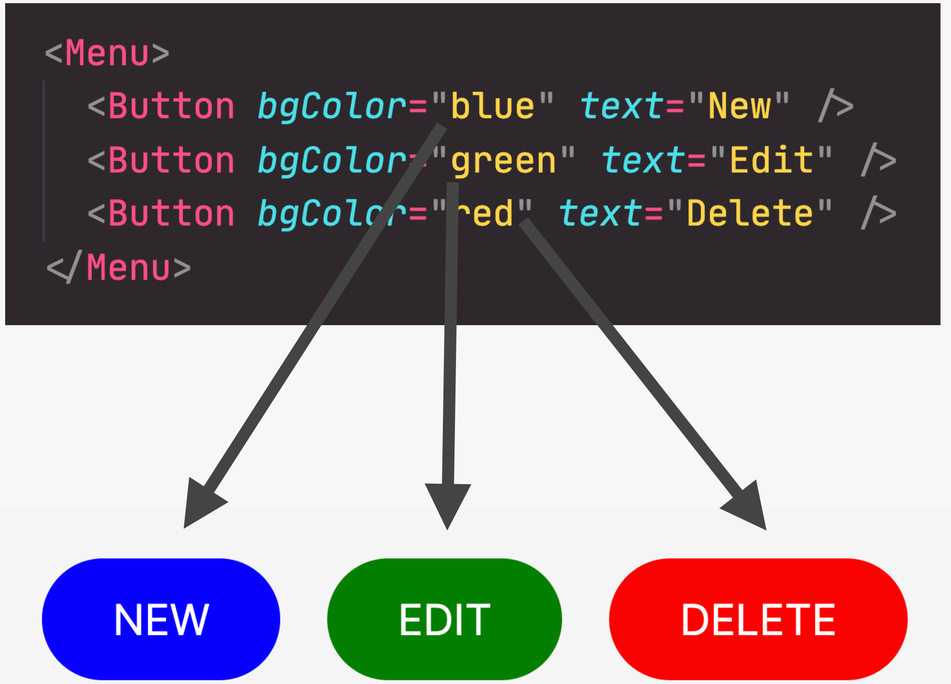
👉 Props are used to pass data from **parent components** to **child components (down** the component tree)



REVIEWING **PROPS**

PROPS

- 👉 Props are used to pass data from **parent components** to **child components** (**down** the component tree)
- 👉 Essential tool to **configure** and **customize** components (like function parameters)
- 👉 With props, parent components **control** how child components look and work



REVIEWING **PROPS**

PROPS

- 👉 Props are used to pass data from **parent components** to **child components** (**down** the component tree)
- 👉 Essential tool to **configure** and **customize** components (like function parameters)
- 👉 With props, parent components **control** how child components look and work
- 👉 **Anything** can be passed as props: single values, arrays, objects, functions, even other components

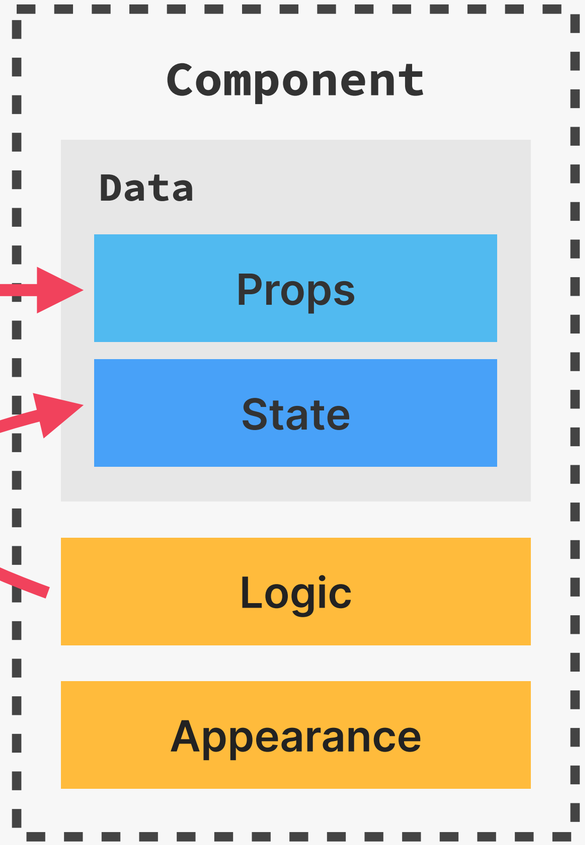
```jsx
function CourseRating() {
  const [rating, setRating] = useState(0);

  return (
    <Rating
      text="Course rating"
      currentRating={rating}
      numOptions={3}
      options={["Terrible", "Okay", "Amazing"]}
      allRatings={{ num: 2390, avg: 4.8 }}
      setRating={setRating}
      component={Star}
    />
  );
}

function Star() {
  // To do
}
```

PROPS ARE **READ-ONLY!**

Props is data coming from the **outside**, and can **only** be updated by the **parent component**

Parent Component

State is **internal** data that can be updated by the **component's** logic




```
let x = 7;

function Component(){
  x = 23;
  return <h1>Number {x}</h1>
}
```

Don't do this!

☝️ Props are read-only, they are **immutable**! This is one of React's strict rules.

☝️ If you need to mutate props, you actually **need state**

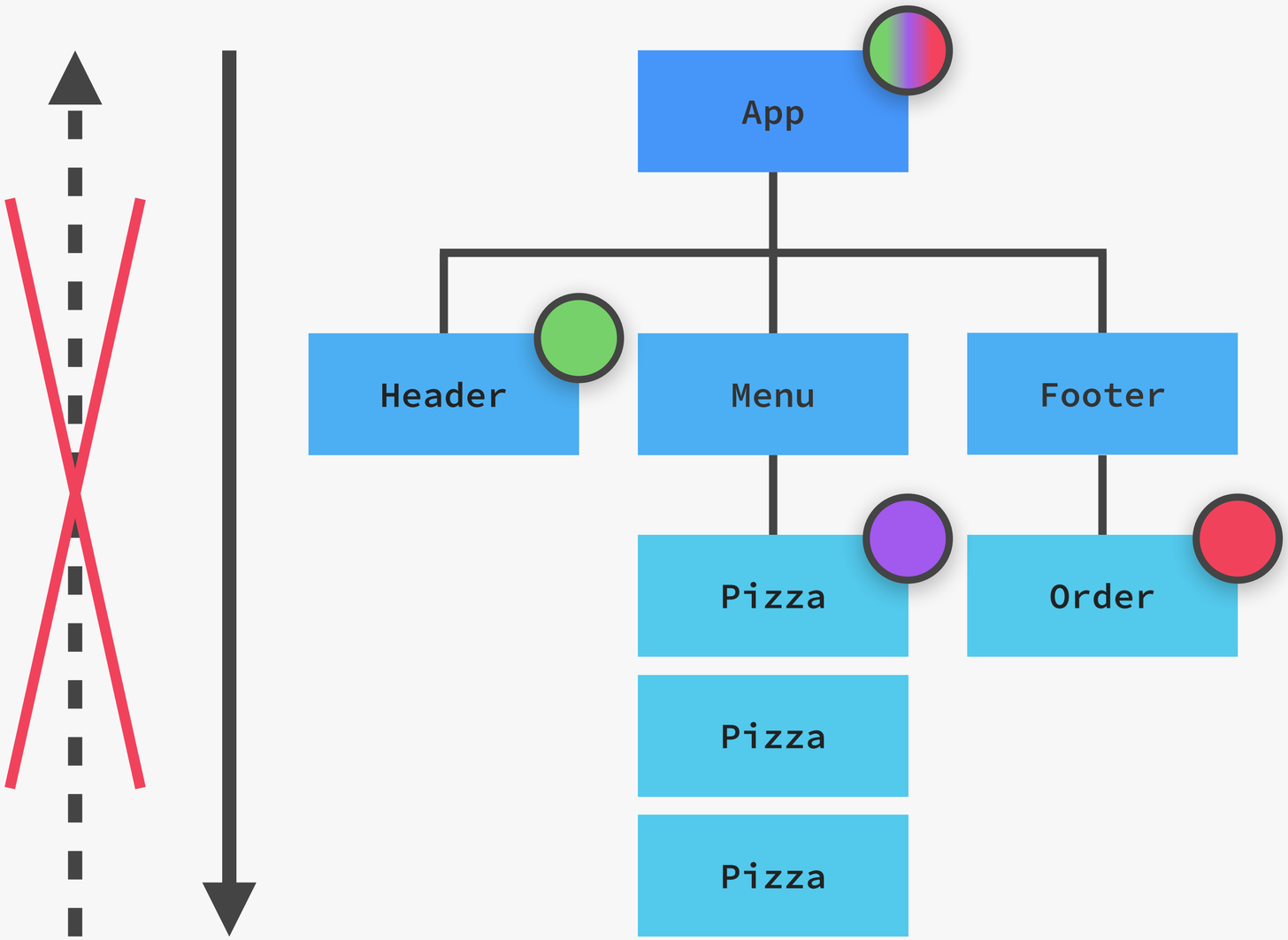
⬇️ WHY?

👉 Mutating props would affect parent, creating **side effects** (not pure)

👉 Components have to be **pure functions** in terms of props and state

👉 This allows React to optimize apps, avoid bugs, make apps predictable

ONE-WAY DATA FLOW



ONE-WAY DATA FLOW...

- 👍 ... makes applications more predictable and easier to understand
- 👍 ... makes applications easier to debug, as we have more control over the data
- 👍 ... is more performant

*Angular has **two-way** data flow*

RULES OF JSX

GENERAL JSX RULES

- 👉 JSX works essentially like HTML, but we can enter "**JavaScript mode**" by using `{}` (for text or attributes)
- 👉 We can place **JavaScript expressions** inside `{}`. Examples: reference variables, create arrays or objects, `[].map()`, ternary operator
- 👉 Statements are **not allowed** (`if/else`, `for`, `switch`)
- 👉 JSX produces a **JavaScript expression**

==

```
const el = <h1>Hello React!</h1>;
```

```
const el = React.createElement("h1", null, "Hello React!");
```

  1. We can place **other pieces of JSX** inside `{}`
  2. We can write JSX **anywhere** inside a component (in `if/else`, assign to variables, pass it into functions)

- 👉 A piece of JSX can only have **one root element**. If you need more, use `<React.Fragment>` (or the short `<>`)

DIFFERENCES BETWEEN JSX AND HTML

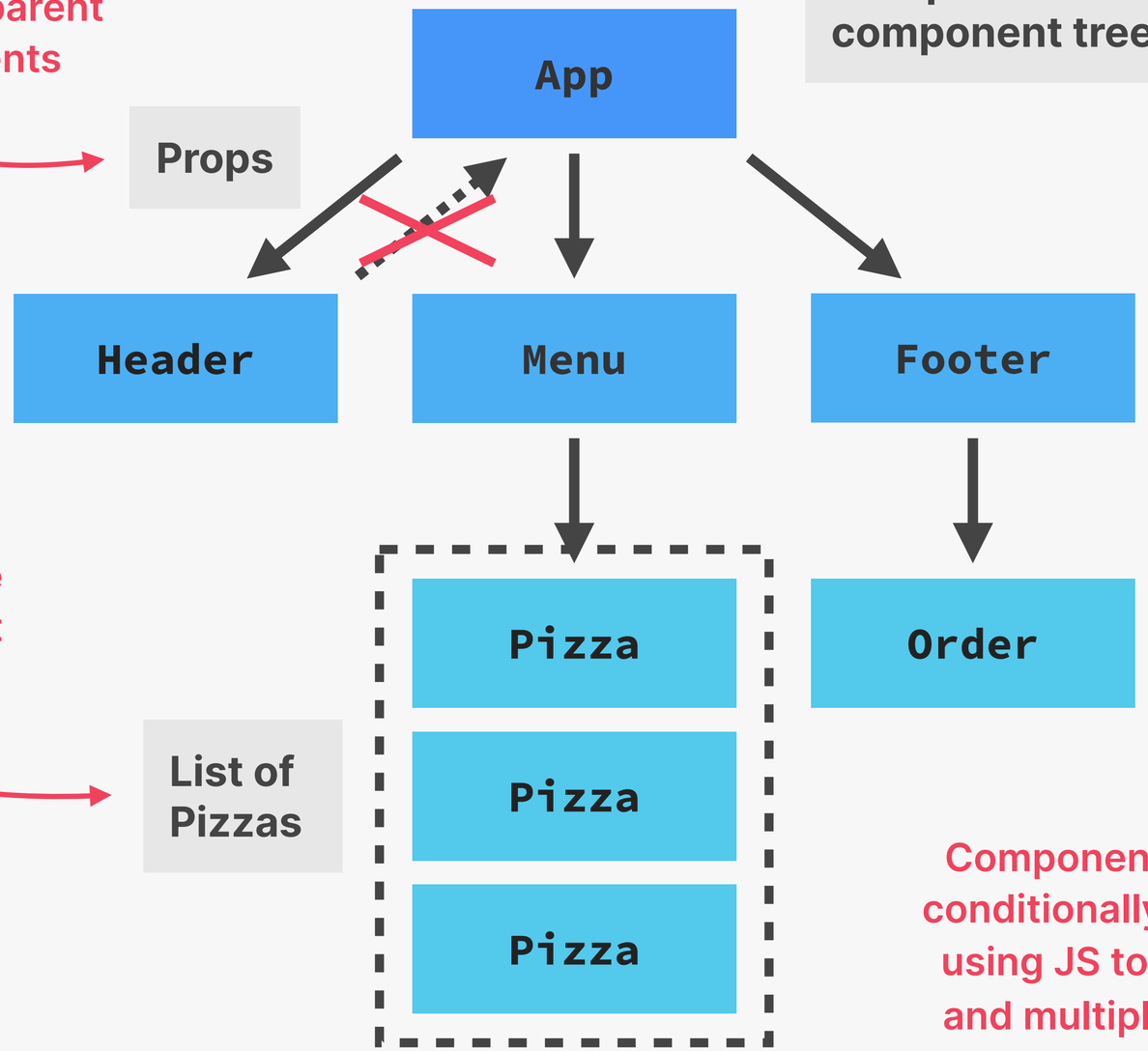
- 👉 `className` instead of HTML's `class`
- 👉 `htmlFor` instead of HTML's `for`
- 👉 Every tag needs to be **closed**. Examples: `<img />` or `<br />`
- 👉 All event handlers and other properties need to be **camelCased**. Examples: `onClick` or `onMouseOver`
- 👉 **Exception**: `aria-*` and `data-*` are written with dashes like in HTML
- 👉 CSS inline styles are written like this: `{{<style>}}` (to reference a variable, and then an object)
- 👉 CSS property names are also **camelCased**
- 👉 Comments need to be in `{}` (because they are JS)

🎯 SECTION **SUMMARY**

To pass data from parent to child components

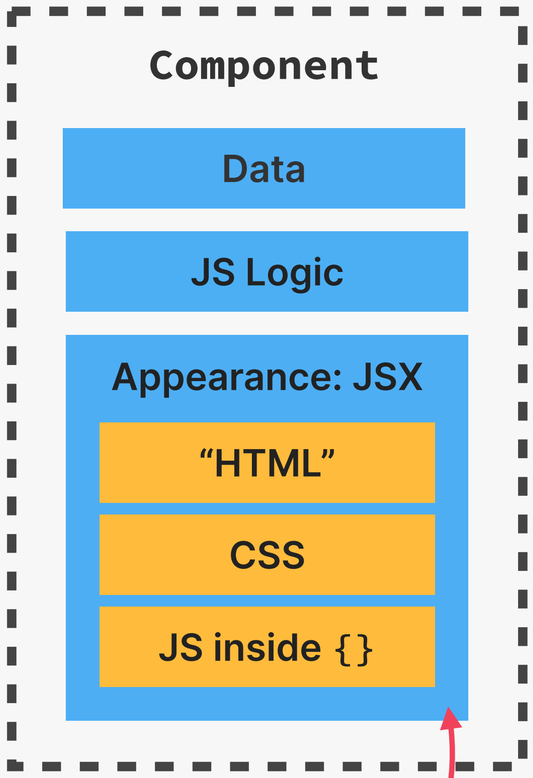
Props

Components in component tree



Rendering multiple components at once using the JavaScript `.map()` method

List of Pizzas



Components can be conditionally rendered using JS tools: `&&`, `?`, and multiple `return`

JSX block is what we return from a component