MobX Notes:

MobX is a great compromise between React's default state management and something more complex like Redux.

As an exercise we made a Counter app. In our App.js component:

```
<p>0<p>
<button>Increment</button>
<button>Decrement</button>
```

Then for our state we have ——> this.state = { value: 0 }
To display this we would do ——> {this.state.value}

For each button we add an onClick with increment and decrement functions ({this.increment} and {this.decrement}) that each change the state using setState()

If we want multiple counters or other child components, we would need to start passing state down as props and calling back to App.js to change state from our child components. In this case we create a Counter component and pass the functions from App.js and the value in state down into it as props. This is standard React, but to set ourselves up for more complex applications, we bring in a library like MobX.

yarn add mobx mobx-react (do this for any projects that you want to add mobx or any other external libraries to)

MobX works with an idea of "observers" and "observables". An observable is something that we want MobX to watch for us and observers are the things that are watching it. They will change dynamically with changes in the observable.

Our first step is removing React's built in state. We change this.state = { value: 0 } to simply this.value = 0.
Next, inside of our increment() and decrement() functions we can remove setState() functions and replace them with this.value += 1 and this.value -= 1

At this stage our app isn't working. We have to import MobX and set up our observers and our observable.

```
import { decorate, computed, observable } from 'mobx'
import { observer } from 'mobx-react'
```

After we've done this import, we can tell our App component that it has MobX capabilities and that 'value' should be observable, since this is the information that will be changing. We do this at the bottom of the screen right above our export with the decorate() method.

```
decorate(App, {
  value: observable
})
```

Now we go to our Counter (or whatever component we want to be observed) and:
import { observer } from 'mobx-react'

Here when we export we have to say export default observer(Counter) since the Counter component will be observing changes in 'value' in App.js. We do this in App.js as well because it also has to observe its own changes.

Up until this point all we've done is replace React's state with MobX. Doing it this way means we still have to pass props and functions down to child components, so our level of organization isn't improved much. If we want to avoid adding a bunch of props and functions to each instance of a component, like Counter, we can create a new file that is a plain old JavaScript class (no props or super(props)) and use that as a centralized location to store all of our application's data. This is how we stay organized.

We can take everything from App.js except for the render and anything else that App.js directly needs and put it into this new file (our 'state' and the functions that manipulate it). Then do all of the same MobX imports. We'll call this a store.

We can take the decorate() method out of App.js and put it inside our store because it no longer contains anything that needs to be watched. We only want MobX to observe what's inside the store.

Instead of exporting the entire class of Store, we create a new instance variable of the Store class and export that. This is because we want the entire app to be on the same page. This variable is referred to as a "singleton" because it is created once and never changes.

*I'm not entirely clear what exporting the entire class would do, but I think it means that at least in the case of the counter, each counter component would have the same properties and functionality, but they would function independently rather than in sync with one another. I know that exporting a singleton ensures that they all do the same thing.

**Or maybe: Every component that imports the Store class would then need to create its own instance of the class to pass around as a variable. Even still I don't know if this would work. I've been experimenting and getting a lot of errors; it doesn't seem to work. Check with Gavin or Mark.

Every component that is an observer now needs to:
import myStore from ‘./Store’

Now everywhere our components say this.props we can replace it with myStore
ex: this.props.value ——> myStore.value

We can apply all of this same logic to Minesweeper. Instead of our state, we create a file that is a class called Game (this word summarizes what all of our data and functionality is centered around) and export a singleton. Everything in App.js outside of the render (again, with the exception of some of it’s functions that directly affect the user interface) will go into the Game class. The Cell, Board and App components all become observers.

From here all we need to do is change our wording around. We do away with any setState()’s we have and instead directly assign values to our singleton’s properties
(ex: this.game.playing = true or this.game.api = response.data
Also, note: We changed this.state = {} to this.game = {} but we could have just done without that and instead had everything in it like this.playing, this.api etc and then in place of our setState() we could just say this.playing = true or this.api = response.data etc etc.)

We can also move in the checkCell() and flagCell() functions from the Cell component to our Game class and remove these functions from the Cell instances. We would still have some props that will be used in these functions (row and col were created by mapping the board in Game) but all in all things are much more tidy.

In summary: We have a centralized location where all of our data is stored, make it observable, export that data as a singleton and then make any component we want to change with it an observer. We still use all of the component structure but we no longer use React’s built-in state management.