codecademy

Components Interacting

Returning HTML Elements and Components

A class component's `render()` method can return any JSX, including a mix of HTML elements and custom React components.

In the example, we return a `<Logo />` component and a "vanilla" HTML title.

This assumes that `<Logo />` is defined elsewhere.

```
class Header extends React.Component {
  render() {
    return (
      <div>
        <Logo />
        <h1>Codecademy</h1>
      </div>
    );
  }
}
```

React Component File Organization

It is common to keep each React component in its own file, `export` it, and `import` it wherever else it is needed. This file organization helps make components reusable. You don't need to do this, but it's a useful convention.

In the example, we might have two files: **App.js**, which is the top-level component for our app, and **Clock.js**, a sub-component.

```
// Clock.js
import React from 'react';

export class Clock extends
React.Component {
  render() {
    // ...
  }
}

// App.js
import React from 'react';
import { Clock } from './Clock';

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>What time is it?</h1>
        <Clock />
      </div>
    );
  }
}
```

this.props

React class components can access their props with the `this.props` object.

In the example code below, we see the `<Hello>` component being rendered with a `firstName` prop. It is accessed in the component's `render()` method with `this.props.firstName` .

This should render the text "Hi there, Kim!"

```
class Hello extends React.Component {
  render() {
    return <h1>Hi there,
{this.props.firstName}!</h1>;
  }
}

ReactDOM.render(<Hello firstName="Kim"
/>, document.getElementById('app'));
```

defaultProps

A React component's `defaultProps` object contains default values to be used in case props are not passed. If a prop is not passed to a component, then it will be replaced with the value in the `defaultProps` object.

In the example code, `defaultProps` is set so that profiles have a fallback profile picture if none is set. The `<MyFriends>` component should render two profiles: one with a set profile picture and one with the fallback profile picture.

```
class Profile extends React.Component {
  render() {
    return (
      <div>
        <img src=
{this.props.profilePictureSrc} alt="" />
        <h2>{this.props.name}</h2>
      </div>
    );
  }
}

Profile.defaultProps = {
  profilePictureSrc:
'https://example.com/no-profile-
picture.jpg',
};

class MyFriends extends React.Component {
  render() {
    return (
      <div>
        <h1>My friends</h1>
        <Profile
          name="Jane Doe"
          profilePictureSrc="https://exam
ple.com/jane-doe.jpg"
        />
        <Profile name="John Smith" />
      </div>
    );
  }
}
```

props

Components can pass information to other components. When one component passes information to another, it is passed as *props* through one or more *attributes*.

The example code demonstrates the use of attributes in props. `SpaceShip` is the component and `ride` is the attribute. The `SpaceShip` component will receive `ride` in its props.

```
<SpaceShip ride="Millennium Falcon" />
```

this.props.children

Every component's `props` object has a property named `children`. Using `this.props.children` will return everything in between a component's opening and closing JSX tags.

```
<List>  // opening tag
  <li></li> // child 1
  <li></li> // child 2
  <li></li> // child 3
</List> // closing tag
```

Binding `this` keyword

In React class components, it is common to pass event handler functions to elements in the `render()` method. If those methods update the component state, `this` must be bound so that those methods correctly update the overall component state.

In the example code, we bind `this.changeName()` so that our event handler works.

```
class MyName extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: 'Jane Doe' };
    this.changeName
= this.changeName.bind(this);
  }

  changeName(newName) {
    this.setState({ name: newName });
  }

  render() {
    return (
      <h1>My name is {this.state.name}
</h1>
      <NameChanger handleChange=
{this.changeName} />
    )
  }
}
```

Call `super()` in the Constructor

React class components should call `super(props)` in their constructors in order to properly set up their `this.props` object.

```
// WRONG!
class BadComponent extends
React.Component {
  constructor() {
    this.state = { favoriteColor: 'green'
};
  }
  // ...
}

// RIGHT!
class GoodComponent extends
React.Component {
  constructor(props) {
    super(props);
    this.state = { favoriteColor: 'green'
};
  }
  // ...
}
```

`this.setState()`

React class components can change their state with `this.setState()`. `this.setState()` should always be used instead of directly modifying the `this.state` object.

`this.setState()` takes an object which it merges with the component's current state. If there are properties in the current state that aren't part of that object, then those properties are unchanged.

In the example code, we see `this.setState()` used to update the `Flavor` component's state from `'chocolate'` to `'vanilla'`.

```
class Flavor extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      favorite: 'chocolate',
    };
  }

  render() {
    return (
      <button
        onClick={(event) => {
          event.preventDefault();
          this.setState({ favorite:
'vanilla' });
        }}
      >
        No, my favorite is vanilla
      </button>
    );
  }
}
```

Dynamic Data in Components

React components can receive dynamic information from *props*, or set their own dynamic data with *state*. Props are passed down by parent components, whereas state is created and maintained by the component itself.

In the example, you can see `this.state` set up in the constructor, used in `render()`, and updated with `this.setState()`. `this.props` refers to the props, which you can see in the `render()` method.

```
class MyComponent extends React.Component
{
  constructor(props) {
    super(props);
    this.state = { showPassword: false };
  }

  render() {
    let text;
    if (this.state.showPassword) {
      text = `The password is
${this.props.password}`;
    } else {
      text = 'The password is a secret';
    }

    return (
      <div>
        <p>{text}</p>
        <button
          onClick={(event) => {
            event.preventDefault();
            this.setState((oldState) =>
({
              showPassword:
!oldState.showPassword,
            }));
          }}
        >
          Toggle password
        </button>
      </div>
    );
  }
}
```

Component State in Constructor

React class components store their state as a JavaScript object. This object is initialized in the component's `constructor()`.

In the example, the component stores its state in `this.state`.

```
class MyComponent extends React.Component
{
  constructor(props) {
    super(props);
    this.state = {
      favoriteColor: 'green',
      favoriteMusic: 'Bluegrass',
    };
  }

  render() {
    // ...
  }
}
```

Don't Change State While Rendering

When you update a React component's state, it will automatically re-render. That means you should never update the state in a render function because it will cause an infinite loop.

In the example, we show some bad code that calls `this.setState()` inside of its `render()` method.

```
class BadComponent extends
React.Component {
  constructor(props) {
    super(props);
    this.count = 0;
  }
  render() {
    // Don't do this! This is bad!
    this.setState({ count:
this.state.count + 1 });
    return <div>The count is
{this.state.count}</div>;
  }
}
```