# Managing State

## tl;dr

- If a component needs to keep track of changing data, this can't be props, it is "state"
- You shouldn't change state directly. Call setState() because it always triggers a re-render
- Controlled components have input fields bound to state. When that occurs we must create event handlers to copy the values to state and then re-render
- Try to avoid having state whenever possible. When you must, use a state container and keep state as simple as possible

State is a POJSO* that represents the mutable data of your React component

*Plain old JavaScript object

# State is not props!

| state | props |
| --- | --- |
| • Data known by a component<br>• Can be changed in the component<br>• Lives 100% within this component | • Data known by a component<br>• Once set, they don't change<br>• Passed in to this component from its parent |

# Stateful components are classes

# You must use a more complex form of the component

Functional component

```
function Foo(props) {
 return <div>
  {props.foo}
 </div>
}
```

Class-based component

```
import { Component } from 'react';
class Foo extends Component {
 render() {
  return <div>
   {this.props.foo}
  </div>
 }
}
```

# This form does open some cool capabilities, though

- get, set
- methods
- properties
- this.*
- Makes Java devs happier!

| Functional components | Stateful components |
| --- | --- |
| • Simpler! Thus easier to ...<br>  ○ Write<br>  ○ Test<br>  ○ Understand<br>  ○ Maintain<br>  ○ Extend | • Class-based<br>• Can use refs<br>• Can tap into lifecycle events<br>• Can have state<br>• Can use forceUpdate() |

# Initializing a component's state

## Initialize state in the constructor

**Person.js**

```
export class Person extends Component {
  constructor() {
    super()
    this.state = {first: "", last: ""};
  }
  render() { ... }
}
```

**If you have a constructor, you have to call super() as the first thing in it.**

# It is very common to pass props into a child and then set initial state from those props.

**Person.js**

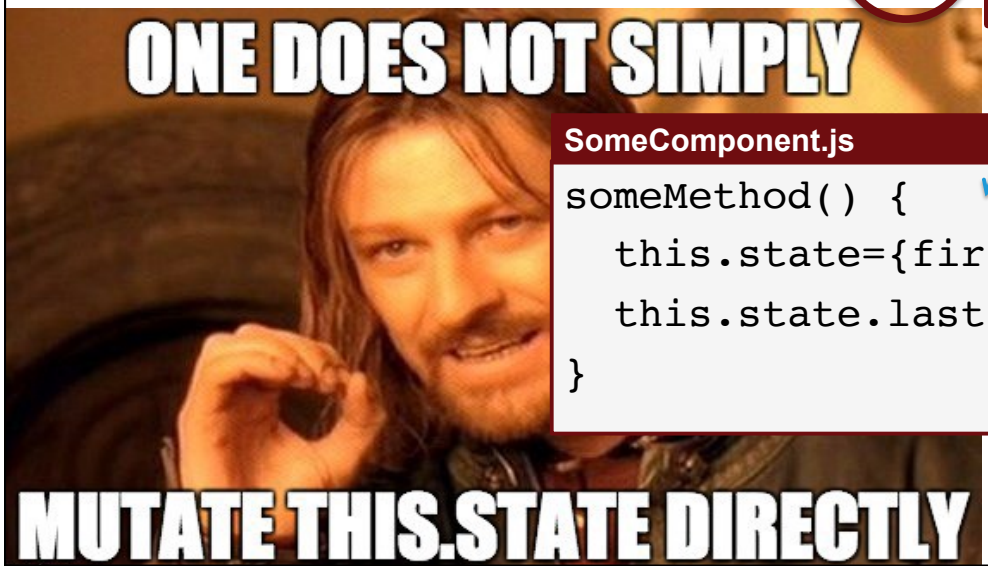
```
export class Person extends Component {
  constructor(props) {
    super(props)
    this.state = {first: props.first, last: props.last};
    // Or this.state = { ...props };
  }
  render() { ... }
}
```

**if your constructor receives props, you have to pass those props to super.**

# Be careful, though!

- React tries to be super-efficient. When a component is re-rendered React uses that object rather than disposing and re-instantiating it.
- The constructor is only called once on instantiation.
- So if the parent's props change, it will NOT re-set state in the child.

# Changing state with setState()




**Do not set state by brute force!**

**SomeComponent.js**

```
someMethod() {
  this.state={first:"Jo"};
  this.state.last = "Kim";
}
```

# setState() rerenders!


SomeComponent.js

```
someMethod() {
 this.setState({
  first:"Jo", last:"Kim"
 });
});
```

# setState() is asynchronous

SomeComponent.js

```
someMethod() {
  console.log(this.state);
  this.setState({foo:"bar"});
  console.log(this.state);
}
```

**These will be exactly the same!**

Why? For performance; multiple setState() calls might not trigger multiple refreshes.

# But you can make it synchronous

**SomeComponent.js**

```
someMethod() {
  console.log(this.state);
  this.setState((ps,p) => ({foo:"bar"}));
  console.log(this.state);
}
```

**The function can receive the previous state and props.**

Pass in a function that returns the upserted state object.

# setState upserts to the state

- If you have a huge state object with 10 keys but you only pass one key into setState, it adds that value (or updates it if it already exists).

```
{
 first:"Jo",
 last:"Day",
 email:"j@a.com",
 cell:"5551212",
 city:"Van"
}
```

+

```
{
 last:"Nguyen",
 email:"j@n.com",
 country:"US",
}
```

=

```
{
 first:"Jo",
 last:"Nguyen",
 email:"j@n.com",
 cell:"5551212",
 city:"Van",
 country:"US"
}
```

It upserts? So how do you delete from state?

- For object properties:

**SomeComponent.js**

```
someMethod() {
 this.setState({first:undefined});
}
```

- For array elements

**SomeComponent.js**

```
removeFriend(friendToRemove) {
 this.setState({friends:this.state.filter(
  f => f !== friendToRemove)});
}
```

# Forms in React

Say you have a React component that displays a person

**ShowPeople.js**

```
export class ShowPeople extends Component
 render() {
  {people.map(p => <ShowPerson
   first={p.first} last={p.last}
    eyes={p.eyes} email={p.email} />)}
 }
}
```

# Is this props or state?

You also have an AddPerson component

**AddPerson.js**

```
export class AddPerson extends Component {
  render() {
    <form onSubmit={createPerson}>
      <input name="first" />
      <input name="last" />
    </form>
  }
  createPerson() { /* Create the person here */}
}
```

**This is an 'uncontrolled component'**

# Is this props or state?

... and there's a ModifyPerson component that you pass data to.

**ShowPeople.js**

```
export class ShowPeople extends Component {
 render() {
  // Must pass values so they can be modified
  <ModifyPerson first={p.first}
    last={p.last} ... />
 }
 otherMethod() { /* Do stuff here */}
}
```

Is this props or state?

And inside of ModifyPerson, we use those values

**ModifyPerson.js**

```
export class ModifyPerson extends Component
 render() {
  <input value={this.props.first} />
  <input value={this.props.last} />
  <input value={this.props.email} />
 }
}
```

**This cannot work b/c props can't change**

Is this props or state?

# So clearly we need both for a complete app

Props when being passed in

Then state when it is being changed.

**ModifyPerson.js**

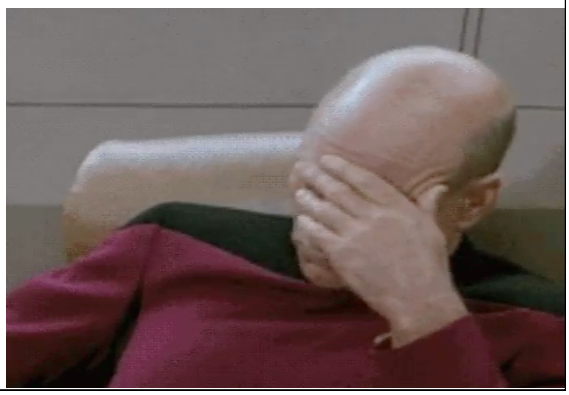
```
export class ModifyPerson extends Component {
 constructor(props) {
  super(props);
  this.state = props;
 }
 render() {
  <input value={this.state.first} />
  <input value={this.state.last} />
  <input value={this.state.email} />
 }
}
```

**When form values are bound to state, it is a 'controlled component'**

# But we now have a new problem!

```
<input value={this.state.fname} />
```

- input.value is bound to state.
- When the user changes value by typing, the form immediately re-binds the value to what is in state
- End result: <u>The user types and nothing appears to happen!</u>



# Here's a solution ... change state!

1. Add a change event handler to the input
2. In the handler, call setState() with the new value
3. Since setState() calls render, the component is re-drawn
4. The new value is now bound to the input
5. The user sees their changes!

**ModifyPerson.js**

```
export class ModifyPerson extends Component {
 ...
 render() {
  <input value={this.state.first}
    onChange={this.handleChange} />
 }
 handleChange(e) {
  this.setState(
   {first: e.target.value});
 }
}
```

# Remember that events in React are "synthetic"

- They do not behave like native DOM events. They're better!
- The event object is normalized for all browsers
- The synthetic onChange event fires on every native keyup
- React allows the value property to be used with <select>s and <textarea>s
- For example ...

**ChooseSchool.js**

```
export class ChooseSchool extends Component {
 ...
 render() {
  <select value={this.state.college} onChange={this.ch}>
   <option value="10">Faber</option>
   <option value="27">South Harmon</option>
   <option value="51">Greendale</option>
   <option value="34">Wassamotta</option>
   <option value="86">UC Sunnydale</option>
  </select>
  <textarea value={this.state.description}
   onChange={this.ch}></textarea>
 }
}
```

# So use checked and value

**checked**

```
<input type='radio' />
<input type='checkbox'/>
```

**value**

```
<input type='text' />
<input type='number' />
<input type='email' />
<input type='date' />
<input type='tel' />
```

- etc. etc.
- Literally everything else including ...

```
<select>
<textarea>
```

# 5 tips for handling state

## 1. If you can avoid state at all, do!

- Any state means that there are side-effects.
- The component is no longer a pure function.
- Understanding is harder
- Testing is harder
- Modification is harder
- Extension is harder

# 2. Use a state manager

- State is the most complex thing in React.
- A state manager like Redux and MobX can greatly simplify your React components

# 3. Keep state as small as possible

- The smaller, the simpler
- Simpler is more abstract.

# 4. If you don't use something in the render method, it doesn't belong in state

- It can be just a class-scope variable.
- In other words, use this.whatever instead of this.state.whatever

# 5. Combine form inputHandlers

- There seems to be a pattern of very smart devs having a single do-it-all handler for forms.
- In this pattern there is one "handleChange" method and every onChange event uses it.

# tl;dr

- If a component needs to keep track of changing data, this can't be props, it is "state"
- You shouldn't change state directly. Call setState() because it always triggers a re-render
- Controlled components have input fields bound to state. When that occurs we must create event handlers to copy the values to state and then re-render
- Try to avoid having state whenever possible. When you must, use a state container and keep state as simple as possible