State!

We've learned how to display data, and pass data as props. What we haven't learned is how to manipulate data! This is where state comes in.

```
const handleAction = someEvent => {
  console.log("Some event : ", someEvent);
};

const Child = props => {
  return <button onClick={props.onAction}> Click Me! </button>;
};

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

Suppose we wanted to start counting upward when clicking the child's button. There are two ways that you'll see as an approach to solving this problem:

- Classes
- Hooks

State: Classes

We're not going to spend too much time on classes, but since they're not deprecated, we're definitely going to talk about them for a brief moment. For reference to what's happening, the following code is from `ParentClassComponent.jsx` in the `example-state` project:

```
import React from "react";
import { Child } from "../Child";


// Instead of defining our component functionally like we've done before,
// we define our component as a class extending `React.Component`
export class ParentComponent extends React.Component {

  // We need a constructor to build our component (which takes in our props)
  constructor(props) {
    super(props);  // and then call our superconstructor.

    // We assign our state variables by using `this.state`.
    // Any key within state's object will be accessible
    this.state = {
      timesPressed: 0
    };

    // When we want to have functions access state, we need to bind them like so
    this.handleAction = this.handleAction.bind(this);
  }


  // Here is a handle action function that we'll send to our child component.
  handleAction(action) {
    console.log("Handling the action! ", action);

    // To manipulate state, we can use `this.setState`,
    // and pass in a new updated state object
    // (here, we're adding one to our timesPressed)
    this.setState({
      timesPressed: this.state.timesPressed + 1
    });

    console.log("TIMES PRESSED? ", this.state.timesPressed);
  }


  // And below, we need a way to work with our JSX, so we use the `render` function.
  // You can put anything inside this function (i.e. anything before the return),
  // it just needs to return JSX at the end.
  render() {

    return (
      <>
        <Child onClick={this.handleAction} />
        <div>That dang button got clicked {this.state.timesPressed} times </div>
      </>
    );
  }
}
```

Each Component's State is Independent!

Regardless of whether or not you use multiple of the same component, they will each have their own state. Try replacing `<ParentComponent />` in the `ReactDOM.render` with `<ParentChildParty />` in the `index.js` file.

```
//... above code removed

const ParentChildParty = () => {
  return (
    <>
      <h1>Parent 1</h1>
      <ParentComponent />
      <br />
      <h1>Parent 2</h1>
      <ParentComponent />
      <br />
      <h1>Parent 3</h1>
      <ParentComponent />
      <br />
      <div>
        Clearly, every single instance of "ParentClassComponent" has its own
        state. Try opening the devtools and looking around
      </div>
    </>
  );
};

ReactDOM.render(<ParentComponent />, document.querySelector("#root"));

```

Each `ParentComponent` has its own state that is accessed individually!

State's Asynchronicity:

Asynchronicity, not a word, but it sounds cool, and `setState` is an asynchronous function. Let's try adding a console log to read from state immediately after we set it:

```
...
 handleAction(action) {
   this.setState({
     timesPressed: this.state.timesPressed + 1
   });

   console.log("TIMES PRESSED? ", this.state.timesPressed);
 }
 ...
```

What ends up happening is that we wind up getting a console log in our browser that's logging the previous state, while our browser itself is rendering what the actual state is. For example, suppose `this.state.timesPressed` was `2`, if we called `handAction`, we'll then asynchronously call `setState`, and then move onto `console.log`, where `2` will get printed to the console while state is being updated to 3!

Keeping your code clean with setState:

To keep your code clean, you can just as easily pass a function to your set state that returns a state object of whatever you wish to update. The function by default takes in `(state, props)`. That is, it takes in the previous state and props (prior to being updated). You can see an example of this in `ParentClassWithStateFunction.jsx` in the corresponding code.

```
...
  addOneToState(state, props) {
    console.log("State?", state);
    console.log("Props???", props);
    return {
      timesPressed: this.state.timesPressed + 1
    };
  }

  handleAction(action) {
    this.setState(this.addOneToState);
  }
...
```

More State Functions

You can just as easily create other methods that work on the same state variables, that is, it's somewhat pointless having only a single function that works on a state variable. You'll often want to have multiple functions referencing the same state variables to tie your app together, such as a reset function for our button counter (an example can be seen in the `ParentClassWithResetFunction.jsx` file:

```
...
 constructor(props) {
    super(props);
    this.state = {
      timesPressed: 0
    };

    this.handleAction = this.handleAction.bind(this);

    // MAKE SURE TO BIND YOUR ADDITIONAL FUNCTIONS!
    // This will constantly turn around and bite you (it does for me)
    this.resetTimesPressed = this.resetTimesPressed.bind(this);
  }

  addOneToState() {
    console.log("inside add1 to state");
    return {
      timesPressed: this.state.timesPressed + 1
    };
  }

  // added resetStateToZero function
  // this just sends a `timesPressed: 0` object to state!
  resetStateToZero() {
    console.log("inside reset state to 0");
    return {
      timesPressed: 0
    };
  }

  handleAction(action) {
    console.log("Handling the action! ", action);

    this.setState(this.addOneToState);

    console.log("TIMES PRESSED? ", this.state.timesPressed);
  }

  resetTimesPressed(action) {
    this.setState(this.resetStateToZero);
  }
  ...
```

REITERATING: In order to make sure that you don't find yourself in a pickle (i.e. if you use a class component with constructors), make sure you always bind your functions! (Try removing a `this.<functionName> = this.<functionName>.bind(this)` from the constructor)

Multiple State Elements:

What happens if you have more than one element in your state? You likely will want more than one element in your state, such as a text field (example can be seen in `ParentClassMultipleStateItems.jsx`):

```
export class ParentComponent extends React.Component {
  constructor(props) {
    console.log("This instatiates the component and brings in props", props);
    super(props);
    this.state = {
      timesPressed: 0,
      value: ""
    };

    this.handleAction = this.handleAction.bind(this);
    this.resetTimesPressed = this.resetTimesPressed.bind(this);

    // BINDING THE NEW FUNCTION THAT WILL HANDLE OUR TYPING IN THE TEXT FIELD
    this.handleTypeAction = this.handleTypeAction.bind(this);
  }

  // Removed handleAction/ResetTimesPressed/addOneToState/resetStateToZero
for space

  // UPDATING THE VALUE (just like with addOneToState and resetStateToZero)
  updateTheValue(event) {
    return {
      value: event.target.value
    };
  }

  // Handling the type action!
  // Notice that we're passing event into this.updateTheView
  // which is returning an object, and then being passed into setState?
  // First class functions rule!
  handleTypeAction(event) {
    this.setState(this.updateTheValue(event));
  }


  // Below, we have our text box in our render function!
  render() {
    return (
      <>
        <Child onClick={this.handleAction} />
        <div>That dang button got clicked {this.state.timesPressed} times </div>
        <div>
          <input
            type="text"
            value={this.state.value}
            onChange={this.handleTypeAction}
          />
        </div>
        <GrandChild onClick={this.resetTimesPressed} />
      </>
    );
  }
}
```

How to Update State (and not)

When you have more and more complex states, you'll often have compartmentalized data in the form of an object, or an array, etc (may as a user object with a username and password?). When updating your state, I'm sure you've noticed that you only insert the portion of state that you'd like to update. This works great at the top level. That's because state is updated with a shallow copy. If you try to do this with objects in state, it will not deep copy. Take a look at the below code (working components in `ParentComponentShallowDeepState.jsx`). When we update username or password, we're not passing in its corresponding value (that is, on username, we're not also passing in password, and vice versa).

Try running the below code. It doesn't necessarily make the most sense for the text boxes (since they don't rerender the values), so we have the username and password displaying within a div after each:

```
export class ParentComponent extends React.Component {
  constructor(props) {
    console.log("This instatiates the component and brings in props", props);
    super(props);
    this.state = {
      timesPressed: 0,
      value: "",
      user: {
        username: "",
        password: ""
      }
    };

    this.handleAction = this.handleAction.bind(this);
    this.resetTimesPressed = this.resetTimesPressed.bind(this);
    this.handleUsernameAction = this.handleUsernameAction.bind(this);
    this.handlePasswordAction = this.handlePasswordAction.bind(this);
  }

  // handle times pressed/reset with corresponding functionsremoved

  updateUsername(event) {
    return {
      user: {
        username: event.target.value
      }
    };
  }

  updatePassword(event) {
    return {
      user: {
        password: event.target.value
      }
    };
  }
  handleUsernameAction(event) {
    this.setState(this.updateUsername(event));
    console.log("State after setting username", this.state);
  }

  handlePasswordAction(event) {
    this.setState(this.updatePassword(event));
    console.log("State after setting password", this.state);
  }

  render() {
    return (
      <>
        <Child onClick={this.handleAction} />
        <div>That dang button got clicked {this.state.timesPressed} times
</div>
        <div>
          Username:
          <input
            type="text"
            value={this.state.user.username}
            onChange={this.handleUsernameAction}
          />
          <div> Username typed: {this.state.user.username} </div>
        </div>
        <br />
        <br />
        <div>Password:</div>
        <input
          type="password"
          value={this.state.user.password}
          onChange={this.handlePasswordAction}
        />
        <div> Password typed: {this.state.user.password} </div>

        <br />
        <GrandChild onClick={this.resetTimesPressed} />
      </>
    );
  }
}
```

I'm sure you noticed that the second you started typing in password, the username went away (and the same goes for username). To avoid this problem, you only need to either pass in the part of state you wish to keep:

```
...
updateUsername(event) {
  return {
    user: {
      username: event.target.value,
      password: this.state.user.password
     }
  };
}
```

```

updatePassword(event) {
  return {
    user: {
      password: event.target.value,
      username: this.state.user.username
    }
  };
}
...
```

However, it's pretty likely that you'll have more than just two keys within your object, so you can use the spread operator to get around that and spread whatever object you'd like (in our case, it's `this.state.user`):

```
...
updateUsername(event) {
  return {
    user: {
      ...this.state.user,
      username: event.target.value,
     }
  };
}

updatePassword(event) {
  return {
    user: {
      ...this.state.user,
      password: event.target.value,
    }
  };
}
...
```

What's happening above is that we're essentially copying `this.state.user` and then overwriting a specific key. **CAVEAT:** Make sure you spread your object first. If you spread it last, then you'll be overwriting your new data with your old data! Go to `ParentComponentShallowDeepState.jsx` to see a working example. Try placing the the spread after your updated values. What do you see?

Cleaning Up:

We've seen that we can update and manipulate our state with the setState methods, but our components are getting incredibly large. Keeping file sizes below a hundred lines is definitely nice to do. Let's clean our code up a little bit. We can start with making use of arrow functions! Because arrow functions don't bind to their own `this` and move up a level, they implicitly bind to the class itself (so we don't have to remember all of those `this.whatever = this.whatever.bind(this`)).

So now we can remove the constructor entirely, set our state by just creating a state variable and turning our functions into arrow functions (seen in `ParentClassClean.jsx`)!

```
import React from "react";
import { Child } from "../Child";
import { GrandChild } from "../GrandChild";

export class ParentComponent extends React.Component {
  // get rid of that constructor
  state = {
    timesPressed: 0,
    value: "",
    user: {
      username: "",
      password: ""
    }
  };

  // change all
  addOneToState = () => ({
    timesPressed: this.state.timesPressed + 1
  });

  resetStateToZero = () => ({
    timesPressed: 0
  });

  updateUsername = event => ({
    user: {
      username: event.target.value,
      password: this.state.user.password
    }
  });

  updatePassword = event => {
    return {
      user: {
        password: event.target.value,
        username: this.state.user.username
      }
    };
  }

  handleAction = action => {
    this.setState(this.addOneToState);
  };

  resetTimesPressed = action => {
    this.setState(this.resetStateToZero);
  };

  handleUsernameAction = event => {
    this.setState(this.updateUsername(event));
    console.log("State after setting username", this.state);
  };

  handlePasswordAction = event => {
    this.setState(this.updatePassword(event));
    console.log("State after setting password", this.state);
  };

  render() {
    return (
          <>
        <Child onClick={this.handleAction} />
        <div>That dang button got clicked {this.state.timesPressed} times
</div>
        <div>
          Username:
          <input
            type="text"
            value={this.state.user.username}
            onChange={this.handleUsernameAction}
          />
          <div> Username typed: {this.state.user.username} </div>
        </div>
        <br />
        <br />
        <div>Password:</div>
        <input
          type="password"
          value={this.state.user.password}
          onChange={this.handlePasswordAction}
        />
        <div> Password typed: {this.state.user.password} </div>

        <br />
        <GrandChild onClick={this.resetTimesPressed} />
      </>
    );
  }
}
```