STATE AND PROPS

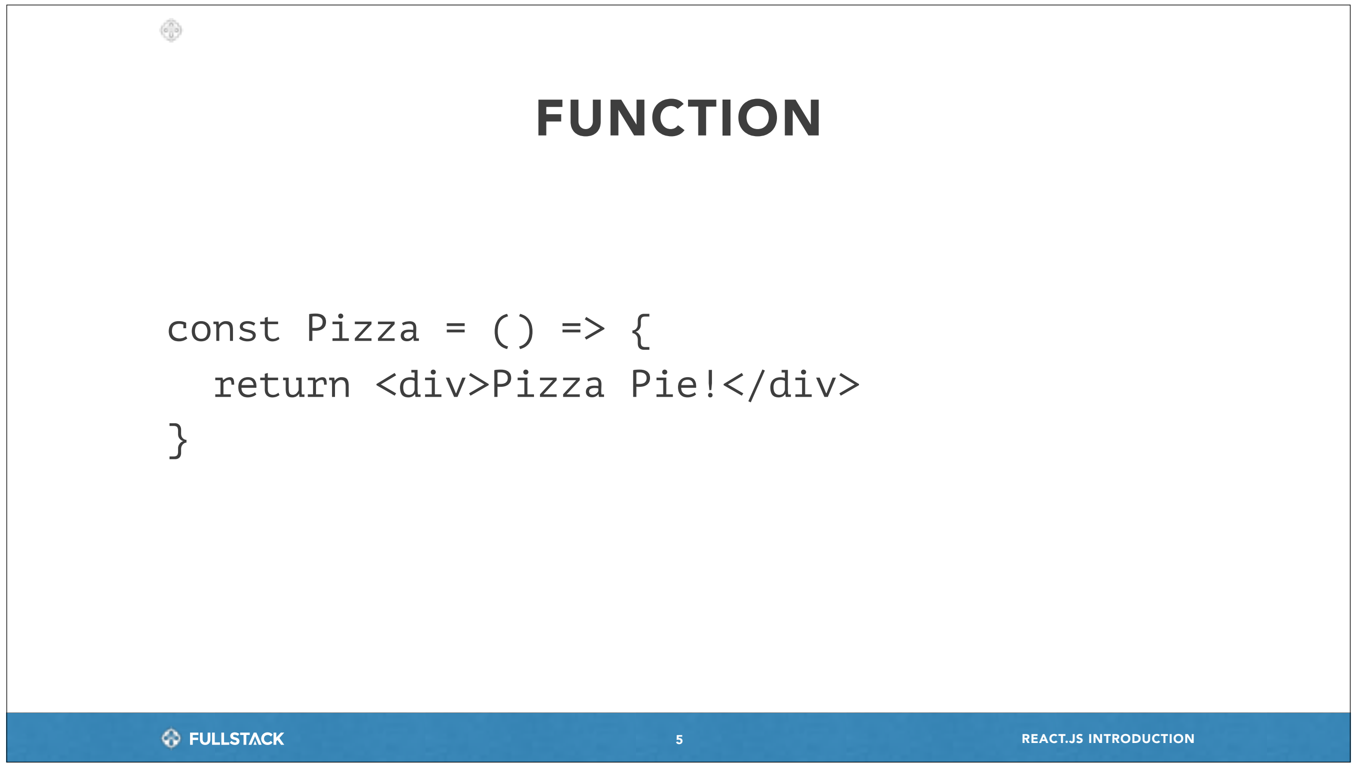
TRAJECTORY

- Reusing components with props
- Unidirectional data flow via props
- Class components vs. stateless functional components

TWO WAYS TO WRITE A COMPONENT

CLASS

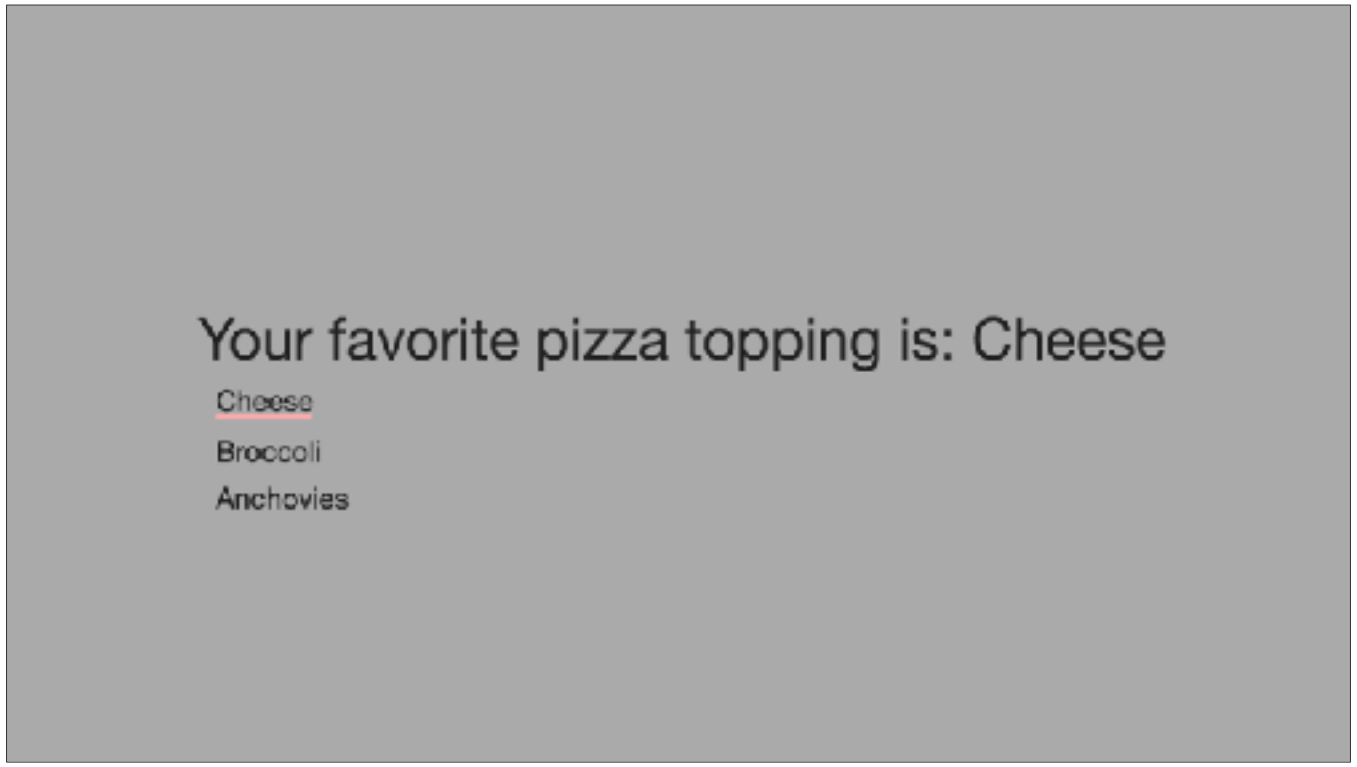
```
class Pizza extends React.Component {
  render () {
    return <div>Pizza Pie!</div>
  }
}
```



FUNCTION

```
const Pizza = () => {
  return <div>Pizza Pie!</div>
}
```

Right now, these two functions do the exact same thing. We'll explore the difference between the two in depth later, but right now just know that you can write a component as a class or as a function

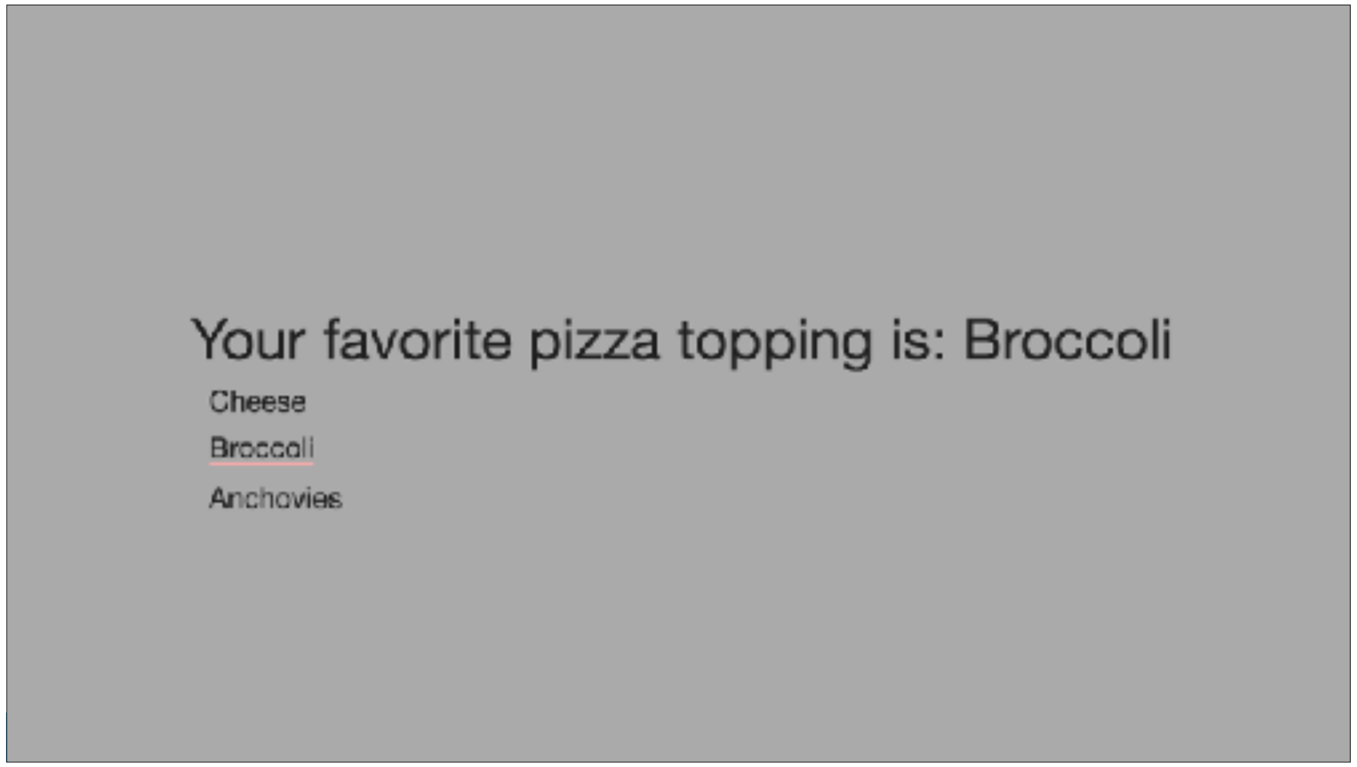

Your favorite pizza topping is: Cheese

Cheese

Broccoli

Anchovies

Our example in the following slides will use this example app. When we click on one of these toppings, the name of our favorite topping will change, and we'll add a border underneath the topping we clicked.

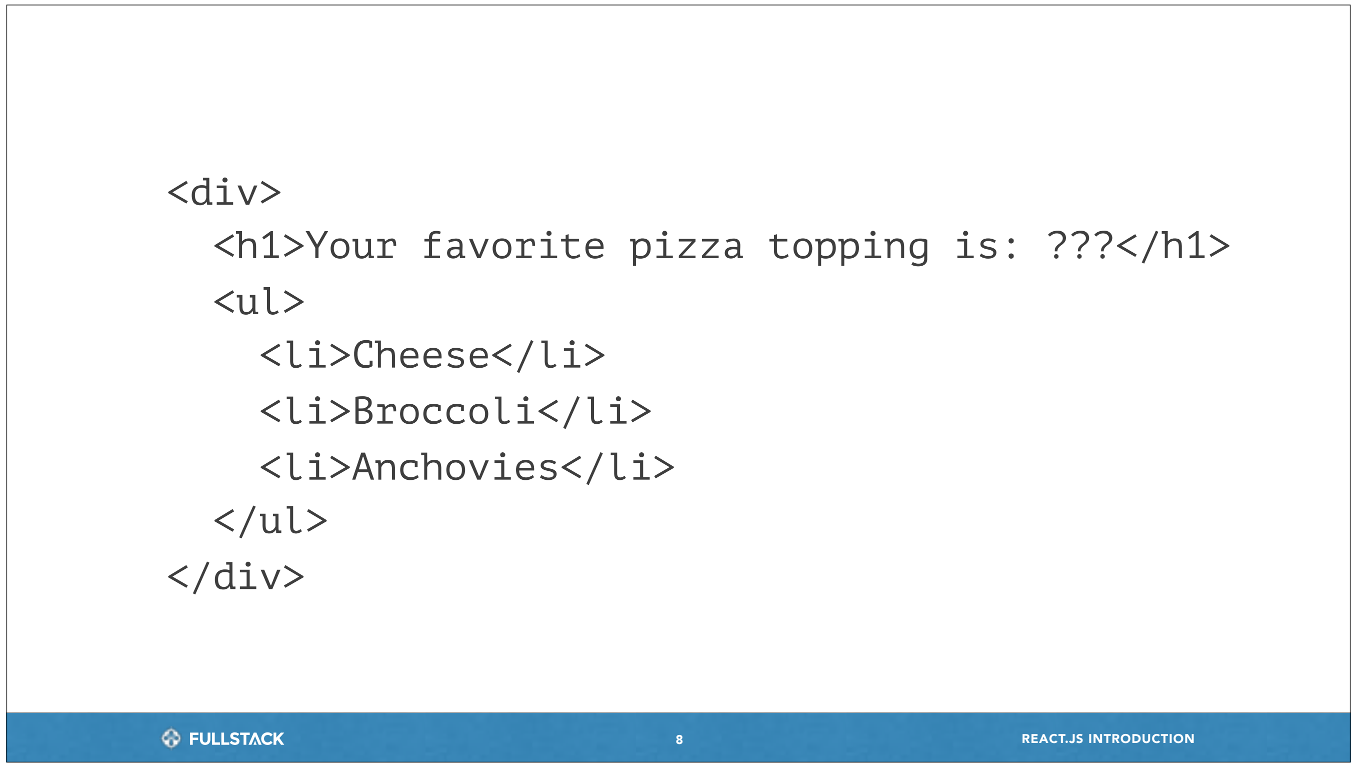


Your favorite pizza topping is: Broccoli

Cheese

Broccoli

Anchovies



```
<div>
  <h1>Your favorite pizza topping is: ???</h1>
  <ul>
    <li>Cheese</li>
    <li>Broccoli</li>
    <li>Anchovies</li>
  </ul>
</div>
```

Let’s start with a view composed of HTML.

```
<ToppingList>
   {/* ingredients go here… */}
</ToppingList>
```

```
<ToppingList>
  <Cheese />
  <Broccoli />
  <Anchovies />
</ToppingList>
```

```
const Cheese = () => {
```

```
  return <li>Anchovies</li>
}
```

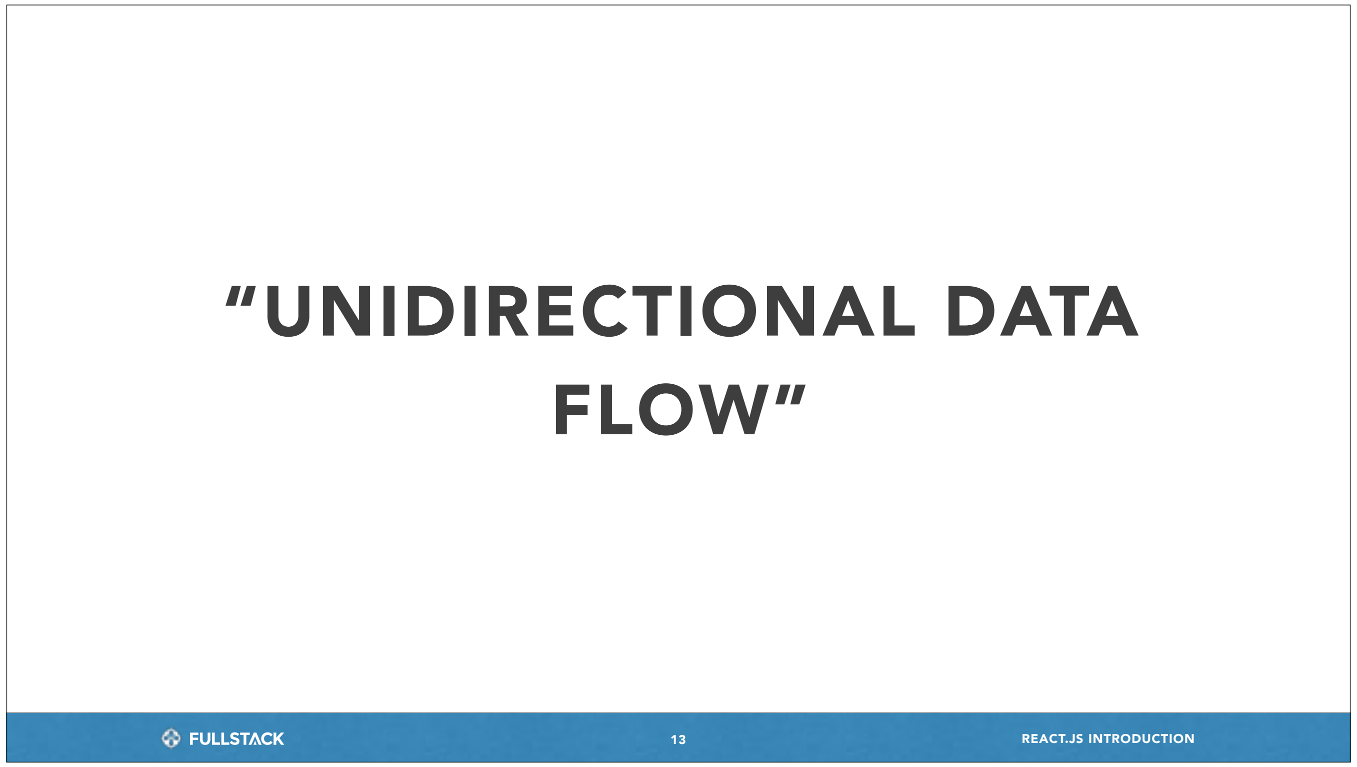
```
const Topping = (props) => {
  return <div>{props.type}</div>
}
```

```
<ToppingList>
  <Topping type="cheese" />
  <Topping type="broccoli" />
  <Topping type="anchovies" />
</ToppingList>
```

PROPS

- Conceptually and syntactically very similar to an HTML *property*

- All props that are passed into a component become key-value pairs on that component's "props" object

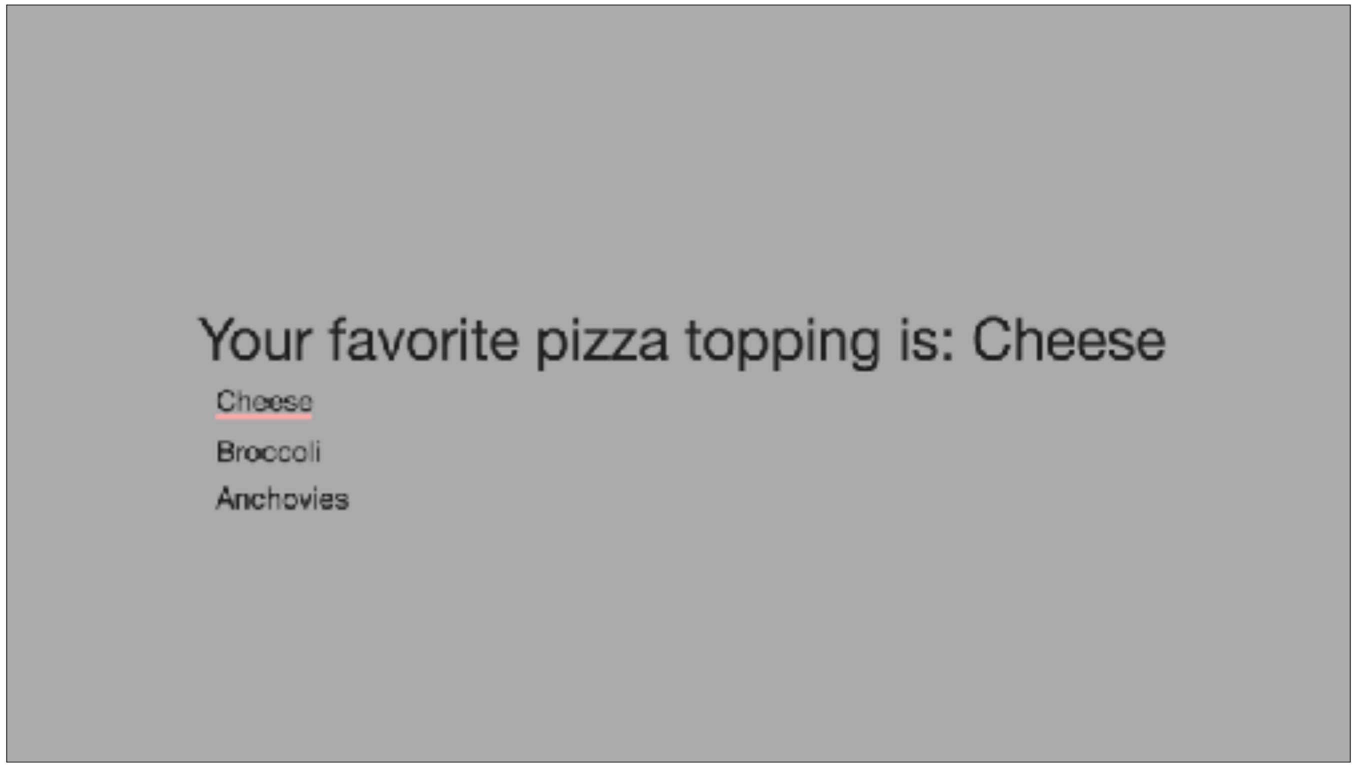


"UNIDIRECTIONAL DATA FLOW"

What does this mean?

UNIDIRECTIONAL DATA FLOW

- **We view our UI as a hierarchy of components**
  - Which is intuitive - we already think of HTML this way
- **The big difference: our state is also communicated via that hierarchy**
- **Means of communication: passing down props to components**



Your favorite pizza topping is: Cheese

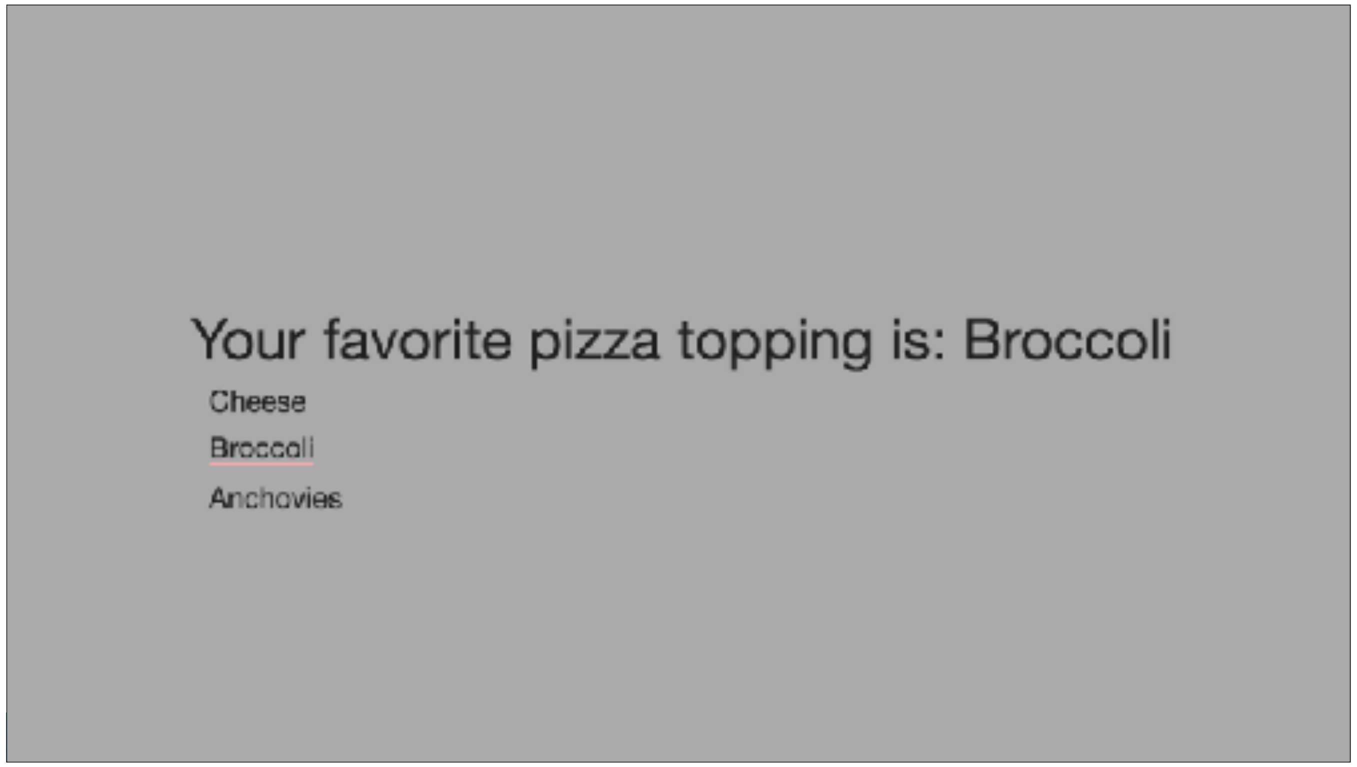
Cheese

Broccoli

Anchovies

Let’s look at this example again and think about state. How could we represent state?

We might have a string called “selectedTopping” or something like that. And this state is used by the ToppingList component itself to display the favorite topping, and it’s also used by child Topping components

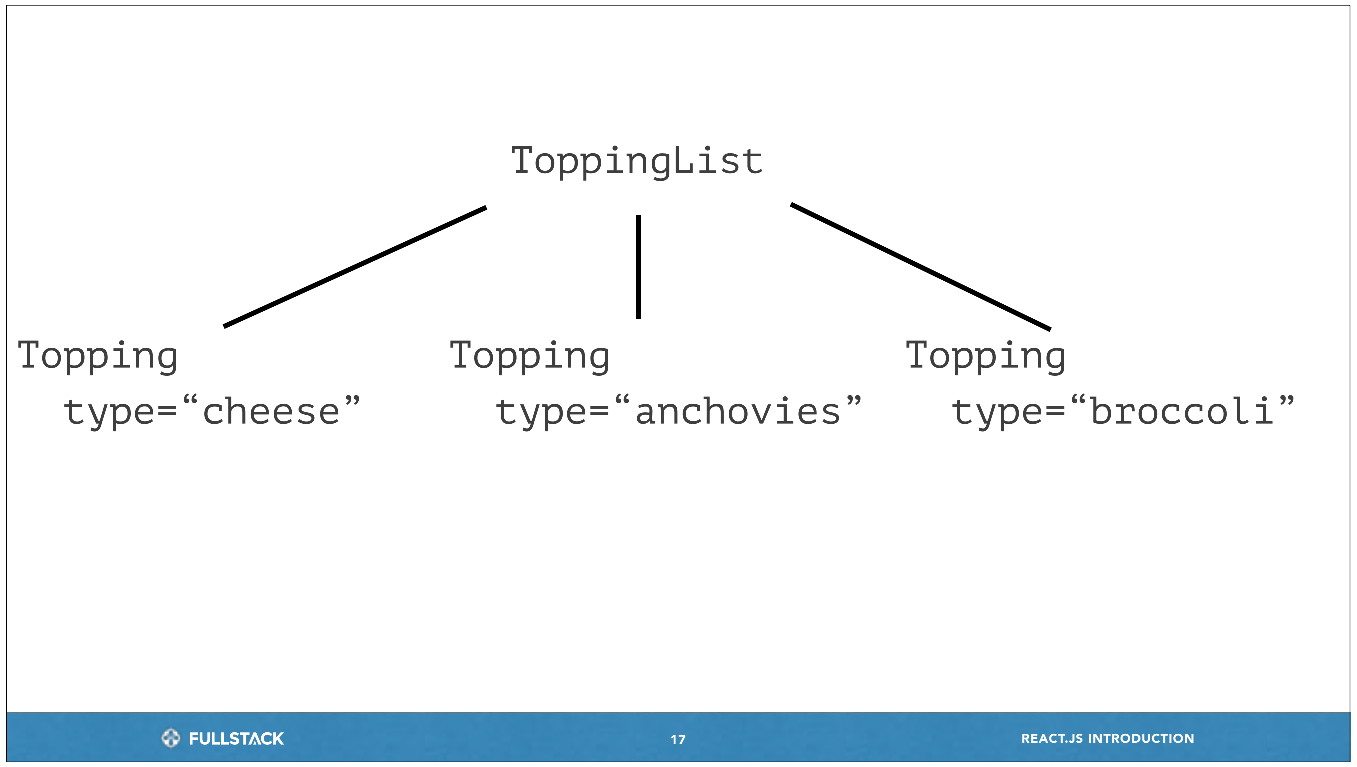


Your favorite pizza topping is: Broccoli

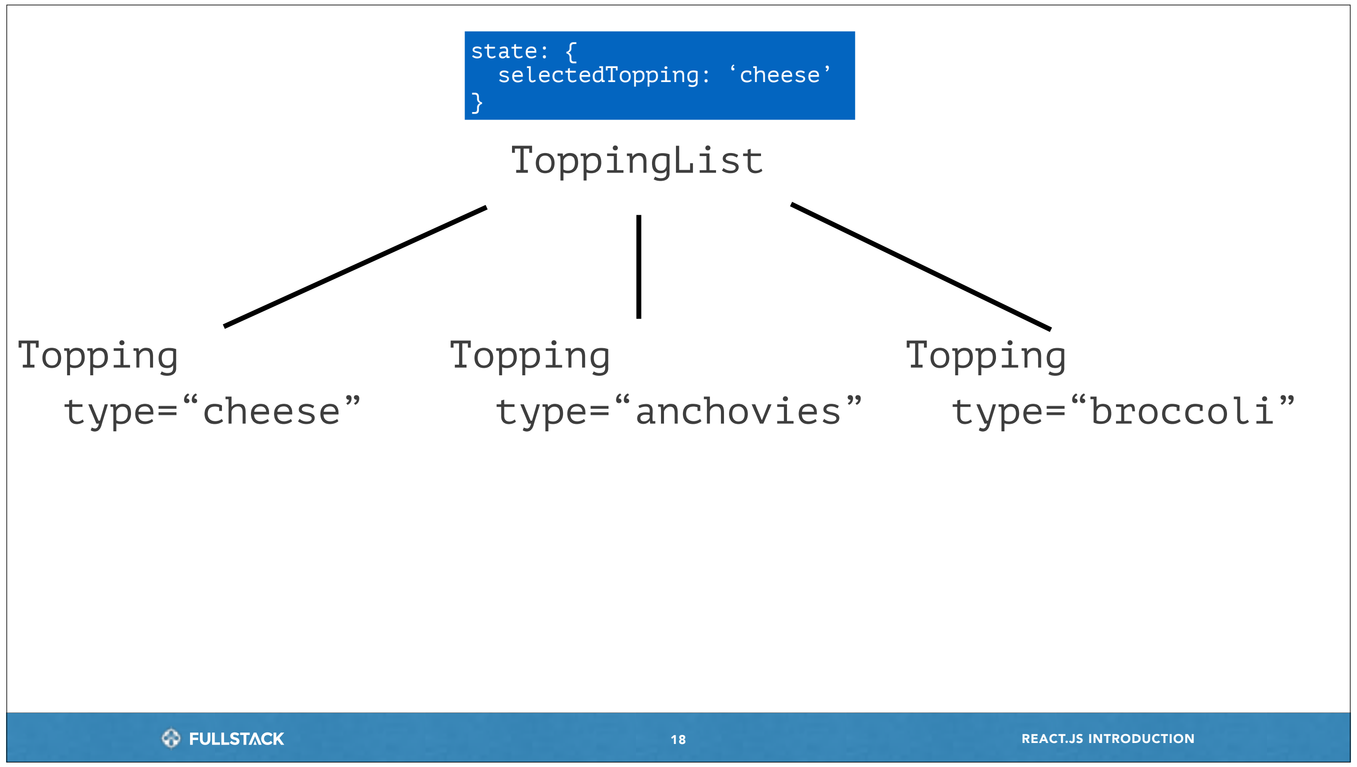
Cheese

Broccoli

Anchovies

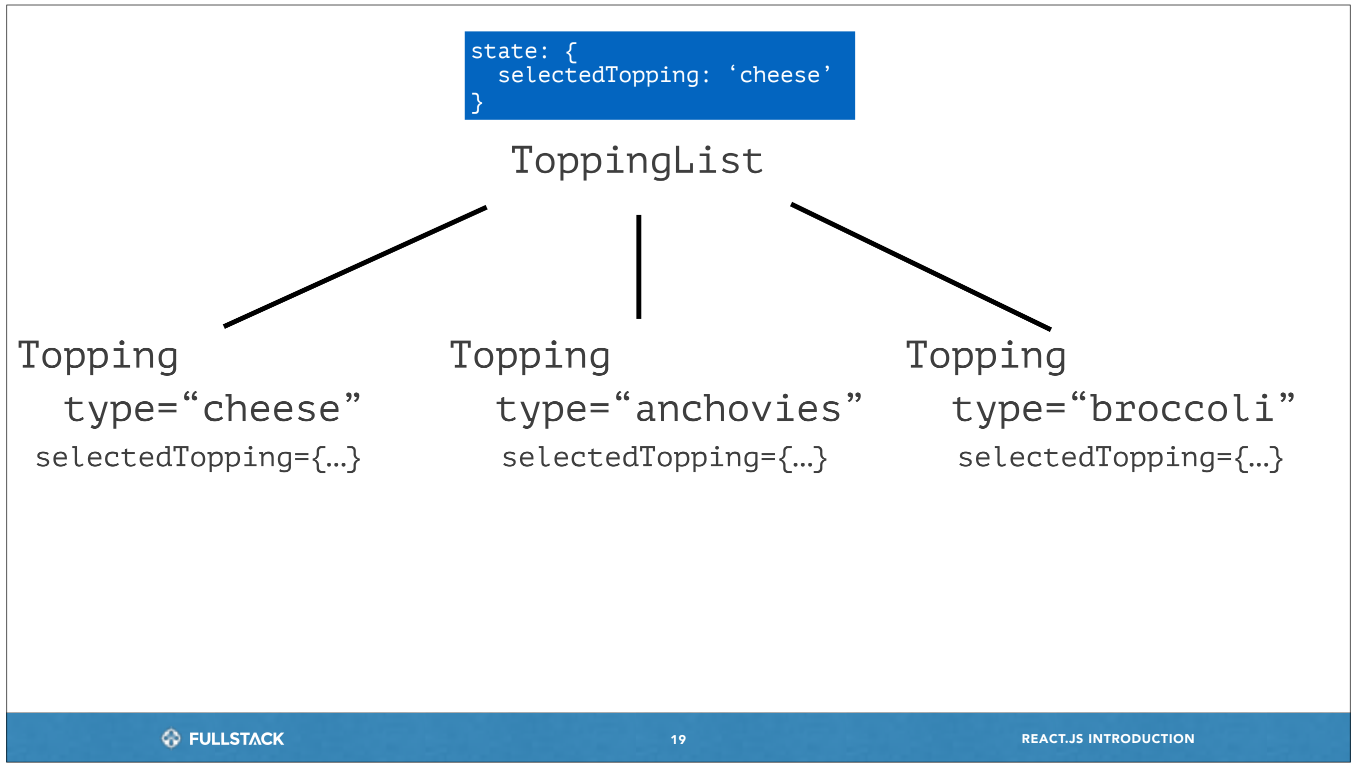


Looking at this hierarchy, where should the state in our app live? Should it live in both the ToppingList and the Ingredients?


```
state: {
  selectedTopping: 'cheese'
}
```

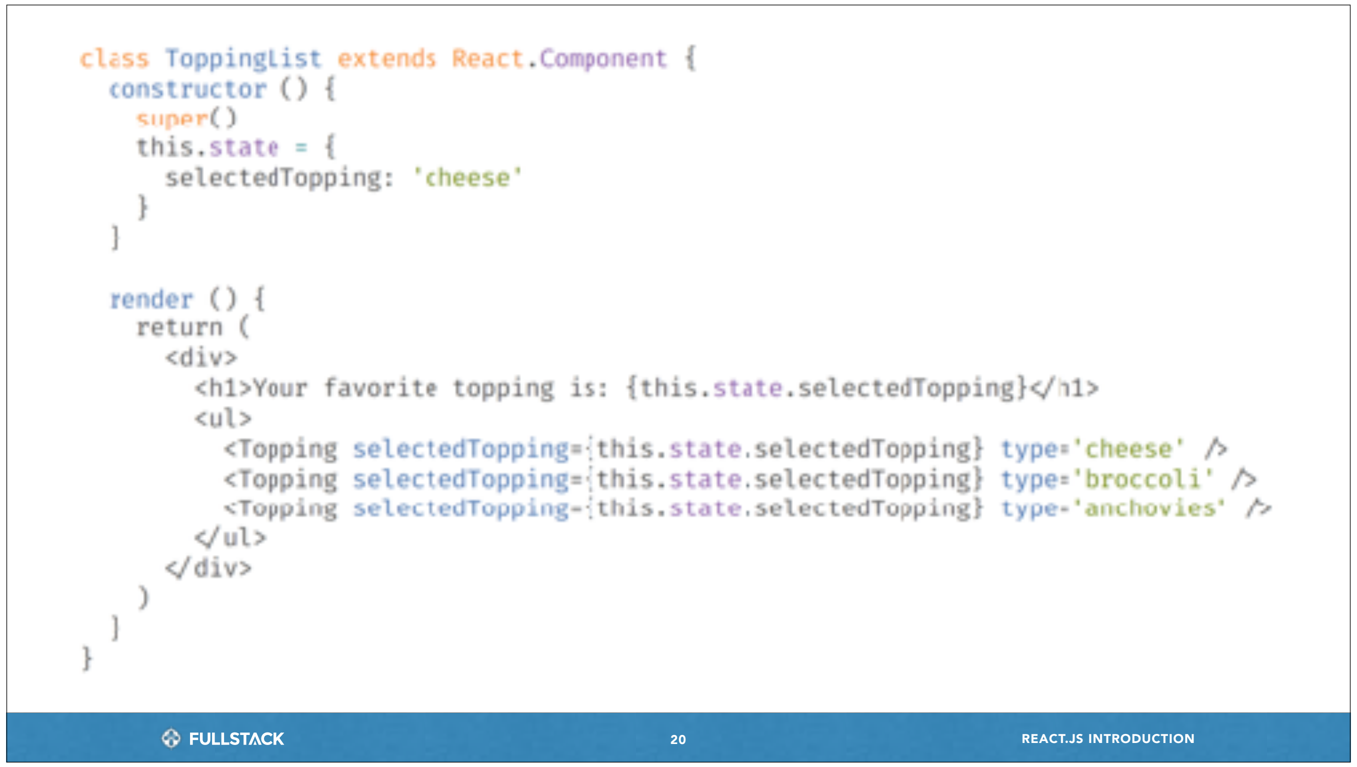
It should be in the ToppingList. If we were to make the individual topping components have state, then which would be the real selectedTopping? We would have multiple sources of truth, which React is structured to prevent.

So instead, state lives in the hierarchy as far up the tree as necessary. We know



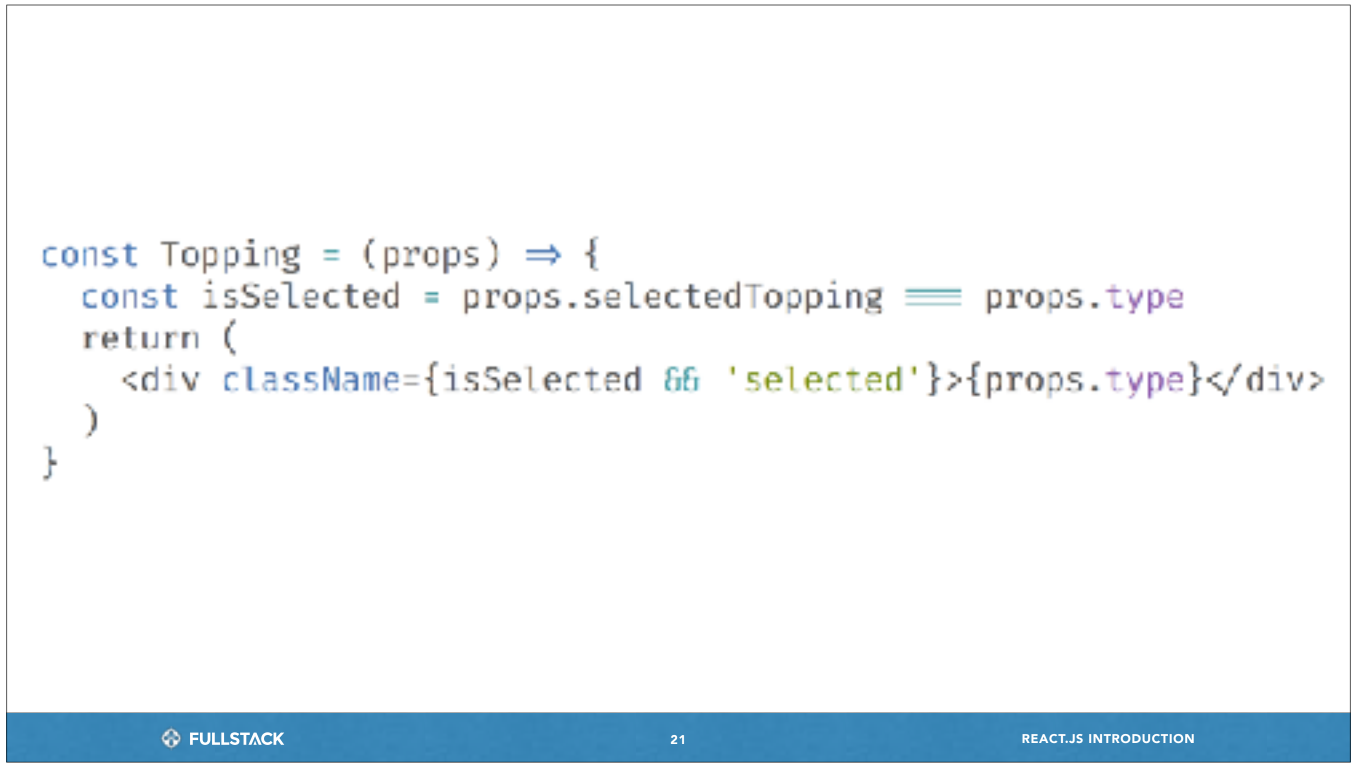
It should be in the ToppingList. If we were to make the individual topping components have state, then which would be the real selectedTopping? We would have multiple sources of truth, which React is structured to prevent.

So instead, state lives in the hierarchy as far up the tree as necessary. We know



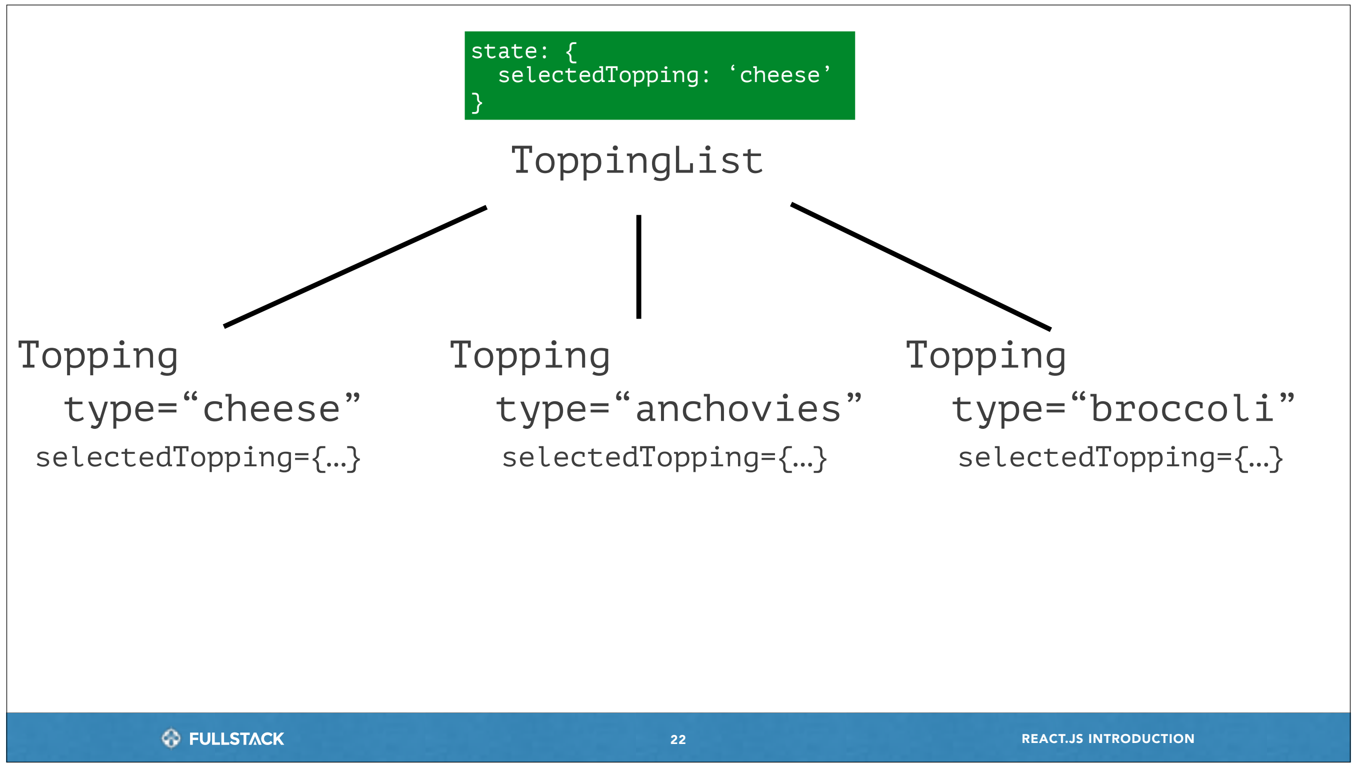
```
class ToppingList extends React.Component {
  constructor () {
    super()
    this.state = {
      selectedTopping: 'cheese'
    }
  }

  render () {
    return (
      <div>
        <h1>Your favorite topping is: {this.state.selectedTopping}</h1>
        <ul>
          <Topping selectedTopping={this.state.selectedTopping} type='cheese' />
          <Topping selectedTopping={this.state.selectedTopping} type='broccoli' />
          <Topping selectedTopping={this.state.selectedTopping} type='anchovies' />
        </ul>
      </div>
    )
  }
}
```

It should live in the ToppingList.



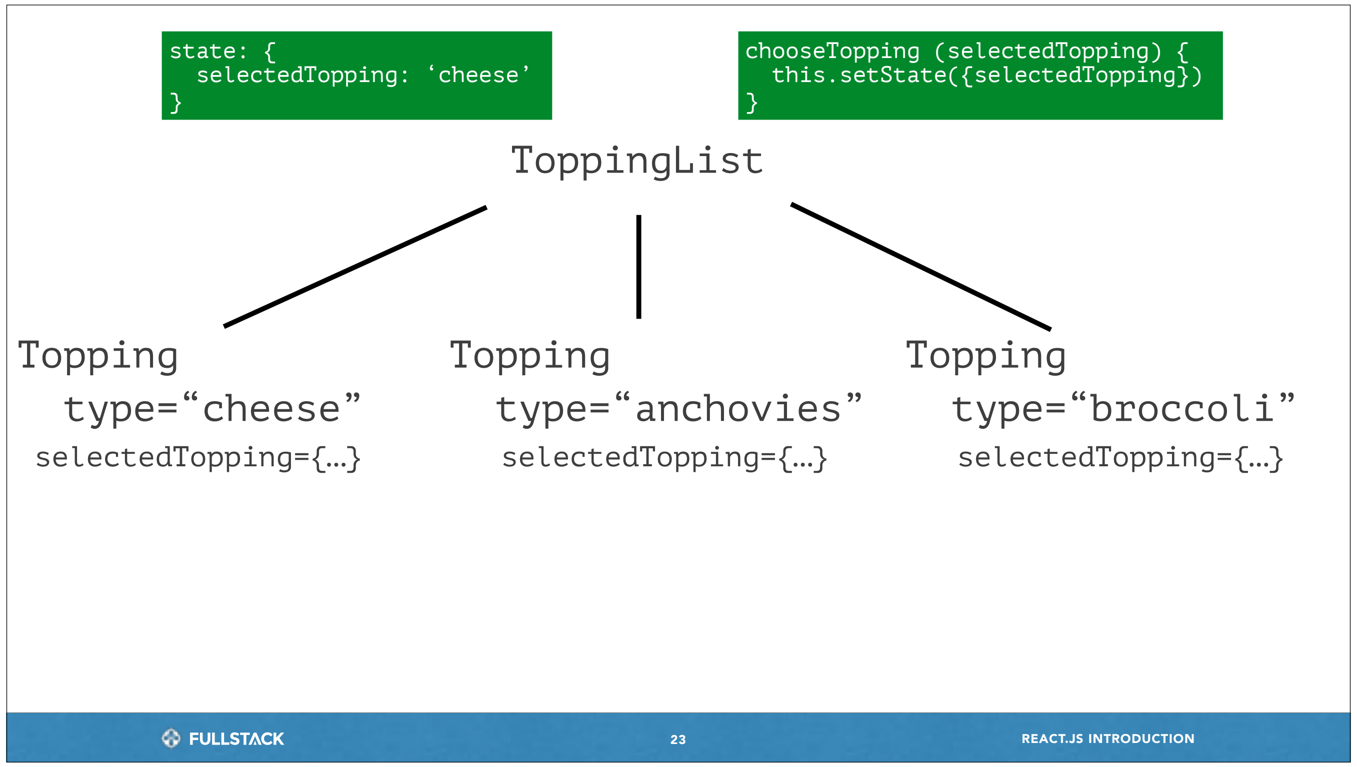
```
const Topping = (props) => {
  const isSelected = props.selectedTopping === props.type
  return (
    <div className={isSelected && 'selected'}>{props.type}</div>
  )
}
```

Then each individual topping can just calculate whether it should be underlined or not (for example, using a css class name”


```
state: {
  selectedTopping: ‘cheese’
}
```

But we need a function that will change our state. And what’s the only way to change state in react?

Where should this method live?

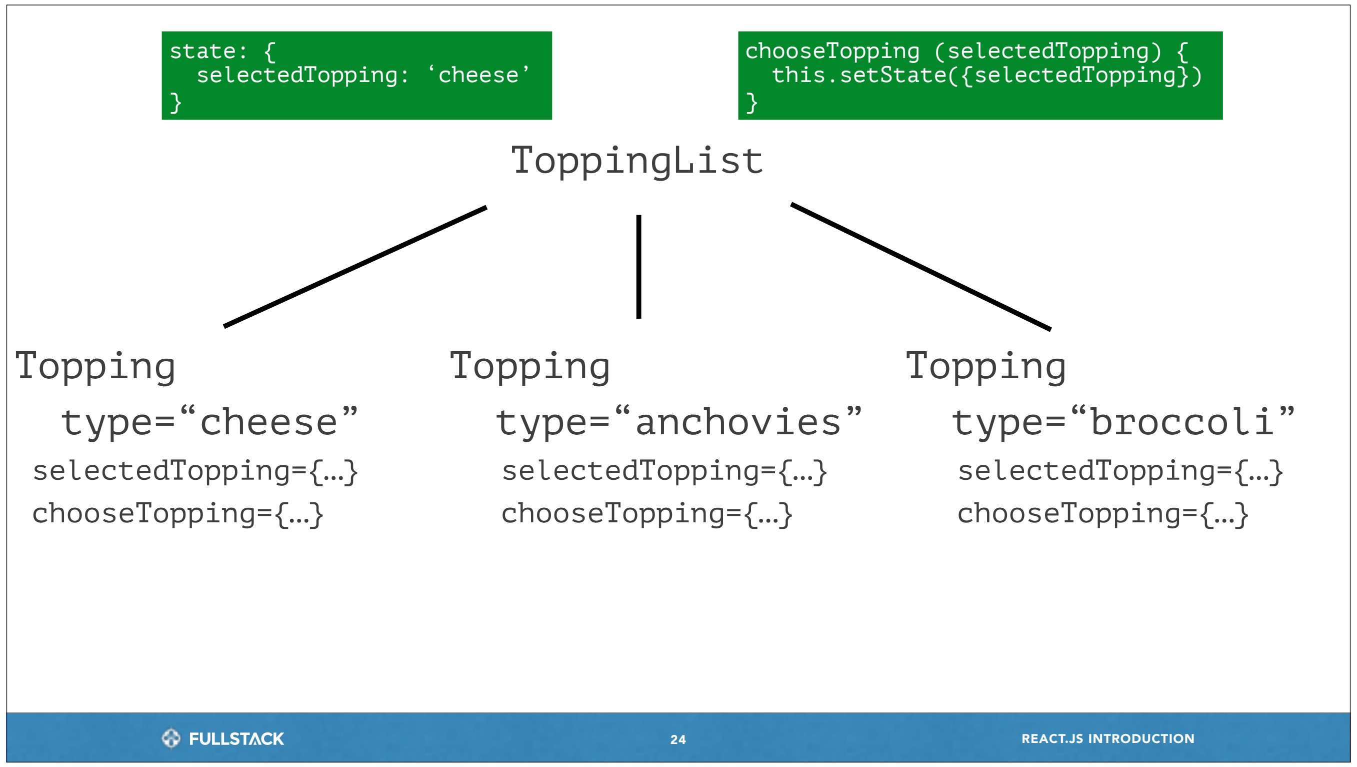


```
state: {
  selectedTopping: ‘cheese’
}
```

```
chooseTopping (selectedTopping) {
  this.setState({selectedTopping})
}
```

It can only live on the ToppingList component.

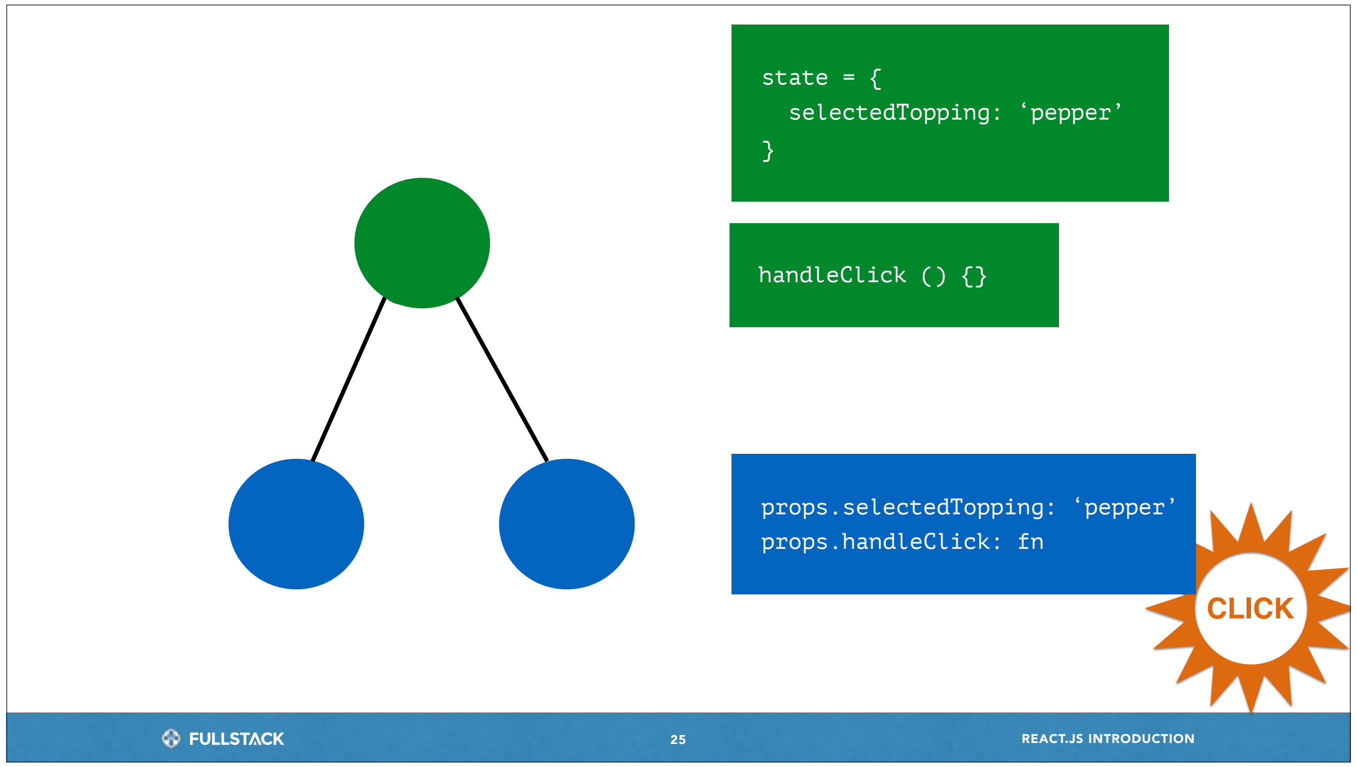
But we want this method to get called when a Topping is clicked.



```
state: {
  selectedTopping: ‘cheese’
}
```

```
chooseTopping (selectedTopping) {
  this.setState({selectedTopping})
}
```

So we pass down the method as well! The ToppingList contains all of the state and behavior, and shares it with its child components.

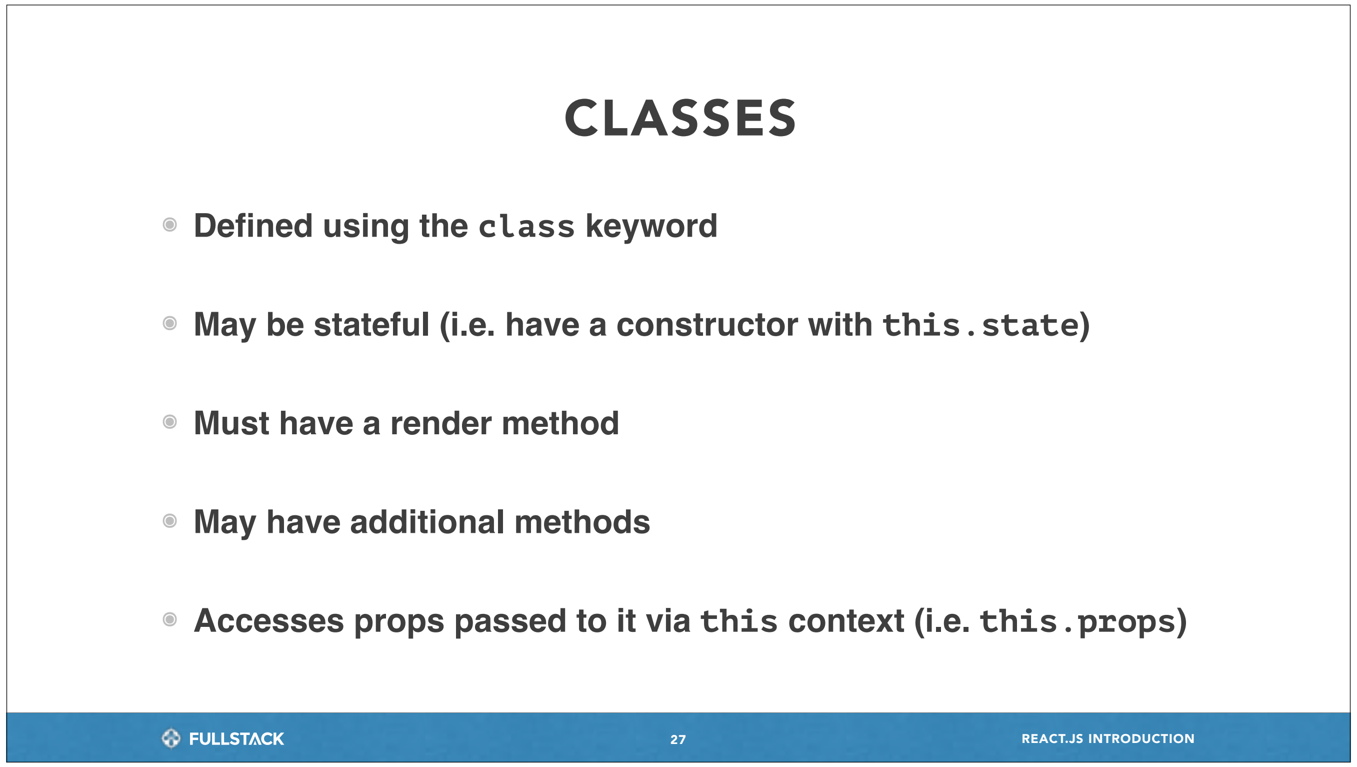


```
state = {
  selectedTopping: ‘pepper’
}
```

```
handleClick () {}
```

```
props.selectedTopping: ‘pepper’
props.handleClick: fn
```

CLASS COMPONENTS VS FUNCTIONAL COMPONENTS



CLASSES

- Defined using the `class` keyword
- May be stateful (i.e. have a constructor with `this.state`)
- Must have a render method
- May have additional methods
- Accesses props passed to it via `this` context (i.e. `this.props`)

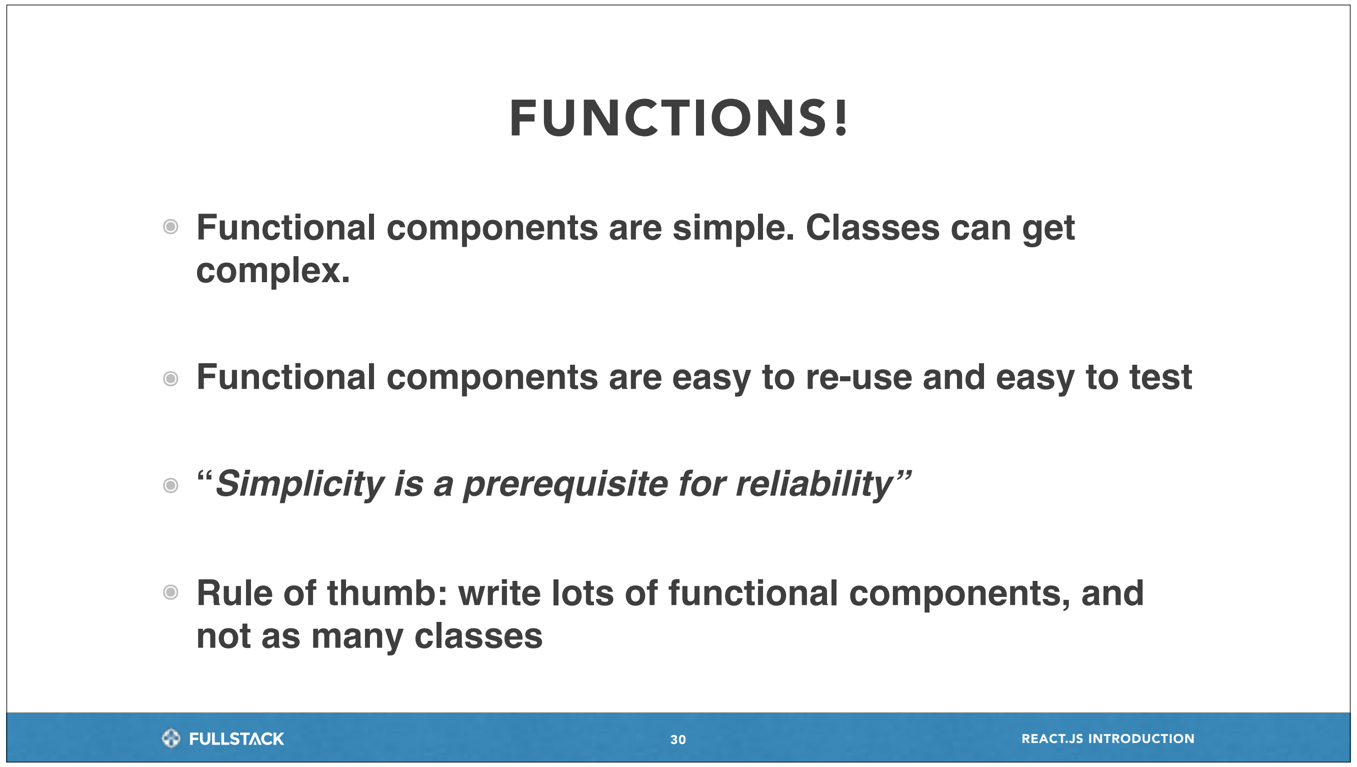
A component written as a class could receive props from another parent component. We haven't seen this yet, but it can. The only difference is that because it's a class, the way that it needs to get at its own props is via this context. React puts them there for you.

FUNCTIONS

- Just a function
- No state, no additional methods or functionality
- The function's return value *is* the "render"
- Accesses props passed to it via the first argument to that function (i.e. `const Topping = (props) => {...}`)

WHICH WOULD YOU PREFER?

Classes or functions?



FUNCTIONS!

- Functional components are simple. Classes can get complex.
- Functional components are easy to re-use and easy to test
- “*Simplicity is a prerequisite for reliability*”
- Rule of thumb: write lots of functional components, and not as many classes

Functions just take props as an argument, and calculate a view. Pure and simple.