Web Front-End Development
Week 9: React Component Architecture

Component architecture
https://reactjs.org/docs/components-and-props.html
Components are the main building blocks of all React apps.
React separates "concerns" (or features) into components. Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.
Instead of separating markup, styling, and logic as we do in regular web development, React couples these together into components.
Components are basically JavaScript classes or functions. They accept inputs and return React elements describing what should appear on the screen.
A component should ideally do one thing, have one responsibility. If it ends up growing, it should be broken up into smaller subcomponents.
Components can refer to other components and render them. This lets us use the same component abstraction for any level of detail. A button, a form, a dialog, a screen: in React apps, all those are expressed as components.
Like classes, always start component names with a capital letter.
A component must return a single root element in render().
You will see Components created 3 different ways:

- Function components: function App(){}
  - This is what create-react-app uses
- ES5 syntax: var App=React.createClass({render: function(){}});
- ES6 syntax: class App extends Component {render(){}}

Example:
https://repl.it/@aileenjp/hello-world-component

- App.js
  - Creates a component named App
  - return() determines what is returned when that component is rendered
  - return can only return one parent element
  - Note that the App component is exported at the bottom of the file. This allows it to be imported in another file, as we do at the top of index.js.
    - There can only be one default export per file. In React it's a convention to export one component from a file, and to export it is as the default export.
    - You could also have the export statement before the class
      - `export default function App(){…}`
- index.js
  - Note that the App component is imported at the top of the file
  - ReactDOM.render() renders the App component

Let's simplify the App component and see what's going on.
Simple JSX:

```
return (
  <div><h1>Hello World</h1></div>
  );
```

- It looks like it's returning HTML but it's actually JSX which gets translated to regular JavaScript at run time.
- The Babel transpiler compiles JSX down to JavaScript React.createElement() calls
    - React.createElement(type, props, children)

JavaScript equivalent for 1 element

```
return(
  React.createElement('div', null, 'Hello World')
  );
```

JavaScript equivalent for 2 elements

```
return(
  React.createElement('div', null,
    React.createElement('h1', null, 'Hello World')
  )
);
```

- Nested calls to React.createElement() for each element

JavaScript equivalent for 2 elements using className

```
return(
  React.createElement('div', null,
    React.createElement('h1', {className:'App'}, 'Hello World')
  )
);
```

- The className attribute is used instead of class
- App.css defines the styling for the class App

JSX equivalent with className

```
return (
  <div><h1 className='App'>Hello Class</h1></div>
  );
```

So you can see that JSX syntax is more concise and it is preferred in React.

Before we go further we should restructure our example so index.js renders the App component which renders the Character component.

Create a new file called Character.js for the Character component.

```
import React from 'react';
```

App is a function component so we're going to create Character as an ES6 class component.

ES6 class components extend `React.Component` and have a single method called `render()` and the body of the component will be in `render()`.

```
class Character extends React.Component{
  render(){
    return (
      <div><h1>I am the Character component</h1></div>
      );
  }
}
export default Character;
```

Now we'll update App to use the Character component.

```
import Character from './Character'
```

App will return the Character component.

```
return (<Character/>);
```

index.js stays the same.

This doesn't look useful right now but moving forward all of our logic for characters will be in the Character component.
The App component acts as the main entry point for the App.

Props

Properties are passed to a component as a single argument called props.
Props is an object that can have as many key/value pairs as you want to describe your data.
Components pass data to another component using an attribute that has the name of the property.
You define an attribute by giving it a name, just like you do with variables.

```
<Greeting firstName='Aileen' />
```

The Greeting component is passed `{firstName: 'Aileen'}` as the props object.

Components accept input through a properties object called "props".
A component can access props with whatever property name they were given using dot notation.
{props.*propertyname*}

```
{props.firstName}
```

The Greeting component accesses the prop firstName through the props object.

In class components we need to use "this" to access props so it would be
{this.props.*propertyname*}

You can also pass functions as props.

```
keepScore(){}

<Greeting score={this.keepScore} />
```

Props are read only. Components should never modify props.
Props can be used within a component or passed to other components.
All React components must act like pure functions with respect to their props.

- Pure functions do not attempt to change their inputs
- Pure functions always return the same result for the same inputs

We will talk about how to deal with data that changes in React next week.

You can also set default values for props using a class property called defaultProps.
`Greeting.defaultProps = {firstName: 'yo'};`

The default will be used if no data is passed to the prop firstName.

Example:
Props
Pass data to a component using props.
App.js

- Pass props to the Character component, property name "name"

Character.js

- Access the property "name" using dot notation `this.props.name`

Props can also be an object.
App.js

- Pass an object as the property "name"

Character.js

- Access the object keys through the property "name" through props using dot notation

The ability to reuse components is the whole reason behind components
App.js

- Return a component multiple times, each time passing it different data values as props.
- return() only takes one root element so we wrap the multiple elements in a

Styling
You can link to an external CSS file for CSS for the whole site.
React favors coupling styling within a component. It's designed to help make your visuals more reusable. The goal is to have your components be independent units where everything related to how your UI looks and works gets stashed there.
The way you specify styles inside your component is by defining an object whose content are the CSS properties and their values.

Because the CSS definitions are a key/value pair in an object, the values are strings so they're in quotes.

```
backgroundColor: "#ffde00"
```
```
<h1 style={{backgroundColor: "#ffde00"}}>
```
Here there are 2 sets of curly brackets – the inside one is the object, the outside one is for JSX to evaluate it. This method can get messy and cumbersome.
Instead you can define a variable and assign the object to it.
```
var headerStyle = {backgroundColor: "#ffde00"}
```
Once you have an object defined, you assign that object to the JSX elements you wish to style in your component by using the style attribute.
```
<h1 style={headerStyle}>
```
In React CSS properties are similar to when we used them in JavaScript.

- Single word CSS properties (like padding, margin, color) remain unchanged.
- Multi-word CSS properties with a dash in them (like background-color, font-family, border-radius) are turned into one camelcase word with the dash removed and the words following the dash capitalized.
    - For example, background-color becomes backgroundColor, font-family becomes fontFamily, and border-radius becomes borderRadius.
- In React if you write a style value as a *number*, then the unit `"px"` is assumed and you don't need quotes.
    - fontSize: 50

You can also keep your styles in a separate js file, export them, and then import them in the components where you want to use them.

Example:
Styling
Character.js

- Create an object with CSS properties and values
- Assign the object to the style attribute

Use props for styling
App.js

- Pass background color as a property (can be named anything)

Character.js

- Update the style object to use props