the Master Course

{CODENATION}

Common situations when using React.

{CODENATION}

React.js

Learning Objectives

To understand that the state object cannot be mutated directly.

To be familiar with working with inputs in React.

Updating state.

The state object should never be mutated directly. As we have seen, the only way of changing the state of a component should be through the use of this.setState( )

A common situation that may occur is when you want to add or remove items from an array which is stored in the state object.

In vanilla JavaScript, we can directly use array methods like push, pop, splice etc to change an array. However, we just have to be careful that we don't do that directly with state. Remember we should always be using the setState( ) method.

{CODENATION}

```
class App extends React.Component {
  state = {
    numbers: [1, 2, 3, 4, 5]
  };

  addHandler = () => {
    let storedNums = [...this.state.numbers];
    storedNums.push(storedNums[storedNums.length - 1] + 1);
    this.setState({ numbers: storedNums });
  };

  render() {
   return (
      <div>
        {this.state.numbers.map((number, index) => {
          return <p key={index}>{number}</p>;
        })}
        <button onClick={this.addHandler}>add a number</button>
      </div>
    );
  }
}
```

In this example, I have a list of numbers I render to the screen. Each time I click the button I want to add a new number to the list which is the last number + 1, and then render it to the screen.

```
class App extends React.Component {
  state = {
    numbers: [1, 2, 3, 4, 5]
  };

  addHandler = () => {
    let storedNums = [...this.state.numbers];
    storedNums.push(storedNums[storedNums.length - 1] + 1);
    this.setState({ numbers: storedNums });
  };

  render() {
   return (
      <div>
        {this.state.numbers.map((number, index) => {
          return <p key={index}>{number}</p>;
        })}
        <button onClick={this.addHandler}>add a number</button>
      </div>
    );
  }
}
```

A common patter here is first you make a copy of your current state and store it in a separate variable.

Quick Note on spread operator

```
class App extends React.Component {
  state = {
    numbers: [1, 2, 3, 4, 5]
  };

  addHandler = () => {
    let storedNums = [...this.state.numbers];
    storedNums.push(storedNums[storedNums.length - 1] + 1);
    this.setState({ numbers: storedNums });
  };
```

Three dots . . . is the spread operator.
We use it to spread out the values of an array into a new array.

[...this.state.numbers] would be spread into a new array.
We need to do this because arrays in JS are passed by reference - feel free to research this.

```
class App extends React.Component {
  state = {
    numbers: [1, 2, 3, 4, 5]
  };

  addHandler = () => {
    let storedNums = [...this.state.numbers];
    storedNums.push(storedNums[storedNums.length - 1] + 1);
    this.setState({ numbers: storedNums });
  };

  render() {
   return (
      <div>
        {this.state.numbers.map((number, index) => {
          return <p key={index}>{number}</p>;
        })}
        <button onClick={this.addHandler}>add a number</button>
      </div>
    );
  }
}
```

Then you do what ever you want to the variable you created. This isn't mutating the state, do it is allowed.

```
class App extends React.Component {
  state = {
    numbers: [1, 2, 3, 4, 5]
  };

  addHandler = () => {
    let storedNums = [...this.state.numbers];
    storedNums.push(storedNums[storedNums.length - 1] + 1);
    this.setState({ numbers: storedNums });
  };

  render() {
   return (
      <div>
        {this.state.numbers.map((number, index) => {
          return <p key={index}>{number}</p>;
        })}
        <button onClick={this.addHandler}>add a number</button>
      </div>
    );
  }
}
```

Finally, when you're finished doing whatever it is your function does and you want to update the state, you call the setState( ) method like usual.

Keypoint:

You must never update the state without using setState( )

{CODENATION}

```
class App extends React.Component {
  state = {
    numbers: [1, 2, 3, 4, 5]
  };

  addHandler = () => {
    let storedNums = [...this.state.numbers];
    storedNums.push(storedNums[storedNums.length - 1] + 1);
    this.setState({ numbers: storedNums });
  };

  render() {

    return (
      <div>
        {this.state.numbers.map((number, index) => {
          return <p key={index}>{number}</p>;
        })}
        <button onClick={this.addHandler}>add a number</button>
      </div>
    );
  }
}
```

Another interesting thing to point out about this example, is we are increasing the size of the array that the map function maps over. As we are changing the state each time we click the button, which in turn triggers a re-render, our new number will be rendered to the screen each time.

Remember, the map function doesn't care how many items are in the array.

Another example, this time removing a specific number when I click on it.

```
class App extends React.Component {
  state = {
    numbers: [1, 2, 3, 4, 5]
  };

  removeHandler = index => {
    let storedNums = [...this.state.numbers];
    storedNums.splice(index, 1);
    this.setState({ numbers: storedNums });
  };

  render() {
    console.log(this.state.numbers);
    return (
      <div>
        {this.state.numbers.map((number, index) => {
          return (
            <p key={index} onClick={() => this.removeHandler(index)}>
              {number}
            </p>
          );
        })}
      </div>
    );
  }
}
```

Handling inputs

```
class App extends React.Component {
  state = {
    inputText: ""
  };

  changeHandler = event => {
    this.setState({ inputText: event.target.value });
  };

  render() {
    return (
      <div>
        <input type="text" onChange={this.changeHandler}></input>
        <p>{this.state.inputText}</p>
      </div>
    );
  }
}
```

The first thing to note when dealing with inputs, is we are using the onChange event. The onChange event fires when we change the value in the input box (ie when we type)

The second thing to note, is that in the function that handles the event, needs to know about the event itself. We just need to accept the event object as a parameter (look up the event object documentation if you need to, but it is no different in react when compared to vanilla JS.

```
class App extends React.Component {
  state = {
    inputText: "",
  };

  changeHandler = event => {
    this.setState({ inputText: event.target.value });
  };

  render() {
    return (
      <div>
        <input type="text" onChange={this.changeHandler}></input>
        <p>{this.state.inputText}</p>
      </div>
    );
  }
}
```

event.target.value is just part of the event object which stores whatever has been typed.

Using Fragments

We have already seen that in any return statement, there must be only one parent element.

In all cases, we could just use a div. Although, this is sometimes unnecessary, especially if we aren't styling the div. It adds a DOM node to the DOM tree, which uses memory. Let's have a look at an example.

```
class App extends React.Component {
  state = {
    inputText: ""
  };

  changeHandler = e => {
    this.setState({ inputText: e.target.value });
  };

  render() {
    return (
      <div>
        <input type="text" onChange={(event) => this.changeHandler(event)}></input>
        <p>{this.state.inputText}</p>
      </div>
    );
  }
}
```

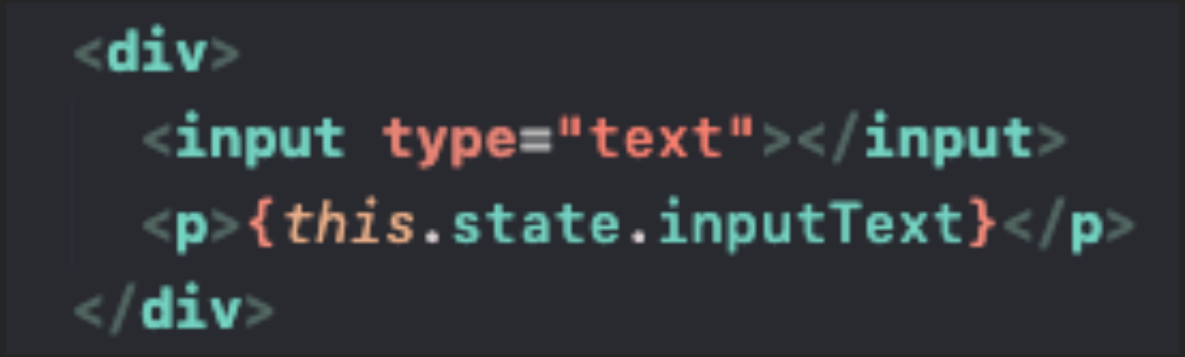
In this example, I have wrapped my input and p elements in a div, as react needs there to be a single parent element.

If we opened up our develop tools, we would see that a div has been added to the DOM. This is generally fine, however there is another way.

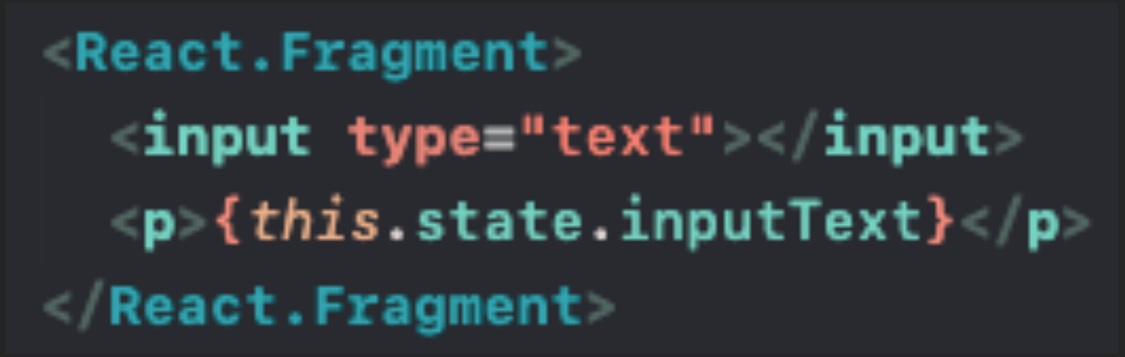
```jsx
class App extends React.Component {
  state = {
    inputText: ""
  };

  changeHandler = e => {
    this.setState({ inputText: e.target.value });
  };

  render() {
    return (
      <React.Fragment>
        <input type="text" onChange={event => this.changeHandler(event)}></input>
        <p>{this.state.inputText}</p>
      </React.Fragment>
    );
  }
}
```

A react fragment is like a ghost div. It doesn't actually get rendered to the DOM, but satisfies Reacts need to have a single parent element.

Let's have a look at the DOM trees side by side...

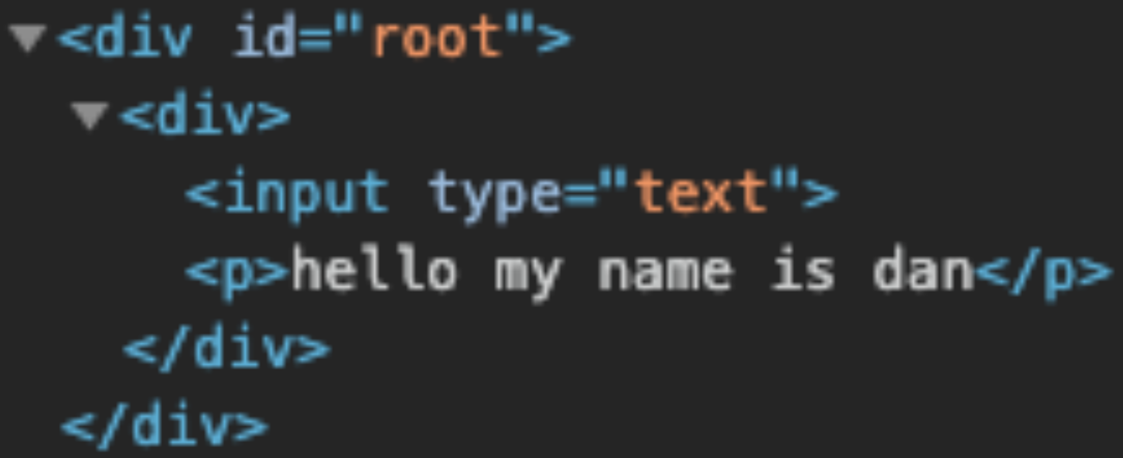


```
<div>
  <input type="text"></input>
  <p>{this.state.inputText}</p>
</div>
```



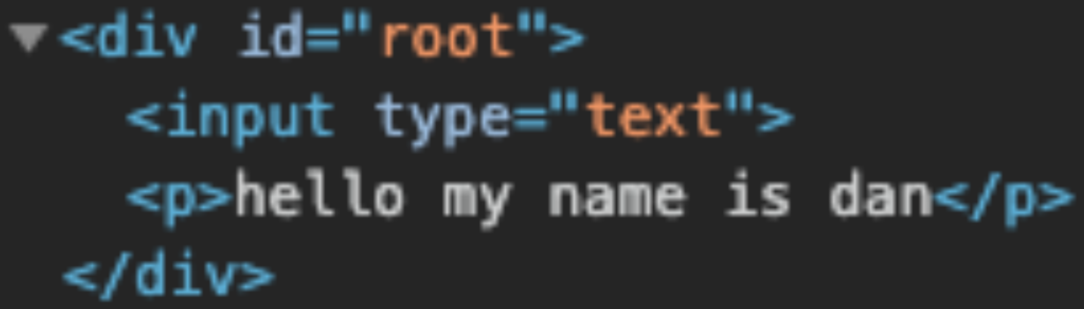
```
<React.Fragment>
  <input type="text"></input>
  <p>{this.state.inputText}</p>
</React.Fragment>
```

The DOM using a div



```
▼<div id="root">
  ▼<div>
     <input type="text">
     <p>hello my name is dan</p>
   </div>
 </div>
```

The DOM using a fragment



```
▼<div id="root">
    <input type="text">
    <p>hello my name is dan</p>
  </div>
```

For smaller applications, it won't actually save you enough memory to make a huge different. But you can still use it for a cleaner DOM tree.

A really cool thing with fragments, is the shorthand way to code them just by using an empty tag.

```
return (
      <>
        <input type="text"></input>
        <p>{this.state.inputText}</p>
      </>
    );
```

Remember though, **you can't add attributes to fragments.**

We have introduced you to the concepts which you may need, but it is important to understand that each application is different, so you will need to use these concepts to form an overall solution to the problem you are trying to solve.

We have given you the puzzle pieces you need to be able to solve the todo list challenges. However it is your job to put the puzzle pieces together.

Revisiting Learning Objectives

To understand that the state object cannot be mutated directly.

To be familiar with working with inputs in React.

{CODENATION}