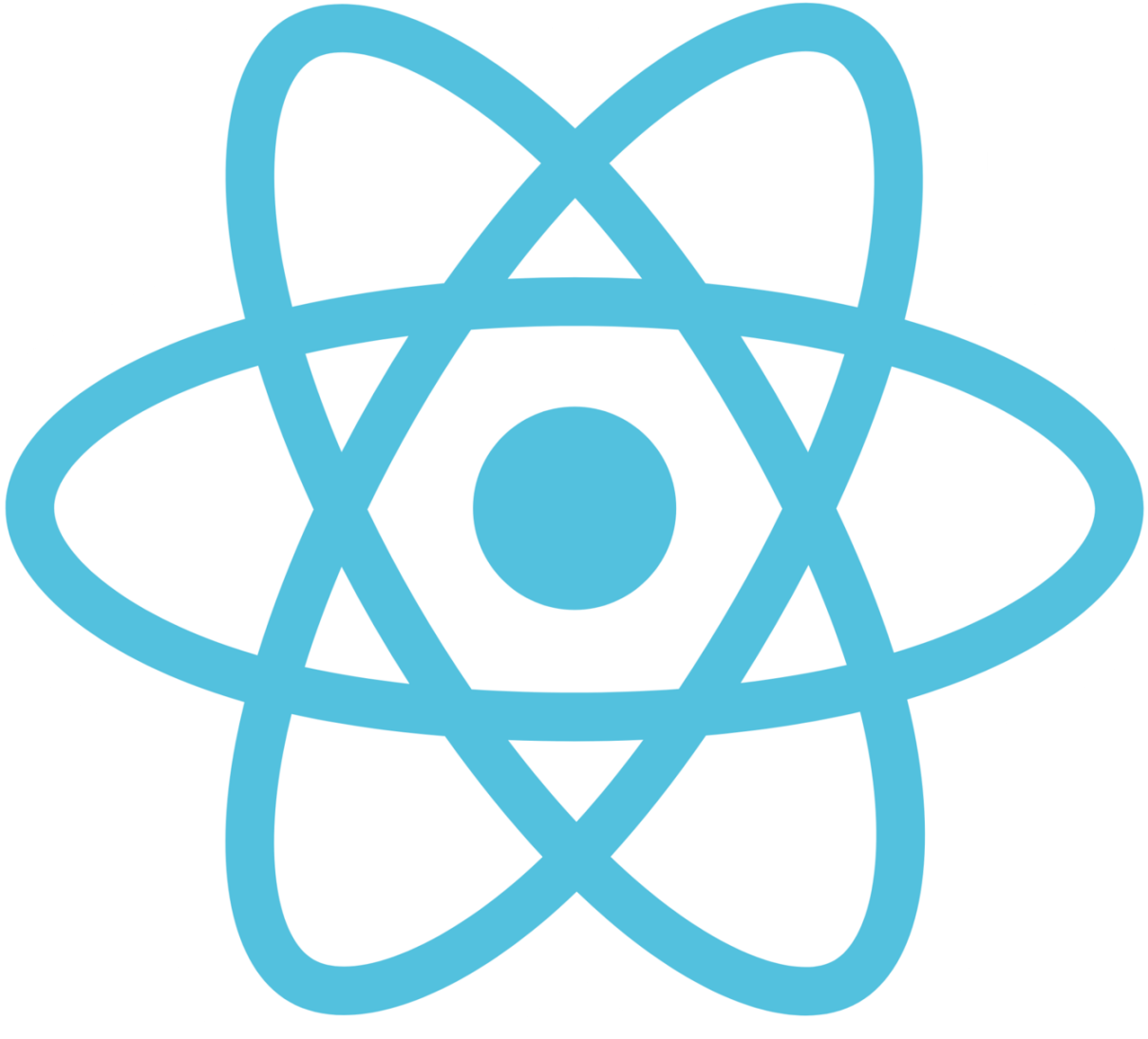


The React fundamentals.

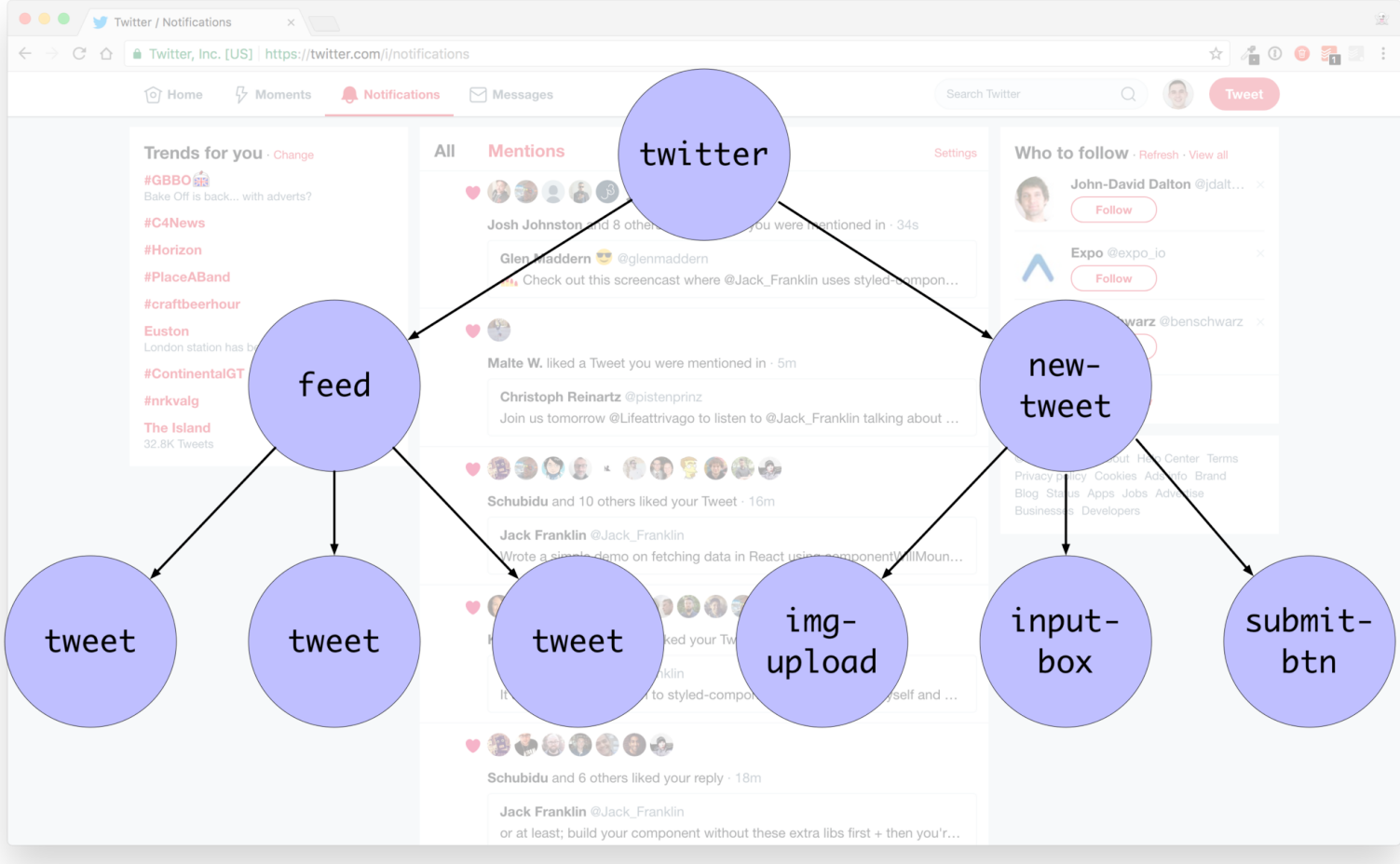
We're going to talk about *only* React for now.

Not a single third party library*

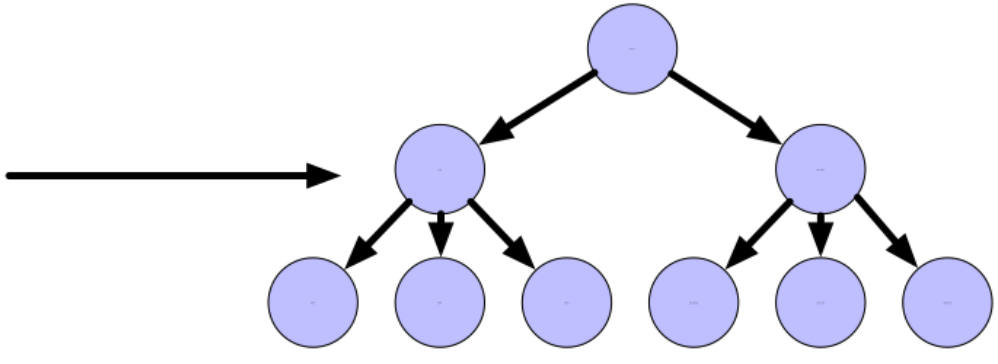
*ok there's one but I wrote it so that's allowed

By learning the core React framework and ideas, you'll be equipped to pick up, learn and work with any additional React libraries.

Components

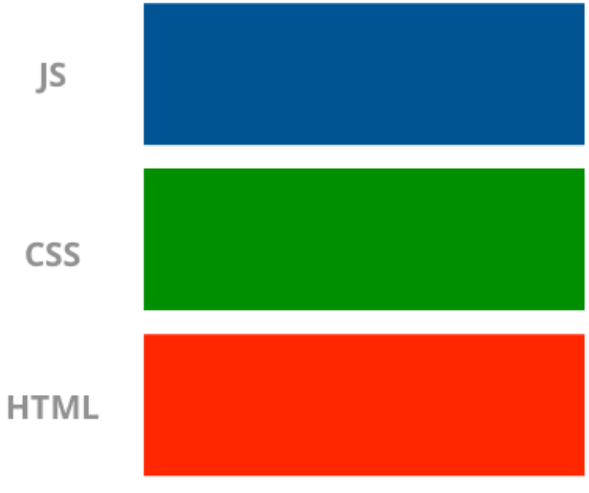

Twitter / Notifications
Twitter, Inc. [US] https://twitter.com/i/notifications
Home
Moments
Notifications
Messages
Search Twitter
Tweet
Trends for you · Change
#GBBO
Bake Off is back... with adverts?
#C4News
#Horizon
#PlaceABand
#craftbeerhour
Euston
#ContinentalGT
#nrkvalg
The Island
32.8K Tweets
All
Mentions
Settings
Josh Johnston
Glen Maddern @glenmaddern
Check out this screencast where @Jack_Franklin uses styled-compon…
Malte W. liked a Tweet you were mentioned in · 5m
Christoph Reinartz @pistenprinz
Join us tomorrow @Lifeattrivago to listen to @Jack_Franklin talking about …
Schubidu and 10 others liked your Tweet · 16m
Jack Franklin @Jack_Franklin
Schubidu and 6 others liked your reply · 18m
Jack Franklin @Jack_Franklin
or at least; build your component without these extra libs first + then you'r…
Who to follow · Refresh · View all
John-David Dalton @jdalt…
Follow
Expo @expo_io
Follow
Privacy policy Cookies Ads info Brand
Blog Status Apps Jobs Advertise
Businesses Developers
twitter
feed
new-tweet
tweet
tweet
tweet
img-upload
input-box
submit-btn

```
{
  url: '/notifications',
  userId: 12345,
  avatar: 'foo.jpg',
  latestTweets: [
    {
      id: 456,
      text: 'Hello world',
    },
    …
  ],
}
```



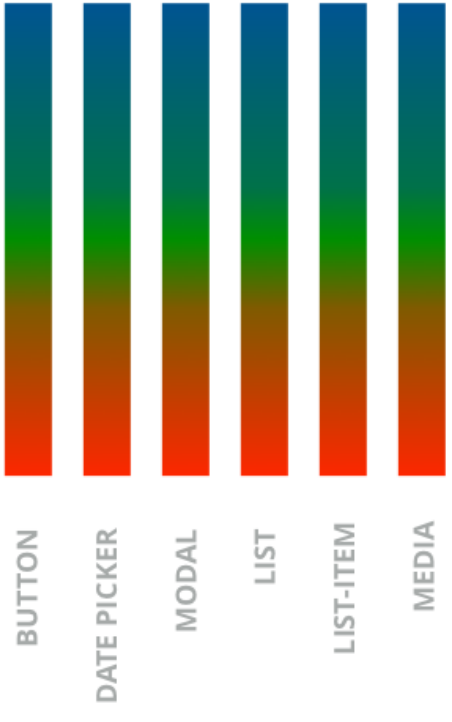
State → UI

Separation of Concerns



Separation of Concerns

(only, from a different point of view)



by Cristiano Rastelli, @areaweb

React + the DOM

- Tell React what each component should render.
- React takes care of the DOM for you.

Let's get started!

Here's the code for exercise 1. Let's walk through it together first.

```
import ReactDOM from 'react-dom'
import React from 'react'

const HelloWorld = () => {
  // TODO: can you change the h1 to another element?
  // how would we give the h1 a class name?
  return React.createElement('h1', null, 'Hello World')
}

ReactDOM.render(
  React.createElement(HelloWorld),
  document.getElementById('react-root')
)
```

```
import ReactDOM from 'react-dom'
import React from 'react'
```

import React and ReactDOM

```
const HelloWorld = () => {
  // TODO: can you change the h1 to another element?
  // how would we give the h1 a class name?
  return React.createElement('h1', null, 'Hello World')
}

ReactDOM.render(
  React.createElement(HelloWorld),
  document.getElementById('react-root')
)
```

```
import ReactDOM from 'react-dom'
import React from 'react'

const HelloWorld = () => {
  // TODO: can you change the h1 to another element?
  // how would we give the h1 a class name?
  return React.createElement('h1', null, 'Hello World')
}

ReactDOM.render(
  React.createElement(HelloWorld),
  document.getElementById('react-root')
)
```

our first component!

```
npm run exercise react 1
```

```
import ReactDOM from 'react-dom'
import React from 'react'

const HelloWorld = () => {
  // TODO: can you change the h1 to another element?
  // how would we give the h1 a class name?
  return React.createElement('h1', null, 'Hello World')
}
```

our first component!

```
ReactDOM.render(
  React.createElement(HelloWorld),
  document.getElementById('react-root')
)
```

render our first component into the DOM

React.createElement is quite verbose...

Exercise 2

```
const HelloWorld = ()
  // TODO: can you change the h1 to another element?
  // how would we give the h1 a class name?
  return <h1>Hello World</h1>
}

ReactDOM.render(<HelloWorld />,
 document.getElementById('react-root')
)
```

JSX!

remember, JSX gets compiled to React.createElement.

It's just JavaScript!

```
const HelloWorld = props => {
  // TODO: pass through another prop to customise the greet
  // rather than it always be hardcoded as Hello
  return <h1>Hello, {props.name}</h1>
}

ReactDOM.render(
  <HelloWorld name="Jack" />,
  document.getElementById('react-root')
)
```

we can pass properties (or, props) through to components to configure them or change their behaviour

remember, props are all just JavaScript, so you can pass through any data type - these aren't HTML attributes

Exercise 3

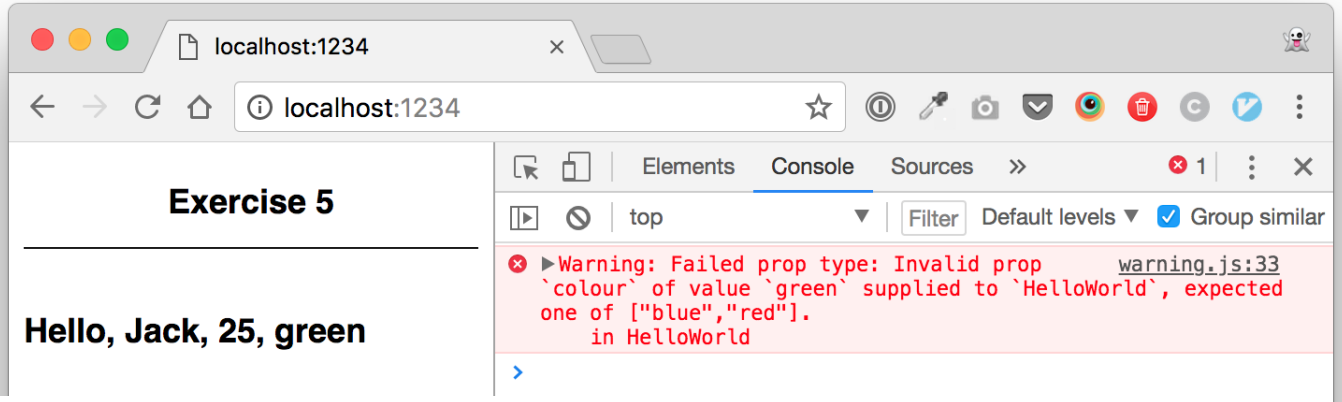
```
const bunchOfProps = {
  name: 'Jack',
  age: 25,
  colour: 'blue',
}

ReactDOM.render(
  <HelloWorld
    name={bunchOfProps.name}
    age={bunchOfProps.age}
    colour={bunchOfProps.colour}
  />,
  document.getElementById('react-root')
)
```

or:

```
ReactDOM.render(
  <HelloWorld {...bunchOfProps} />,
  document.getElementById('react-root')
)
```

Exercise 4

```
HelloWorld.propTypes = {
  name: PropTypes.string.isRequired,
  colour: PropTypes.oneOf(['blue', 'red']).isRequired,
}
```

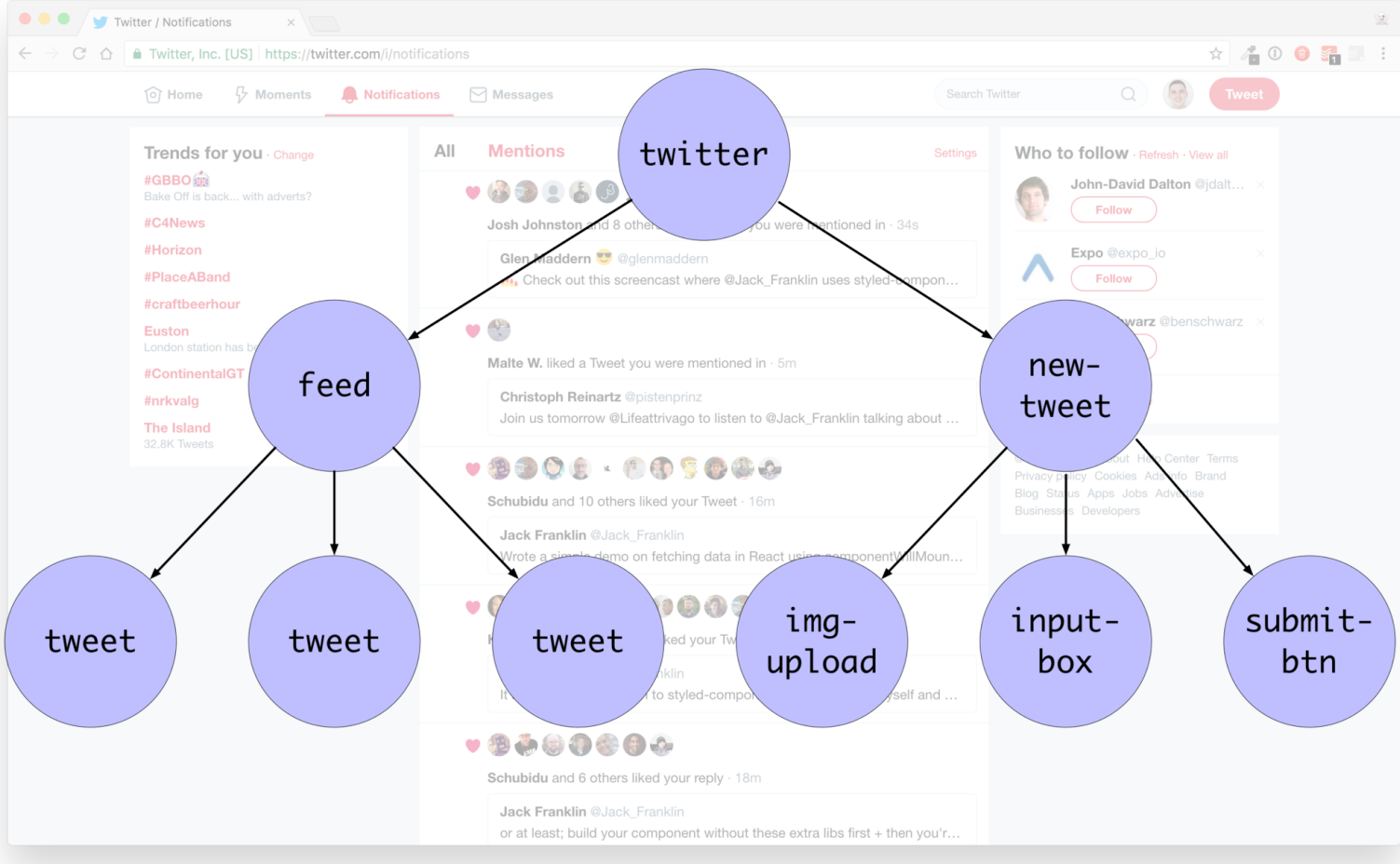


Documenting your components with prop types

This seems like a chore, but trust me, you'll thank yourself in the future!

https://reactjs.org/docs/typechecking-with-proptypes.html

Exercise 5


Twitter / Notifications
Twitter, Inc. [US] https://twitter.com/i/notifications
Home
Moments
Notifications
Messages
Search Twitter
Tweet
Trends for you · Change
#GBBO
Bake Off is back... with adverts?
#C4News
#Horizon
#PlaceABand
#craftbeerhour
Euston
#ContinentalGT
#nrkvalg
The Island
32.8K Tweets
All
Mentions
Settings
Josh Johnston
Glen Maddern @glenmaddern
Check out this screencast where @Jack_Franklin uses styled-compon…
Malte W. liked a Tweet you were mentioned in · 5m
Christoph Reinartz @pistenprinz
Join us tomorrow @Lifeattrivago to listen to @Jack_Franklin talking about …
Schubidu and 10 others liked your Tweet · 16m
Jack Franklin @Jack_Franklin
Schubidu and 6 others liked your reply · 18m
Jack Franklin @Jack_Franklin
or at least; build your component without these extra libs first + then you'r…
Who to follow · Refresh · View all
John-David Dalton @jdalt…
Follow
Expo @expo_io
Follow
Privacy policy Cookies Ads info Brand
Blog Status Apps Jobs Advertise
Businesses Developers
twitter
feed
new-tweet
tweet
tweet
tweet
img-upload
input-box
submit-btn

```
const AskQuestion = () => {
  return <p>How is your day going today?</p>
}

const HelloWorld = props => {
  return (
    <div>
      <AskQuestion />
      <h1>
        {props.greeting}, {props.name}
      </h1>
    </div>
  )
}
```

Components can render other components.

React components must start with a capital letter.

Exercise 6

Managing state

Props

Data a component is given and uses but **cannot change.**

State

Data a component **owns** and can **change.**

Functional components

What we've used so far. These components **cannot have state.**

Class components

We define our components as classes. These components **can have state.**

```
import React, { Component } from 'react'

class MyComponent extends Component {
  constructor(props) {
    super(props)

    this.state = {...}
  }

  render() {
    return <p>Hello world</p>
  }
}
```

this is boilerplate you don't need to worry about

this is like the body of the functional components we have been using so far!

Listening to user events

```
onButtonClickIncrement() {

}

render() {
  return (
    <div>
      <p>current count: {this.state.count}</p>
      <button onClick={this.onButtonClickIncrement.bind(this)}>
        Click to increment
      </button>
    </div>
  )
}
```

we have to bind to ensure the right scope within the event handler

(we'll see a nicer way to do this later on)

Updating state

To update the state, we have to use React's method

This ensures React knows about our state change, and can act accordingly.

```
this.state.foo = 'bar'
```

Updating state

```
this.setState({
  newValue: 3,
})
```

when the new state doesn't depend on the old state

Updating state

```
this.setState(function(previousState) {
  return {
    newValue: previousState.newValue + 1
  }
})
```

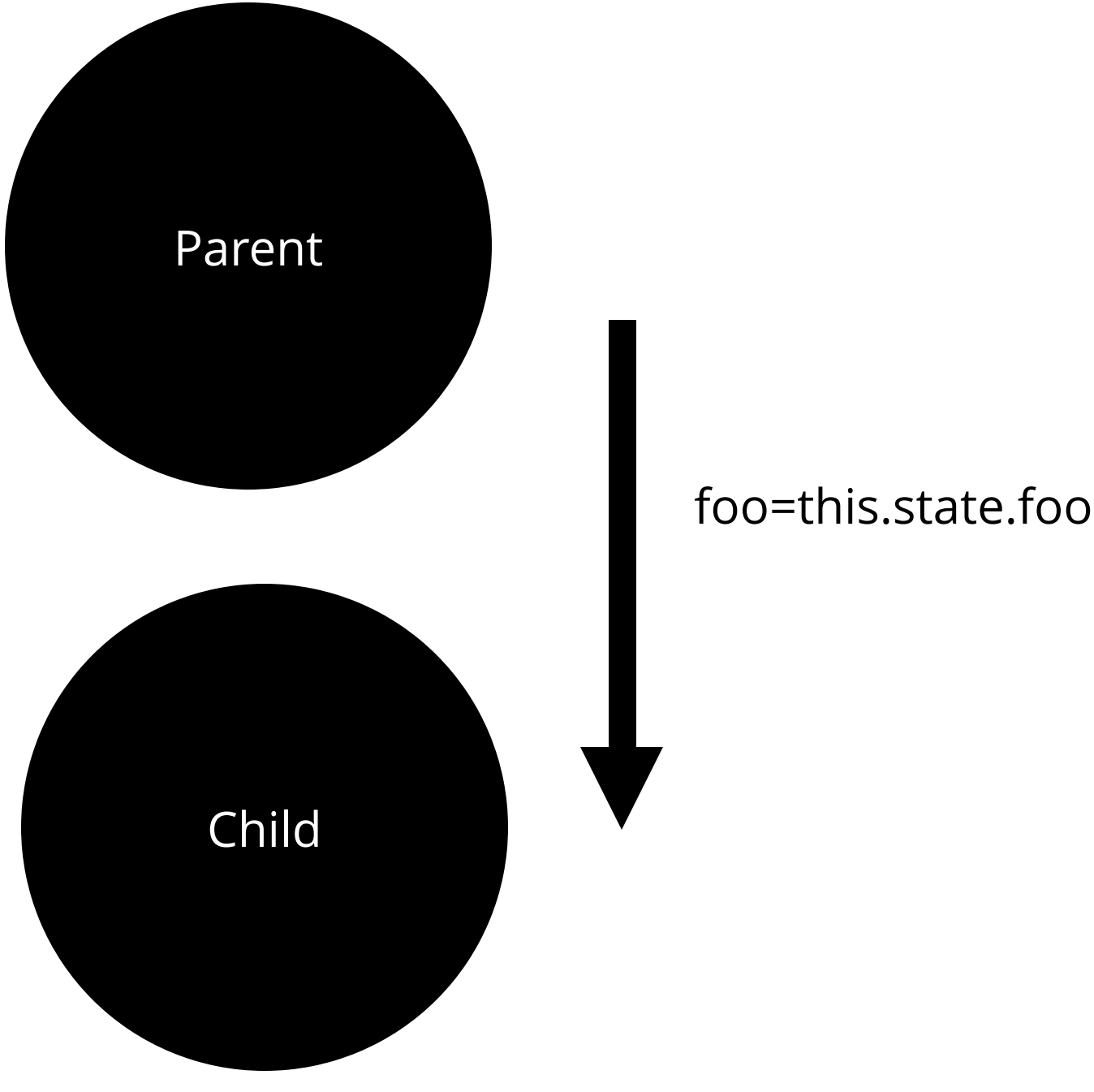
when the new state does depend on the old state

```
class Counter extends Component {
  constructor(props) {
    super(props)

    this.state = {
      count: 0,
    }
  }

  onButtonClickIncrement() {
    this.setState(prevState => {
      return { count: prevState.count + 1 }
    })
  }

  render() {
    return (
      <div>
        <p>current count: {this.state.count}</p>
        <button onClick={this.onButtonClickIncrement.bind(this)}>
          Click to increment
        </button>
      </div>
    )
  }
}
```

Exercise 7

Passing state to child components

```
const Parent = () => {
  return <Child />
}
```


Parent
foo=this.state.foo
Child

```
<div>
  <Count count={this.state.count} />
  <button onClick={this.onButtonClickIncrement.bind(this)}>
    Click to increment
  </button>
</div>
```

The Count component gets a count property, that has come from state of its parent.

Exercise 8

Parent and child communication

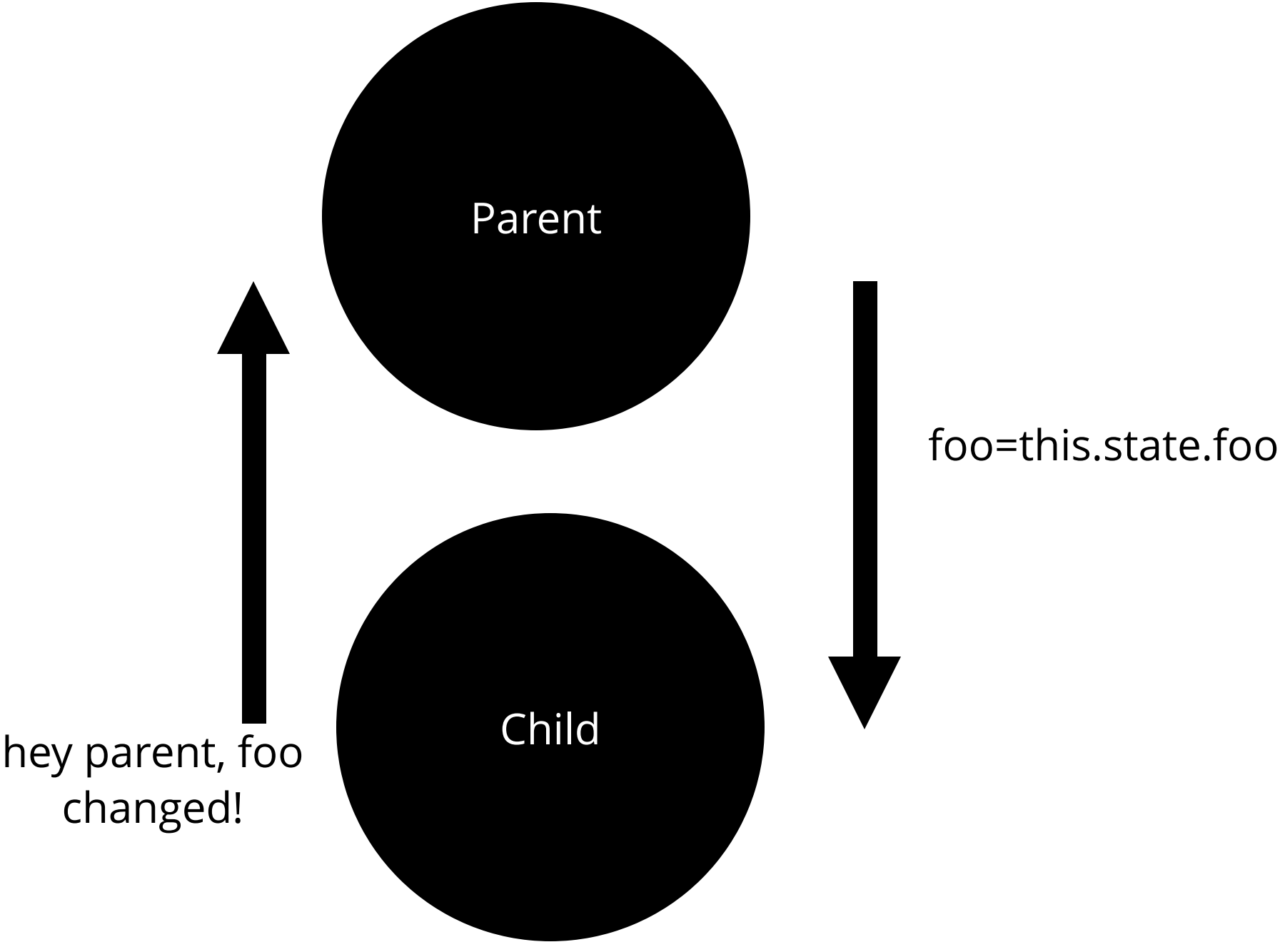
Sometimes we'll have state in the parent

that we give to the child

```
<SomeChildComp foo={this.state.foo} />
```

And sometimes the child needs to let us know that the state has changed.

```
<SomeChildComp
  foo={this.state.foo}
  onFooChange={this.onFooChange}
/>
```


Parent
foo=this.state.foo
Child
hey parent, foo changed!

```
<Count
  count={this.state.count1}
```

here's some data for you to render

```
  onIncrement={this.incrementCount1.bind(this)}
/>
```

and when that data changes, this is how you tell me about it

Exercise 9

Rendering lists of data

```
this.state = {
  counts: [0, 0, 0],
}
```

often we have lists of data that we want to render

```
    return (
      <div>
        <Count
          count={this.state.counts[0]}
          onIncrement={this.incrementCount.bind(this, 0)
        />

        <Count
          count={this.state.counts[1]}
          onIncrement={this.incrementCount.bind(this, 1)
        />
      </div>
    )
```

and this is a pretty manual way of doing it that doesn't deal well with us updating our list of values

mapping over arrays in JavaScript

```
const numbers = [1, 2, 3]

const newNumbers = numbers.map(function(x)
  return x * 2
})

// or

const newNumbers = numbers.map(x => x * 2)

newNumbers === [2, 4, 6]
```

so could we take our array of counts, and map them into an array of <Count /> components?

```
this.state = {
  counts: [0, 0, 0],
}


this.state.counts.map((count, index) => {
  return (
    <Count
      count={count}
      onIncrement={this.incrementCount.bind(this, index)
    />
  )
})
```

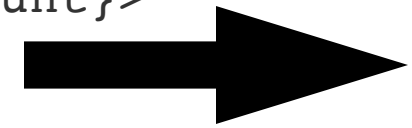
Exercise 10

Allowing our components to take custom children.

React Children

```
<Count count={this.state.count} />
```

```
<Count count={this.state.count}>
  <p>some random child</p>
</Count>
```



```
this.props.children
```

```
<Count
  count={this.state.count}
  children={<p>some random child</p>}
/>
```

remember, props can be anything!

Exercise 11

we'll see more usages of React's children later on over the workshop.

Quick aside

Tooling

This workshop purposefully avoids talking about tooling and set up. We're focusing purely on React today (but questions are welcome!)

But I want to take a few minutes to quickly talk about some of the things going on in the workshop behind the scenes.

Bundler

A tool that takes all of our code and generates a bundle of HTML, CSS and JS for us.

Today I'm using https://parceljs.org/, but I'm also a big fan of https://webpack.js.org/

Transpiler

A tool that takes our modern JS and converts it back into JS that older browsers can use. Depending on what browsers you support, these will do different transforms.

Today we're using https://babeljs.io/ with some plugins:

- babel-preset-env
- babel-preset-react

Object rest spread

Enables https://github.com/tc39/proposal-object-rest-spread, which is at Stage 4, meaning it will become part of JavaScript.

Class properties

Enables https://github.com/tc39/proposal-class-fields, which is a Stage 3 proposal.

Class properties mean much nicer event handlers.

```
onButtonClickIncrement() {

}

<button onClick={this.onButtonClickIncrement.bind(this)}>
```

becomes...

```
onButtonClickIncrement = () => {

}

<button onClick={this.onButtonClickIncrement}>
```

arrow functions are always bound to the scope of the thing that defined them

```
Foo.propTypes = {...}

constructor(props) {
  super(props)
  this.state = {...}
}
```

becomes...

```
class Foo extends Component {
  static propTypes = {...}

  state = {...}
}
```

Code formatting

I find formatting code rather mundane and boring - so I let tools do it for me!

https://prettier.io/ is fantastic and there are editor plugins available for all popular editors.

Dealing with async data

(we're going to dive into async in JS more later)

so-fetch-js

jackfranklin / so-fetch

Unwatch 1 | Star 44 | Fork 7

Code | Issues 4 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

No description, website, or topics provided.

Edit

Add topics

24 commits | 1 branch | 3 releases | 4 contributors

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

jackfranklin Fix up eslint+prettier config — Latest commit 371990e on Aug 28

src	README and Changelog for v0.3	3 months ago
.babelrc	add rollup as bundler to generate separate builds: umd, es modules, c...	3 months ago
.eslintrc	Fix up eslint+prettier config	3 months ago
.gitignore	add rollup as bundler to generate separate builds: umd, es modules, c...	3 months ago
CHANGELOG.md	README and Changelog for v0.3	3 months ago
README.md	README and Changelog for v0.3	3 months ago
package.json	Fix up eslint+prettier config	3 months ago
rollup.config.js	README and Changelog for v0.3	3 months ago
test-setup.js	Initial commit	3 months ago
yarn.lock	Fix up eslint+prettier config	3 months ago

README.md

so-fetch

A small wrapper around the fetch API with some additional behaviours.

Installation

Install this module with npm or yarn.

```
yarn add so-fetch-js
npm install so-fetch-js
```

```
import fetch from 'so-fetch-js'

fetch('/users').then(response => {
  console.log(response.data)
})
```

https://jsonplaceholder.typicode.com/posts


```
[
  {
    userId: 1,
    id: 1,
    title: "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
    body: "quia et suscipit suscipit recusandae consequuntur expedita et cum reprehenderit
            nostrum rerum est autem sunt rem eveniet architecto"
  },
  ...
]
```


https://jsonplaceholder.typicode.com/posts/1

Component lifecycle

https://reactjs.org/docs/react-component.html#the-component-lifecycle

componentDidMount

If you need to load data from a remote endpoint, this is a good place to instantiate the network request.

https://reactjs.org/docs/react-component.html#componentdidmount

```
componentDidMount() {
  const urlForPost = `https://jsonplaceholder.typicode.com/posts
    this.props.id
  }`

  fetch(urlForPost).then(response => {
    const post = response.data
    // TODO: put this post into the state
    console.log('I got the post!', post)
  })
}
```

Exercise 12

```
{this.state.post === null && <div>Loading...</div>}
```

```
render() {
  return this.state.post ? (
    <div>
      <h1>{this.state.post.title}</h1
      <p>{this.state.post.body}</p>
    </div>
  ) : (
    <p>Loading</p>
  )
}
```

Conditional rendering in JSX

forms in React

Controlled inputs

React controls the **value** of an input

And React controls the **onChange** event of an input.

```
<input
  type="text"
  name="post-id"
  value={this.state.userPostInput}
  onChange={this.userInputChange}
/>
```

value: the actual value of the input box

onChange: what React calls when the user types in the box, so we can update the value

```
<form onSubmit={this.onSubmit}>
```

```
userInputChange = event => {
  this.setState({
    userPostInput: event.target.value,
  })
}
```

event.target.value === the latest value
from the input that the user has typed

```
userInputChange = e => {
  console.log('got user input value', e.target.value)
  // TODO: update the userPostInput state with the new value when the user types
}

onSubmit = e => {
  e.preventDefault()
  console.log('got form submit!')
  // TODO: update the searchID state with the latest user post ID when the form is sub
}
```

```
<form onSubmit={this.onSubmit} className="search-form">
  <label>
    Please enter the ID of a post
    <input
      type="text"
      name="post-id"
      value={this.state.userPostInput}
      onChange={this.userInputChange}
    />
  </label>
  <button type="submit">Go</button>
  {/* TODO: add another button that clears out the user input value *
</form>

{this.state.searchId && (
  <p>The ID you searched for is: {this.state.searchId}</p>
)}
```

Exercise 13 (unlucky for some)

Forms and APIs

fetching based off user input

```
onSubmit = e => {
  e.preventDefault()
  console.log('got form submit!')
  // TODO: call this.fetchPost(), passing in the right ID
}
```

Exercise 14

multiple components in multiple files

ES2015 Modules

```
export default class Post extends Component {
    ...
}
```

```
import Post from './post'
```

React lifecycle hooks
componentDidUpdate

https://reactjs.org/docs/react-component.html#componentdidupdate

componentDidUpdate

componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render.

Use this as an opportunity to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props (e.g. a network request may not be necessary if the props have not changed).

```
componentDidUpdate(prevProps, prevState) {
}
```

don't forget to check that the props or state have changed!

```
componentDidUpdate(prevProps, prevState) {
  if (prevProps.id !== this.props.id) {
  }
}
```

if this check passes, we can do our work

PostSearch: the main component that lets a user search by post ID

Post: the component that takes a post ID, fetches it, then renders it.

```
<form onSubmit={this.onSubmit} className="search-form">
  <label>
    Please enter the ID of a post
    <input
      type="text"
      name="post-id"
      value={this.state.userPostInput}
      onChange={this.userInputChange}
    />
  </label>
  <button type="submit">Go</button>
</form>
<Post id={this.state.searchId} />
```

But there's a bug!

Filling in a new ID and hitting 'go' does not load the new post.

```
export default class Post extends Component {
  static propTypes = {
    id: PropTypes.number,
  }

  state = {
    post: null,
  }

  componentDidMount() {
    this.fetchPost()
  }

   ...
}
```

Exercise 15

Extracting the user input.

Building on the last exercise, we've pulled out UserInput

```
render() {
  return (
    <div>
      <UserInput onSearchInputChange={this.onSubmit} />
      <Post id={this.state.searchId} />
    </div>
  )
}
```

UserInput: read the input from the form

Post: fetch and render the post given the ID.

we need to use the parent/child communication pattern we saw earlier

we've pulled out a component for getting the user's search ID, but it's not quite done yet.

```
class PostSearch extends Component {
  state = {
    searchId: 1,
  }

  onSubmit = id => {
    this.setState({ searchId: id })
  }

  render() {
    return (
      <div>
        <UserInput onSearchInputChange={this.onSubmit} />
        <Post id={this.state.searchId} />
      </div>
    )
  }
}
```

this is more parent to child communication like we saw earlier

Exercise 16

React fundamentals: fin

Any questions? :)

Advanced React

Let's pick up from exercise 16 of the fundamentals.

Let's talk about fetching posts

```
<div>
  <UserInput onSearchInputChange={this.onSubmit}
  <Post id={this.state.searchId} />
</div>
```

this is our render function, but we have no control over how a post is rendered!

<Post /> contains logic about fetching a post that we don't want to care about.

But we want to have control over what to render.

Over the next few exercises, we'll see different techniques for solving this problem

each one improves on the previous...so stick with it!

1. Compound components with mapping children.

React provides an API to take all the children given to a component and edit them.

```
// imagine this.props.children are all <p>foo</p>

return React.Children.map(this.props.children, child =>
  // we can edit the child and return a new one
})
```

https://reactjs.org/docs/react-api.html#reactchildrenmap

React also gives us cloneElement to take copies of the child but give it extra properties

```
// imagine this.props.children are all <p>foo</p>

return React.Children.map(this.props.children, child =>
   return React.cloneElement(child, { className: 'foo'}
})
```

https://reactjs.org/docs/react-api.html#reactchildrenmap

so in our case, the Post component can...

1. Take a child component that can output the post

2. Use React.children.map to pass the `post` property into its child.

```
const PostOutput = props =>
  props.post ? (
    <div>
      <span>Loaded post ID: {props.post.id}</sp
      <h1>{props.post.title}</h1>
      <p>{props.post.body}</p>
    </div>
  ) : (
    <p>Loading</p>
  )
```

```
<Post id={this.state.searchId}>
  <PostOutput />
</Post>
```

```
return React.Children.map(...)
```

Exercise 17

Problems with this approach?

- It's very implicit - from looking at the render method we have no idea that this is happening.
- It ties <PostOutput> to being used within <Post>
- We can't just put any old code within <Post>, it has to be a specific component that is expecting a specific prop.

2. Higher order components

Higher order?

```
const createAdder = x => {
  return function(y) { return x + y };
}

const createAdder = x => y => x + y

const addTwo = createAdder(2)

addTwo(2) // 4
```

Higher order component?

functions that return React components

```
const wrapWithDiv = Component => {
  const NewComponent = props => (
    return <div><Component {...props}></div>
  )

  return NewComponent
}

const Hello = () => <p>Hello world!</p>
const WrappedComponent = wrapWithDiv(Hello)

<WrappedComponent />
```

```
const withPost = ChildComponent => {

  return class Post extends Component {
    static propTypes = {
      id: PropTypes.number,
    }

    state = {
      post: null,
    }

    ...


    render() {
      return <ChildComponent {...this.props} post={this.state.post} />
    }
  }

}

export default withPost
```

takes a ChildComponent as an argument

we return a component that renders our child component with some extra props

```
import withPost from './post'
```

Exercise 18

How does this approach compare?

- It's more explicit, and the withPost function at least makes it clear that something is going on.
- Still creates some indirection and harder to follow code.
- The component we pass into withPost still has to know that it will end up being given a `post` prop.

Render functions

What if we gave <Post /> a function to call with the post it has fetched?

So it deals with the data fetching logic, but passes rendering control back to us.

```
export default class Post extends Component {
  static propTypes = {
    id: PropTypes.number,
    render: PropTypes.func.isRequired,
  }

  ...

  render() {
    return this.props.render(this.state.post)
  }
}
```

takes a render prop that we call to render

```
<Post
  id={this.state.searchId}

  render={post =>
    post ? (
      <div>
        <span>Loaded post ID: {post.id}</span
        <h1>{post.title}</h1>
        <p>{post.body}</p>
      </div>
    ) : (
      <p>Loading</p>
    )
  }

/>
```

```
{/* TODO: update this render function to output the post if we have one
     or render "Loading" if we don't have a post
    */}
<Post id={this.state.searchId} render={() => null} />
{/* TODO: once you've done that, pull that logic into a PostOutput componen
  * (hint: you'll find the prop types above) and use that within the render
  */}
```

Exercise 19

Render functions 2: an alternative approach

passing a function as the child prop

```
<Post id={this.state.searchId}
  render={() => null} />
```

these are equivalent but one uses the prop `render` and the other `children`

```
<Post id={this.state.searchId}
  children={() => null} />
```

```
<Post id={this.state.searchId}>{() => null}</Post>
```

but we can also pass the children prop in implicitly.

This is the same as using the `children=` prop syntax!

(note that exercise 20 is broken at the start on purpose! Look at 20.js to see what to do)

```
<Post id={this.state.searchId}>
  {post => <PostOutput post={post} />}
</Post>
```

can we make Post support both of these?

```
<Post
  id={this.state.searchId}
  render={post => <PostOutput post={post} />}
/>
```

So how does this compare?

- Very explicit! It's obvious what's going on
- There is no "magic" or implicit things happpening
- We have full control over how to render and what we render.

Let's take another break!

Any questions? :)

React's context API

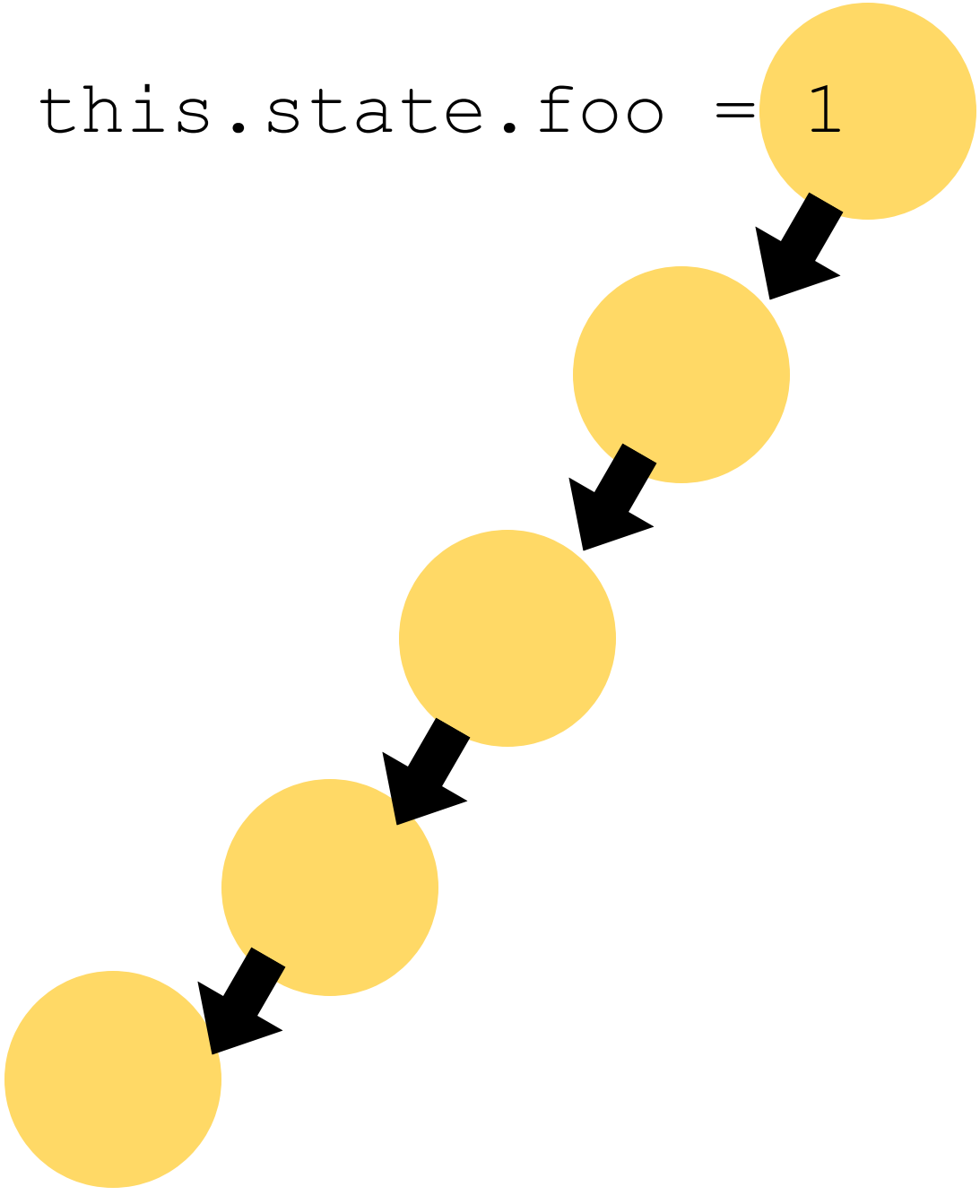
Note: this was introduced in React 16.3, fairly recently.

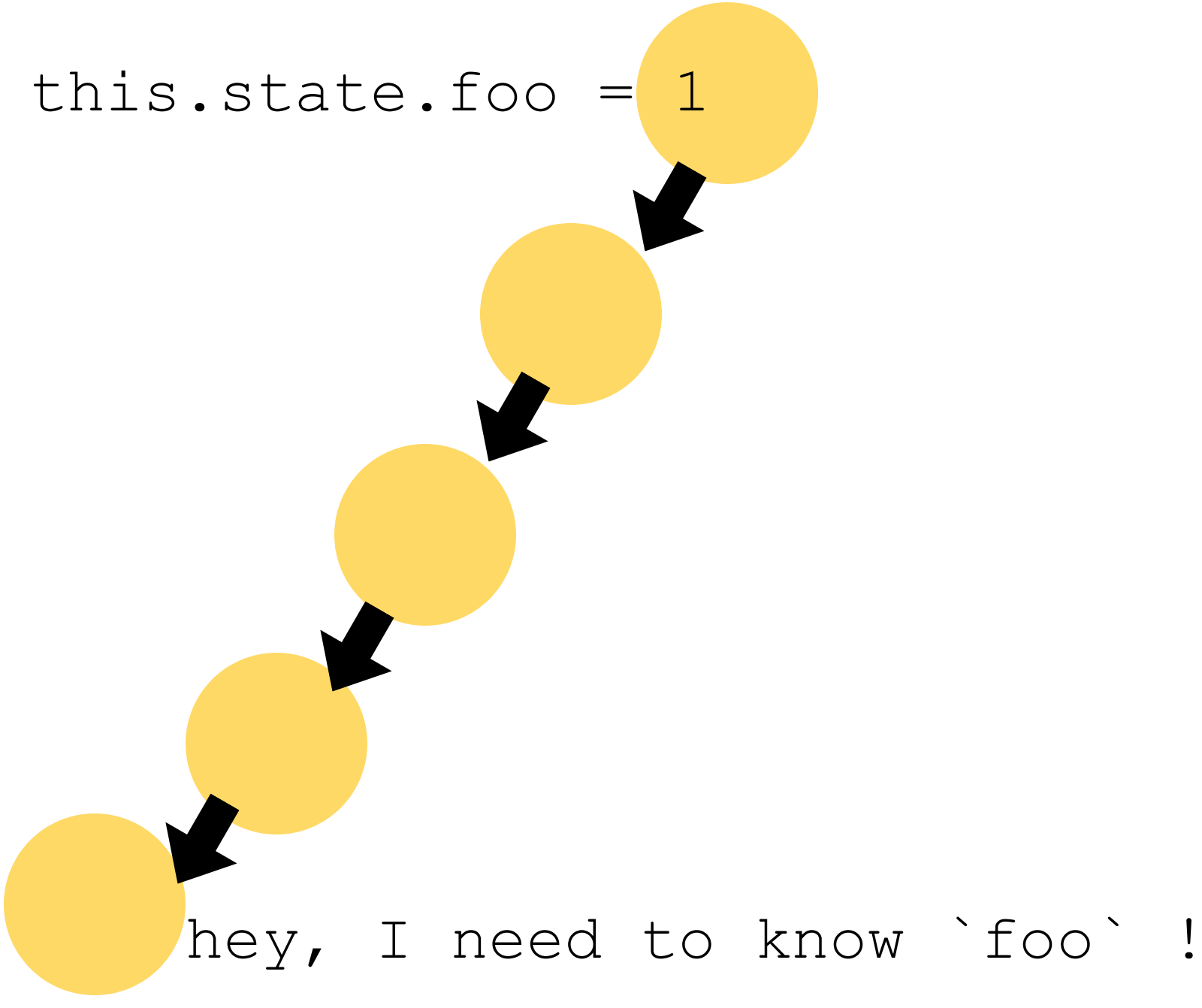
If you google for tutorials, make sure they use React 16.3 or higher.

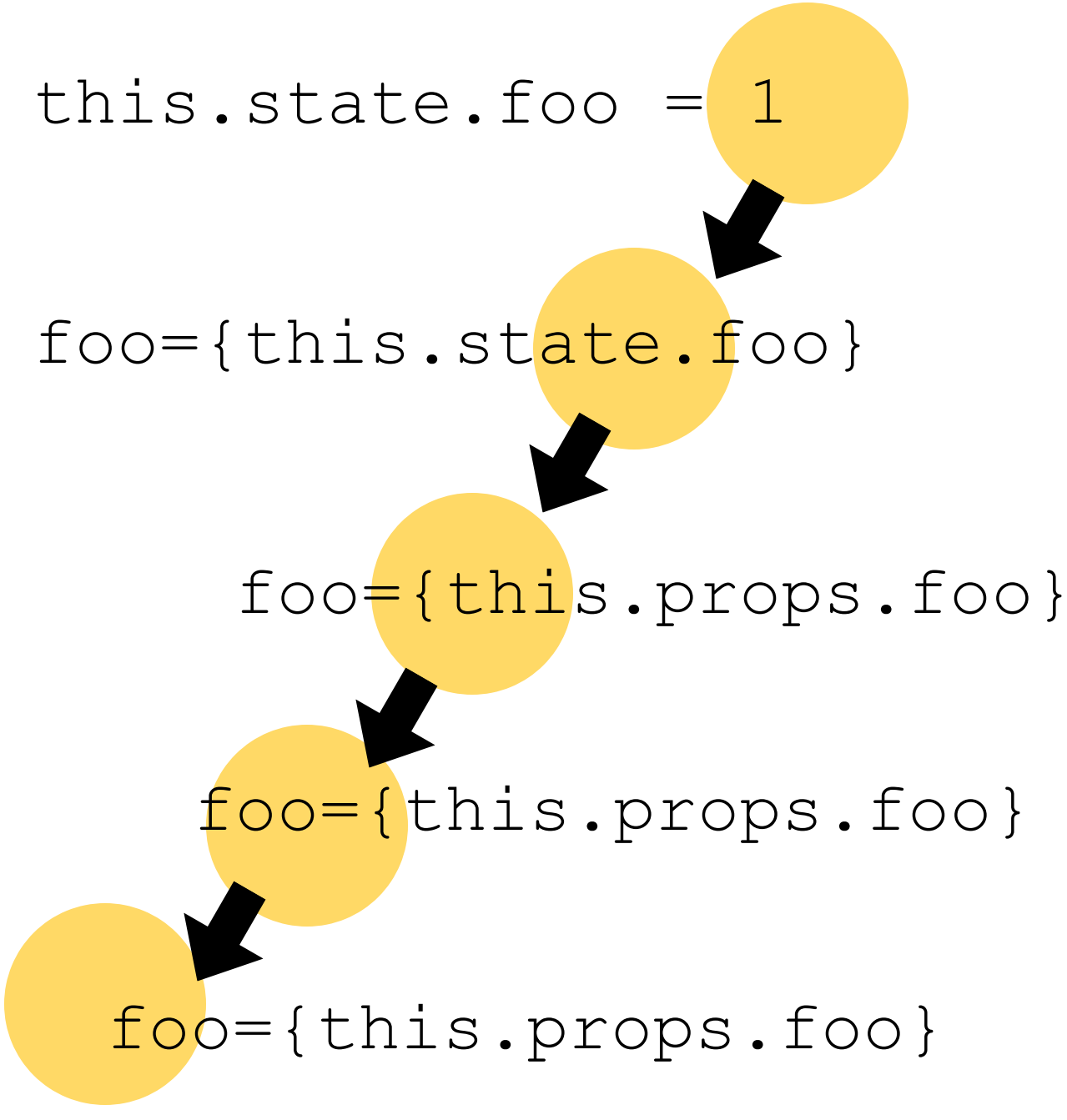
What is context?

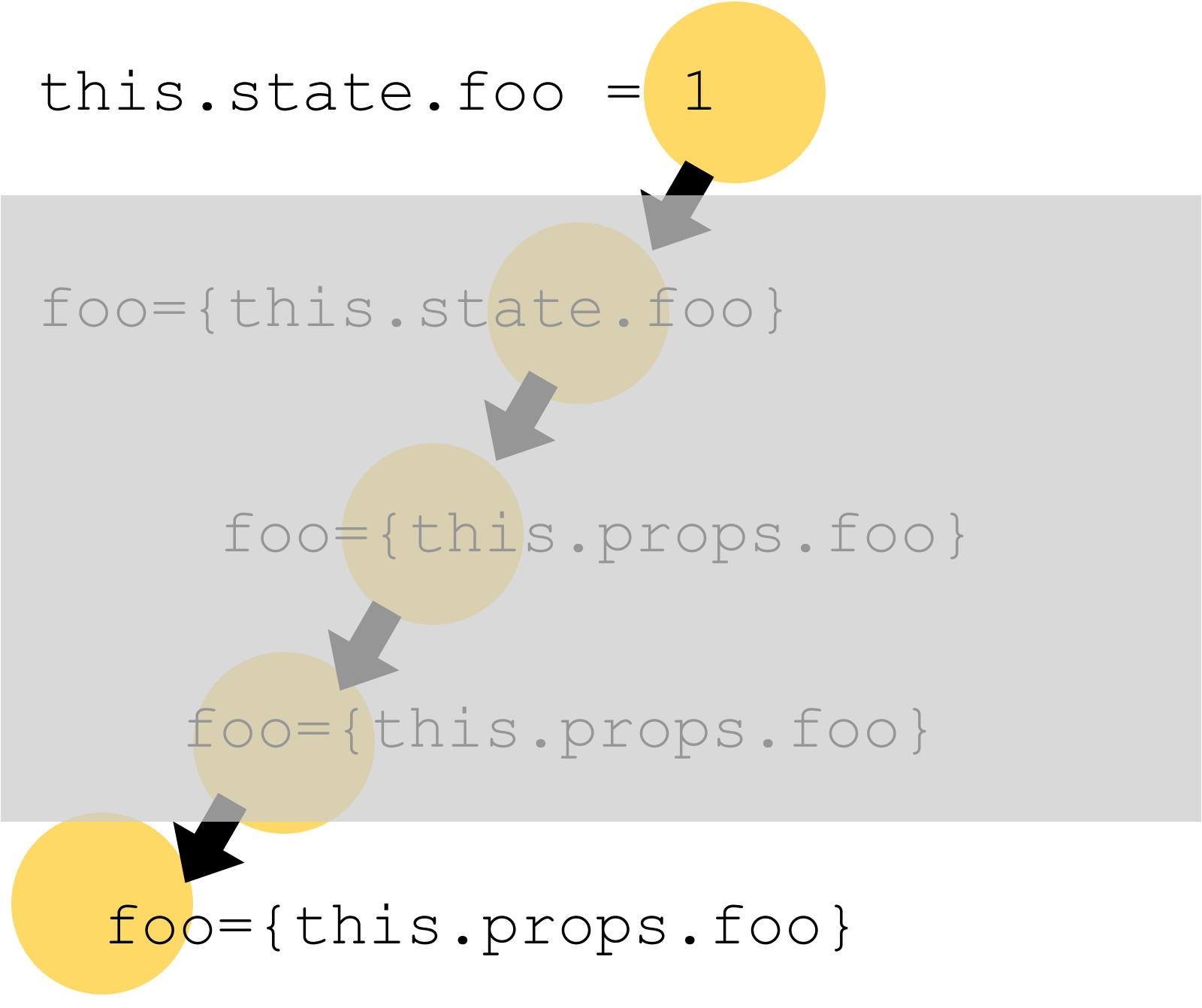
A way to share data in a big tree of components

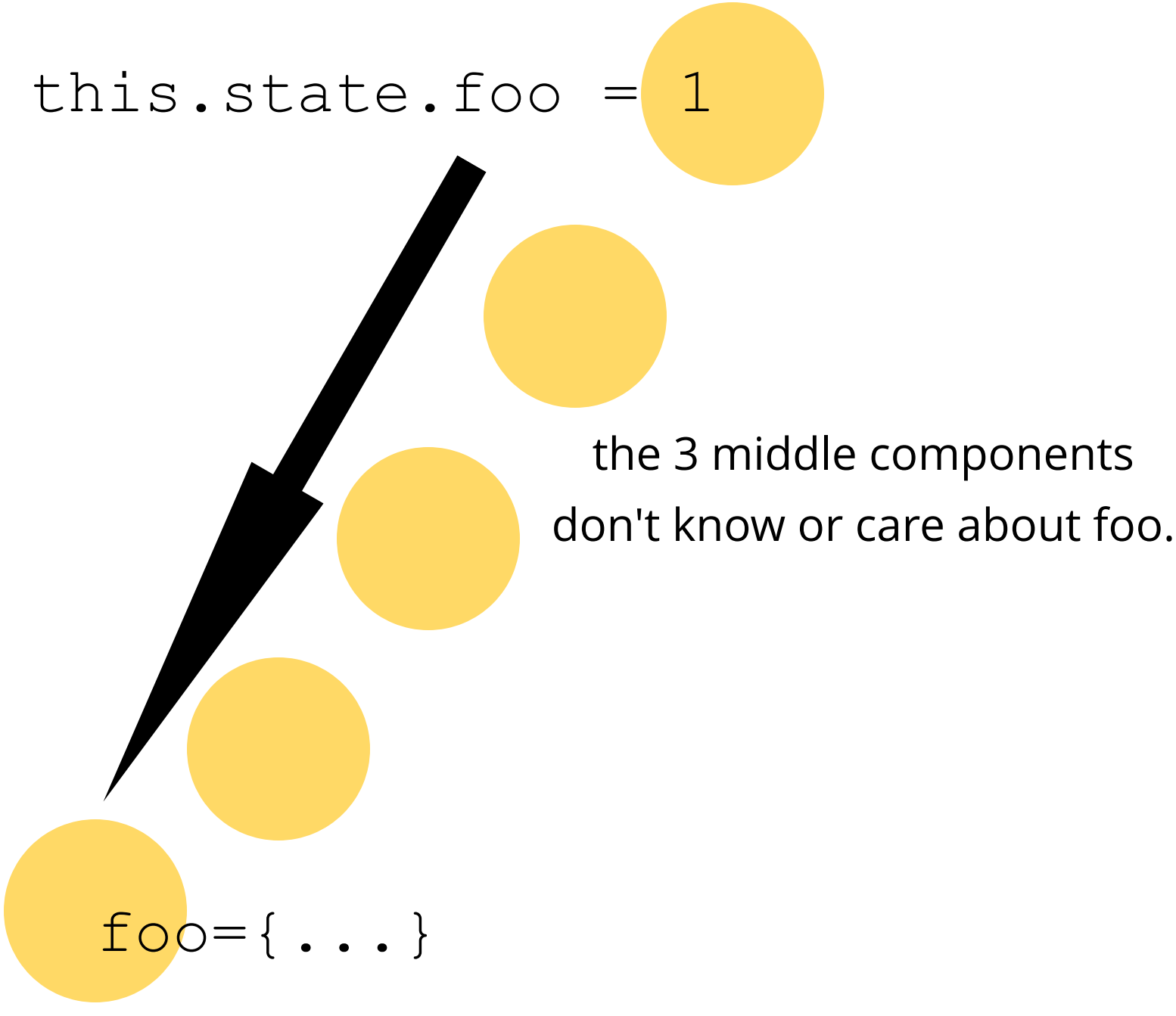
```
this.state.foo = 1
```




this.state.foo = 1
hey, I need to know `foo` !


this.state.foo = 1
foo={this.state.foo}
foo={this.props.foo}
foo={this.props.foo}
foo={this.props.foo}


this.state.foo = 1
foo={this.state.foo}
foo={this.props.foo}
foo={this.props.foo}
foo={this.props.foo}


this.state.foo = 1
the 3 middle components
don't know or care about foo.
foo={...}

```
const Context = createReactContext(defaultValue);
// <Context.Provider value={providedValue}>{children}</Context.Provid
// ...
// <Context.Consumer>{value => children}</Context.Consumer>
```

```
<ThemeContext.Provider value={this.state.theme}>
  {this.props.children}
</ThemeContext.Provider>

// and then later on in any child component

<ThemeContext.Consumer>
  {theme => (
    <h1 style={{ color: theme === 'light' ? '#000' : '#fff' }
      {this.props.children}
    </h1>
  )}
</ThemeContext.Consumer>
```

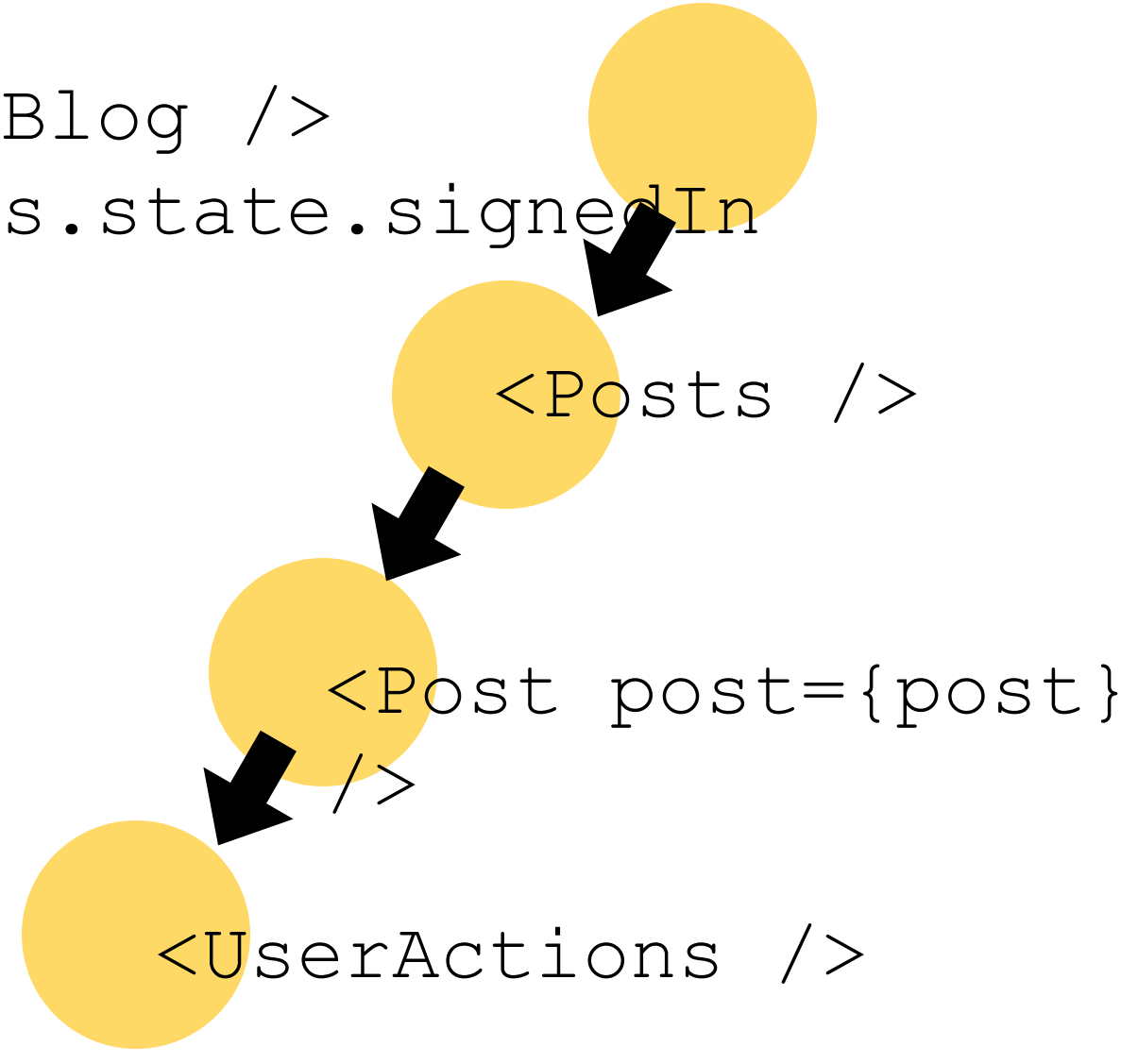
Creating a context gives you:

Provider

The component that makes a value accessible to all consumers in the component tree.

Consumer

The component that can read the value of a piece of state.


<MyBlog />
this.state.signedIn
<Posts />
<Post post={post}
/>
<UserActions />

```
<MyBlog />
this.state.signedIn
```

```
<Posts />
```

```
<Post post={post}
/>
```

```
<UserActions />
```

user actions needs to know if the user is signed in, to know if they can perform the actions

```
import React from 'react'

const AuthContext = React.createContext(false)

export default AuthContext
```

default value

```
import AuthContext from './auth-context'

  <div>
        <h1>Blog posts by Jack</h1>
        <AuthContext.Provider value={this.state.signedIn}>
          <Posts />
        </AuthContext.Provider>
      </div>
```

value for any consumers

any consumer in <Posts /> or below can now consume our auth context

```
import AuthContext from './auth-context'

<AuthContext.Consumer>
  {signedIn => (
    <div>signed in? {signedIn === true ? 'yes' : 'no'}</div>
  )}
</AuthContext.Consumer>
```

```
class MyBlog extends Component {
  state = {
    signedIn: false,
  }

  signIn = () => {
    this.setState({ signedIn: true })
  }

  signOut = () => {
    this.setState({ signedIn: false })
  }

  render() {
    return (
      <div>
        <header>
          {this.state.signedIn ? (
            <Fragment>
              <span>Signed in as jack</span>
              <button onClick={this.signOut}>Sign Out</button>
            </Fragment>
          ) : (
            <button onClick={this.signIn}>Sign In</button>
          )}
        </header>
        <div>
          <h1>Blog posts by Jack</h1>
          <AuthContext.Provider value={this.state.signedIn}>
            <Posts />
          </AuthContext.Provider>
        </div>
      </div>
    )
  }
}
```

using context in our blog post app

Exercise 21!

our app currently only exposes if the user is signed in or not. Can we make it so the user can sign in with a username, and then we show that username throughout the site?

Add a small input to the header so a user can "sign in" with their username, and then add the user's username to the `AuthContext` so it can be used in our application.

Exercise 22

Advanced React: fin!