Web Front-End Development
Week 11: React Interactivity

Today we'll finally add some interactivity.
Components get data through props that are passed in, but these can never be changed. In order to store data in components that can be changed React uses state.

State

Props are used for data that doesn't change, state is used for data that does change.
State is how we store dynamic data, data that can change.
State has some similarities to props, but it is private and fully controlled by the component.
State is not passed into a component. State is defined in a component and that component is the only place that state can be changed.
State is not accessible to any component other than the one that owns and sets it, which makes it encapsulated.
State is an object with key/value pairs. Each key represents a piece of dynamic data you want to store and the value is its initial value.
You set up state in the component's constructor method after calling its super(props) method.

```
constructor(props) {
  super(props);
  this.state = {
      mood: 'great',
      hungry: false
}
```

To read a component's state property, use the expression {this.state.*propertyname*}
`{this.state.mood}`
A component can do more than just read its own state. A component can also *change* its own state by calling the function `setState()`.
`setState()` takes as a parameter an *object* that will update the component's state
`this.setState({ hungry: true });`
The `mood` part of the state remains unaffected.

To change the state values you must call `setState()` you can't update the state directly.
`setState()` takes an object, and *merges* that object with the component's current state. It then tells React that the state has change. React merges in the state update and then calls the **`render()`** method again to update the DOM. If there are properties in the current state that aren't part of that object, then those properties remain unchanged.

React may batch multiple `setState()` calls into a single update for performance so the state may be updated asynchronously, you should not rely on their values for calculating the next state.
A component may choose to pass its state down to its child components as props but the child components won't know or care where it came from, and since they're props, they won't be changed.

This is commonly called a "top-down" or "unidirectional" data flow. Any state is always owned by some specific component, and any data or UI derived from that state can only affect components "below" them in the tree.

The most common way to call `setState()` is to call it in a function, which we'll look at in a bit.

Events

React's approach to handling events is aimed at improved performance and browser compatibility.
Instead of using addEventListener() to handle events, React handles events within its components by assigning event listeners to DOM elements.
There are some syntactic differences:

- React events are named using camelCase, rather than lowercase.
  - onChange rather than onchange
- With JSX you assign a function as the event handler, rather than a string.
  - **`onChange={activateLasers}`** rather than onchange="activateLasers()"

The event handler can be a function in that component or one passed from another component as a prop.

Event Handlers

Functions can be used as event handlers.

```
handleClick(){}

<Greeting onClick={this.handleClick} />
```

When you pass an event handler as a prop there are two names that you have to choose, both in the component class that defines the event handler and passes it. These two names can be whatever you want. However, there is a naming convention that they often follow. You don't have to follow this convention, but you should understand it when you see it.

1. The name of the event handler itself.
   a. Based the name on the type of event you're listening for, such as "click"
   b. The event handler name should be the word "handle", plus the event type
   c. The event handler would then be named "handleClick".
2. The name of the prop that you will use to pass the event handler. This is the same thing as your attribute name.
   a. The prop name should be the word "on", plus the event type
   b. If you are listening for a "click" event, then the prop name would be "onClick"

The name `onClick` does *not* create an event listener when used on `<Greeting />`, it's just an arbitrary attribute name since `<Greeting />` is not an HTML-like JSX element, it's a component.

Names like `onClick` only create event listeners if they're used on HTML-like JSX elements. Otherwise, they're just ordinary prop names.

- You cannot return `false` to prevent default behavior in React. You must call `preventDefault` explicitly.
    - `e.preventDefault()` rather than `return false`
    - event handler functions are automatically passed the event
- In React the event is also passed to functions that are event handlers
    - Arrow function
        - `<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>`
        - must explicitly pass the event(e)
    - Function.prototype.bind
        - automatically passes the event
        - `<button onClick={this.deleteRow} Delete Row</button>`
            - In the constructor you need to bind this method
            - `this.deleteRow = this.deleteRow.bind(this);`

```
  constructor(props) {
    super(props);
    this.state = { mood: 'good' };
    this.toggleMood = this.toggleMood.bind(this);
  }

  toggleMood(e) {
    const newMood = e.target.value;
    this.setState({ mood: newMood });
  }

render() {
return <div>
<h1>I'm feeling {this.state.mood}</h1>
<select onChange={this.toggleMood} />
</div>
}
```

Due to the way that event handlers are bound in JavaScript, `this.toggleMood()` loses its `this` when it is used in render(). Therefore we have to bind `this.toggleMood()` to `this` in the constructor method.
In React, whenever you define an event handler that uses `this`, you need to add `this.methodName = this.methodName.bind(this)` to your constructor function.

Here is how the component's state would be set:

1. A user triggers an onChange event by selecting a value in the select list.
2. The onChange event from is being listened for on the select element.

3. When the event fires it calls the *event handler* function assigned to it -- `this.toggleMood`
4. `toggleMood(e)` takes an *event object* as an argument. Using the event object you can access the target's value. The target is the element that the event was fired on.
5. That value is saved in the constant `newMood`
6. `this.setState()` is sent the object `{ mood: newMood }` to change the value of `mood` to the value in `newMood` and the component's state is changed.

Any time that you call this.setState(), this.setState() AUTOMATICALLY calls .render() as soon as the state has changed.

Think of `this.setState()` as actually being two things: `this.setState()`, immediately followed by `.render()`.

That is why you can't call `this.setState()` from inside of the `.render()` method. `this.setState()` *automatically* calls `.render()`. If `.render()` calls `this.setState()`, then an infinite loop is created.

Example:
Favorite #2 (App2.js, Favorite2.js)

- Adding styling to both App and Favorite components
- Add state to Favorite
    - The constructor() method defines state to include the number of votes
        - Constructor methods always take in props
        - Constructor methods must always first call the super() constructor
- Added a method to increase the number of votes by calling setState()
    - {this.vote} is an event handler and must be bound in the constructor
- render() now includes votes(state), name(props), and a button
- The button uses the onClick event to call the vote function to increase the number of votes