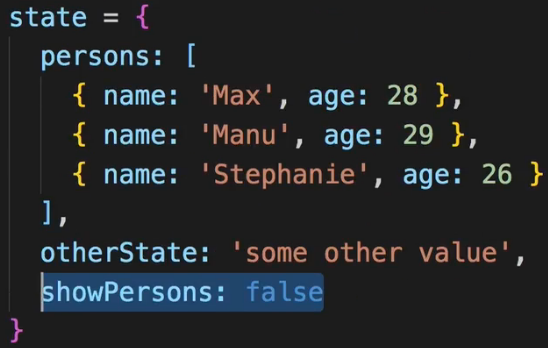
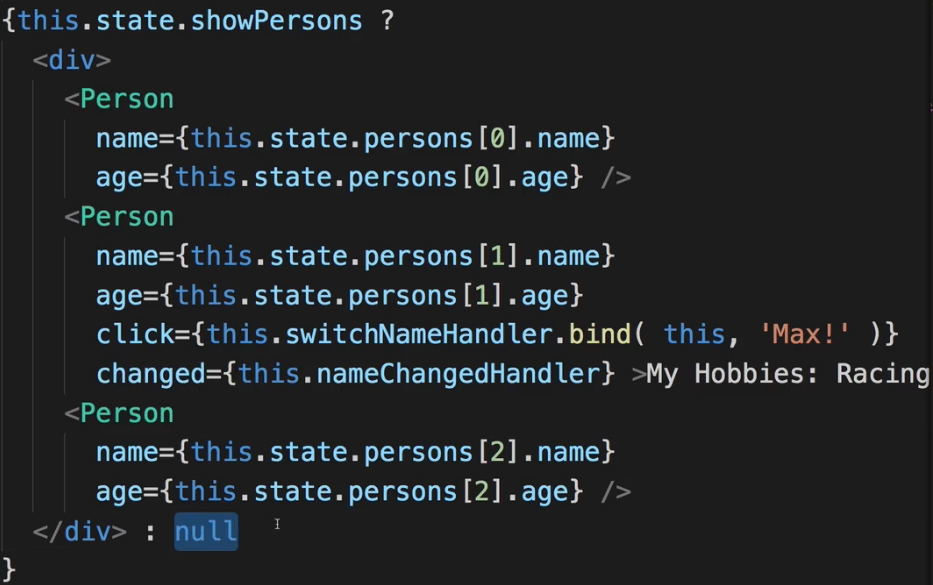
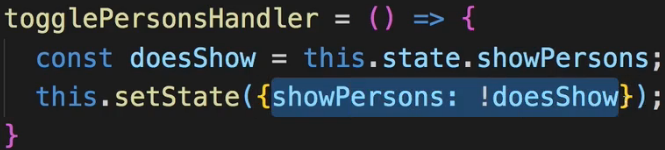
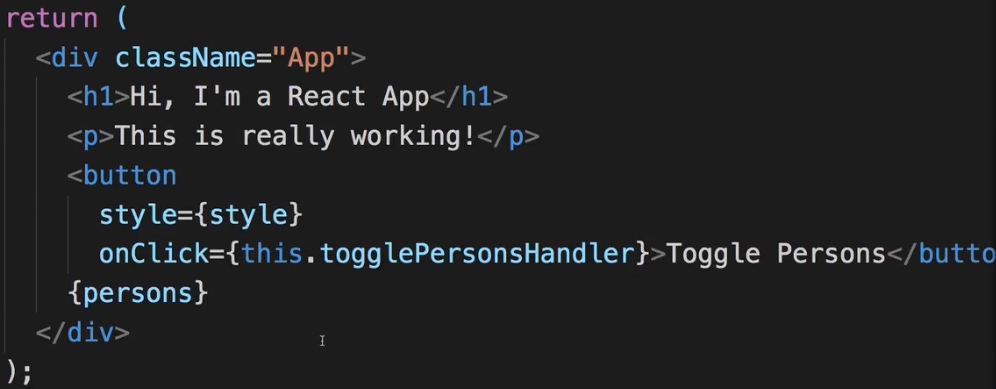
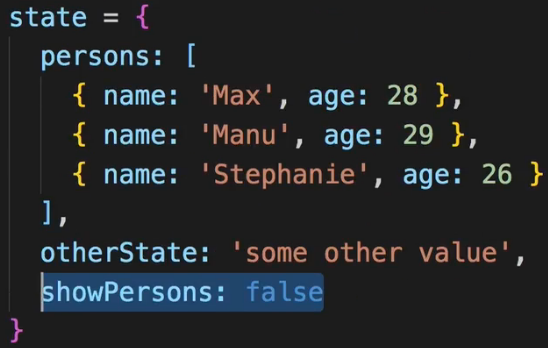
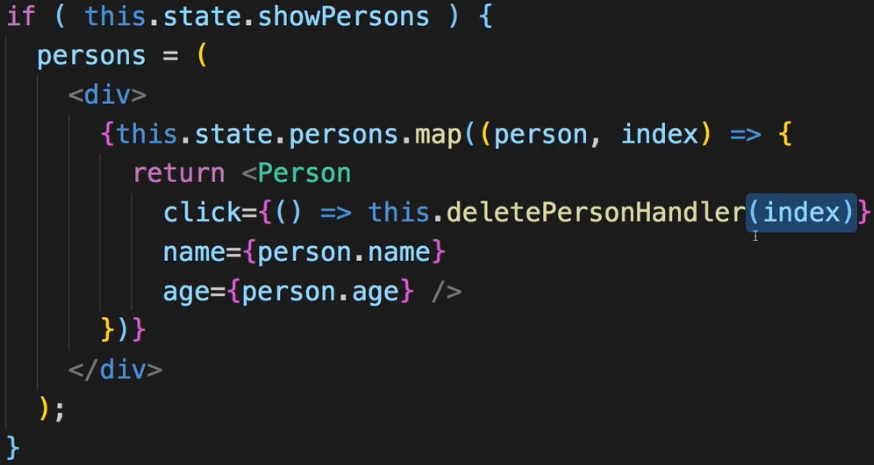
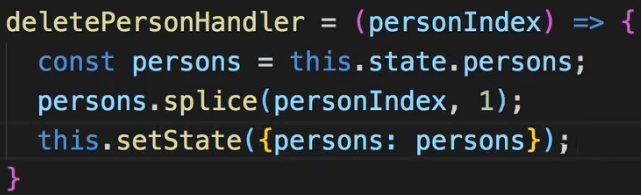
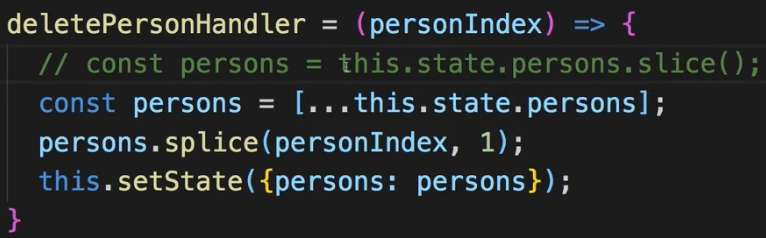
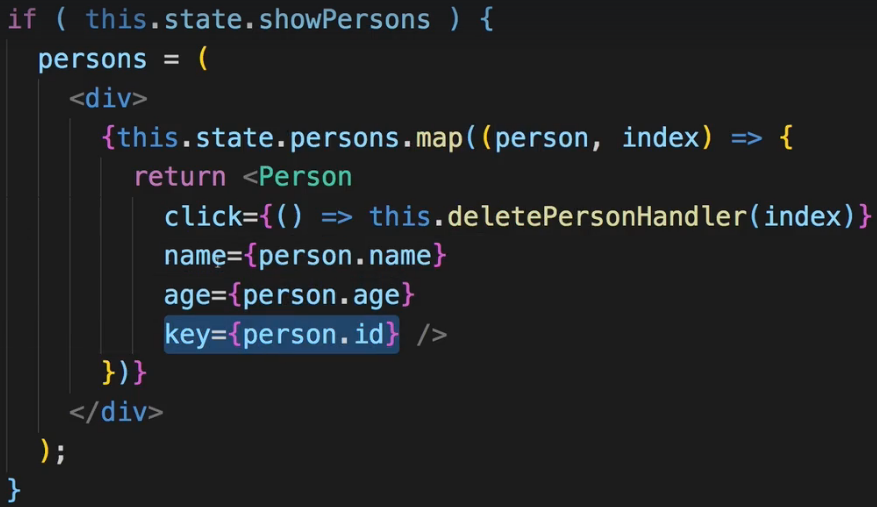
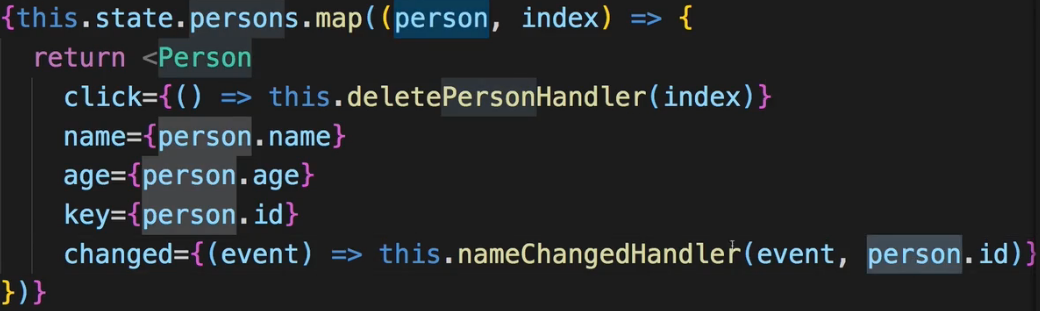
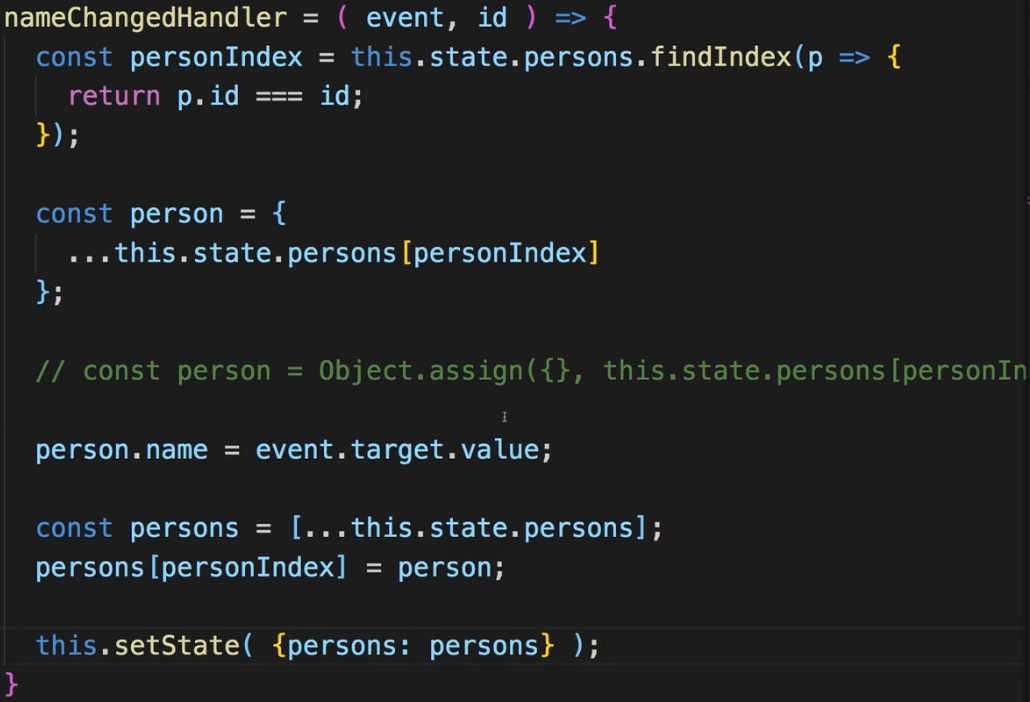
**Section 4 done: 11/12 Working with Lists and Conditionals  
Rendering Content Conditionally**  
\* 1. We can wrap all the Person list items in a div to show/hide them.  
\* 2. We add a state showPersons: false by default  
  
\* 3. We create a togglePersonsHandler method and give it to the button.  
\* 4. Wrap the div in {} so you can use JavaScript expressions. You can only use simple statements, no block statements so if () {} won’t work - use a ternary operator instead.  
\* We can use null in the ternary operator.  


  
\* Now writing conditionals like this (ternary operator) though does work but can also lead to confusing JSX code if you have a lot of possibly even nested checks.  
\* Let’s see an alternative to ternary operators.  
**Handling Dynamic Content “The JavaScript Way”**  
\* There is 1 important thing to keep in mind - when React renders something to the screen, when it decides that it needs to update the screen, it executes this **render** method and NOT JUST the return expression there. So everything inside the render method gets executed whenever React re-renders this component and therefore we can take advantage of this - we can add some code here before we return something.   
  
  
\* This way we keep it clean.  
\* This is actually the preferred way of outputting conditional content.  
Outputting Lists  
\* The persons array in our state should be our single source of truth. This should be our data source. In bigger applications and real applications this might be populated with data you fetch from a server.  
  
\* Let’s learn how we can output lists of data - arrays basically, and how we can also then interact with them, update them and correctly change arrays in our state.

\* We use the {} to render something dynamic within our JSX code.  
\* We need to convert this array of JavaScript which is not valid JSX to valid JSX.  
\* Vanilla JavaScript offers us a function to convert arrays - .map().  
\* **.map()** simply maps every element in the array into something else. It does this by executing a method on every element. It’s not valid JSX, that’s why it’s inside {}.  
\* And now in here you have to `return` something. You should return what you want to map this item into.  
\* This map function conveniently simply returns a new array. And since I just print this new array like this inside JSX, React will basically just try to take the individual elements of this new array and render them to the DOM if they are valid JSX. So our goal is to return JSX in here so that every element in the persons array gets mapped into a JSX representative of it.  
  
\* We do get a warning about using a unique key prop - that is important.  
**Lists & State**  
\* Now let’s see how we can manipulate this array.  
\* Now if we want to apply an event method to the specific clicked component, we can pass a second argument to the .map() function - the **index**.  
\* We could use the .bind instead of the arrow function.  
  
\* So now we can receive this in the deletePersonHandler.  
\* **array.splice(remove-index, how-many-to-remove)**  
  
\* Arrays and Objects are reference types - so it is only holding a pointer, we just change the element it was pointing to.  
\* **This approach has a flaw and we’ll fix it soon**.  
**Updating State Immutably**  
\* The flaw of this approach is that in JavaScript Objects and Arrays are reference types. So when I get persons from my state as I do here, I actually get a pointer to the original state.  
=> If I then splice it here, I already mutate this original data and whilst it does work without throwing an error, this is not really how you should do it, this can lead to unpredictable apps and definitely a bad practice.  
=> A good practice is to create a copy of your persons array before manipulating it.  
=> And a simple way of doing this is by calling the **.slice()** method.  
=> **.slice() without arguments** simply copies the full array and returns a new one which is then stored here and you can now safely edit this new one and then update the React state with your new array.  
\* An alternative to this approach would be to use an ES6 feature - the **spread operator**  
=> It spreads out the elements in this array into a list of elements which simply then gets added to this array so now we have a new array with the objects from the old array but not the old array itself.  
  
=> You should always update state in an immutable fashion. So without mutating the original state.  
**Lists & Keys**  
\* `**key**` is actually a default property React expects to find on an element no matter if it’s a custom component or a default HTML element - which you render through a list. So basically by mapping an array into JSX elements. This key property helps React update the list efficiently. Consider the case we have here - that we delete elements from the list. Of course it works, but behind the scenes React needs to find out what it actually needs to adjust in the overall DOM. And we’ll dive deeper into what React exactly does, in a specific section. But basically what it does is - it has something called a **virtual DOM** where it compares what it would render now if it were to call the render method now or if it did actually adjust the real DOM with the result of the render method, to the previous DOM it rendered. And for lists it of course needs to find out which elements changed. **So by default it will just re-render the whole list** and for a long list, this is very inefficient. That is why we should assign a key property to allow React to keep track of the individual elements so that it has a clear property it can compare between the different elements to find out which elements changed and which didn’t so that it only re-renders the elements which did change and not the whole list.  
=> What do we assign it? Typically you probably have some ID on your elements if you get them from a database or something like that.  
=> Here we could use `index` because after all index changes for every element in the array. That’s right but index also is part of the list itself - if the list changes, every element will receive a new index - at least every element after the change.  
=> So `index` is not a good `key` - in the end it will not really help React.  
=> So we should really try to have a unique identifier so we can simply add an ID into our persons elements in the state. It doesn’t have to be a number, it can be anything unique.  
 

**Flexible Lists**  
  
\* **.find()** - this again is a default JavaScript method which gives us this person.  
\* **.findIndex()** - we can also use this to find an element in an array but then get the index of that element and yes we could’ve passed the index directly into this function - I just want you to show you this alternative approach taking advatange of the ID.  
=> Just as .map() this will execute a function on every element in the array. Unlike map I don’t map this element into something new - instead here I have to return true/false depending on whether this is the element I was looking for or not.  
\* Now just as before - it is a good practice to not mutate the state directly and a person in the persons array is a JavaScript object so it’s a reference type so we shouldn’t mutate them directly because we only get a pointer when we reach out to the person here and hence we would mutate the original object to which this pointer points.  
=> So the better approach is to create a new JavaScript object and then use this `**spread** **operator**` in front of the object and fetch it. It will distribute all the properties of the object we fetch here into this new object we’re creating here.  
  
\* And here we also got an alternative approach which I won’t use - to use an **Object.assign()** - a default JavaScript function where we pass an empty object as the first argument and then the object which properties we want to assign into it as the second argument.  
\* So now that we have the person, now we want to update the person.name and I can do this (person.name =) because I got a copy, I’m not manipulating the original object now. And I can set it to event.target.value - just as we did before.  
\* Now I need to update the array of course and I want to update the array at this position I fetched here. So just as before I can now get my persons - the whole array - with the spread operator or with slice and now I can update it at one position.  
=> Now we got an updated persons array and this finally allows us to set the state to the updated persons array which is the copy of the old array where we updated 1 element with the updated person/we adjusted the name.  
  
=> A lot of code but the best way of doing it - the most efficient way without mutating the state.  
=> So now we got a truly flexible component here, a truly flexible application taking advantage of modern features to render this list efficiently and interact with it efficiently. Super important.  
Conditional Rendering: <https://reactjs.org/docs/conditional-rendering.html>  
Lists & Keys: <https://reactjs.org/docs/lists-and-keys.html>

**Assignment 2: Time to Practice - Lists & Conditionals**  
Conditional Rendering: <https://reactjs.org/docs/conditional-rendering.html>  
Lists & Keys: <https://reactjs.org/docs/lists-and-keys.html>