React State

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>;
  }
}
```