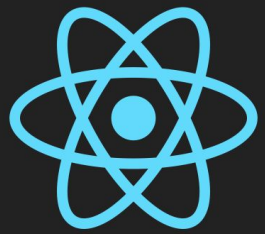
Lecture 2.2

Component State & Life Cycle



Topics

- Components
  - Recap
  - State
  - Life Cycle

Components Recap

Recap

Components

- Like functions
- Input: props, state | Output: UI
- Reusable and composable
- <Component />
- Can manage a private state

Reactive updates

- React will react (to updates)
- Take updates to the browser

Virtual views in memory

- Generate HTML using JavaScript
- No HTML template language
- Tree reconciliation

Functional vs Class Components

- There are two types of components in React, Functional and Class components.
- Prefer to use the Function Components over Class, because they are much simpler. However, Class are much more powerful.
- Both return JSX. (HTML mixed with JavaScript)

```
const Intro = () => {
    return <p>Hi there...</p>
}
```

```
class App extends React.Component {
    render() {
        return <p>Hi there...</p>
    }
}
```

Props vs State

- Both Functional and Class components can use props as input, but props are immutable and cannot be changed. It is read-only.
- State is internal to Class components and is mutable. It can change.
- State is interesting, because of how React uses it to auto-reflect changes in DOM

Component State

What is State?

- State is a JavaScript object that stores a component's dynamic data and determines the component's behavior
- State is dynamic, it enables a component to be dynamic by tracking information between renders
- State can only be used within a class component. It is private to a component.
- State is similar to props, but unlike props, State can change.

State is immutable

- State in React is immutable. State should never be altered/changed directly.
- Changes should be made to a copy of the current version of state.
- This has benefits such as providing the ability to review state at different points in time for apps to hot reload.
- *(hot reload: automatic reloading of the page in browser when you make code changes)*

Adding State

```
function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

- The Clock component is reusable, however it sets up a timer and updates the UI every second.
- Ideally, the Clock component should update itself. We need to add "state" to do this.

Converting Function to a Class

1. Create an ES6 class with same name, that extends React.Component
2. Add a render() and move the JSX into the method
3. Replace props with this.props in the render() body

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Adding Local State to a Class

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

- Replace this.props.date with this.state.date in render() method
- Add a class constructor that assigns the initial this.state.
- We pass props to base constructor
- Remove date prop from <Clock /> element

setState method

One of the rules of React component class is they must extend from React.Component. As a result you will inherit the setState method.

To use state correctly, means we do not modify state directly. We use setState() to modify the state.

When you call this.setState(state) from within your class the following will happen:

- The state object will be copied onto this.state (but won't change immediately)
- React will then re-render your component and it's nested components

We do not modify state directly

```
// Wrong
this.state.comment = 'Hello';
```

We use setState to modify state

```
// Correct
this.setState({comment: 'Hello'});
```

setState is asynchronous

- setState causes reconciliation *(the process of re-rendering the component tree)*
- setState is asynchronous and this allows multiple calls to setState in a single scope and will not trigger re-rendering of the whole tree
- This is why you don't see the new values in state right away after you update it.
- React will try to group or batch setState calls into a single call

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

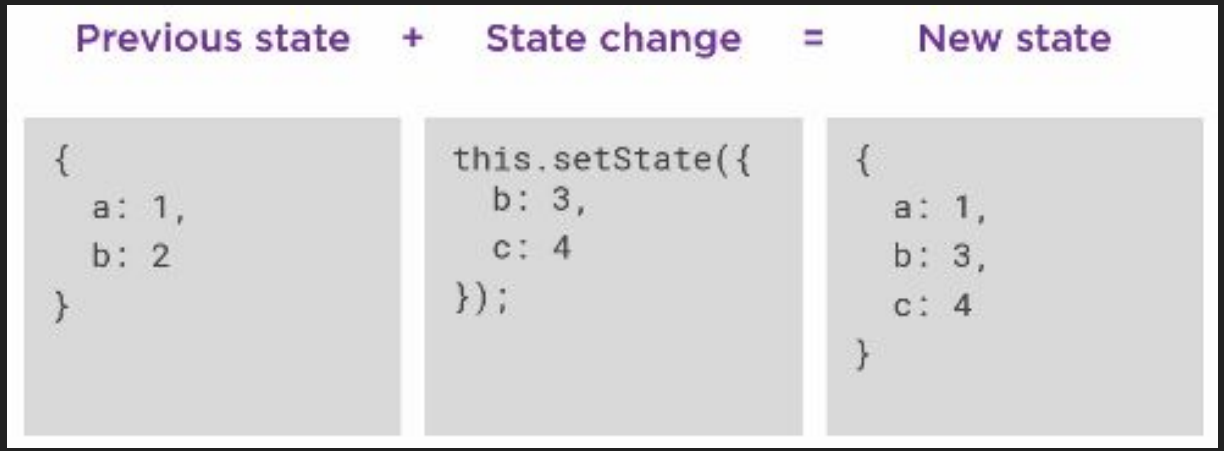
Don’t rely on this.state and this.prop values, they may be updated asynchronously

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

setState() will receive previous state and props at the time of the update

State Updates are merged

- The setState method merges the new state with the old state. All of the previous state remains unless it is overwritten.



React Hooks

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

- *Hooks* are a new addition in React 16.8. They let you use state and other React features without writing a class.
- Hooks let us organize the logic inside a component into reusable isolated units

Video: State of Components

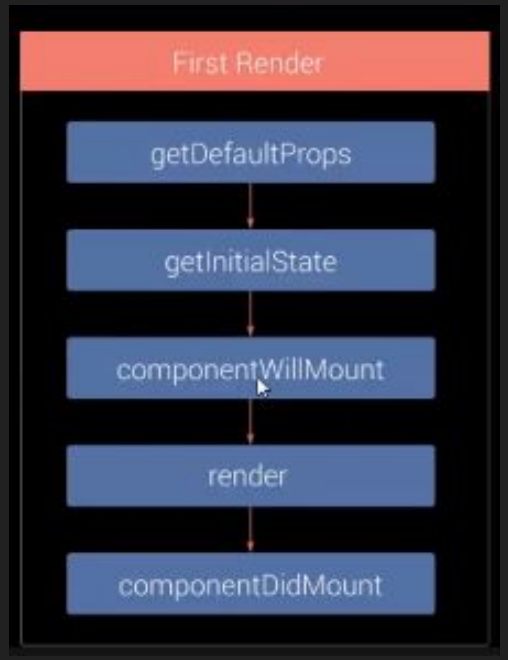
https://youtu.be/e5n9j9n83OM

Component Lifecycle

Video: Component Lifecycle

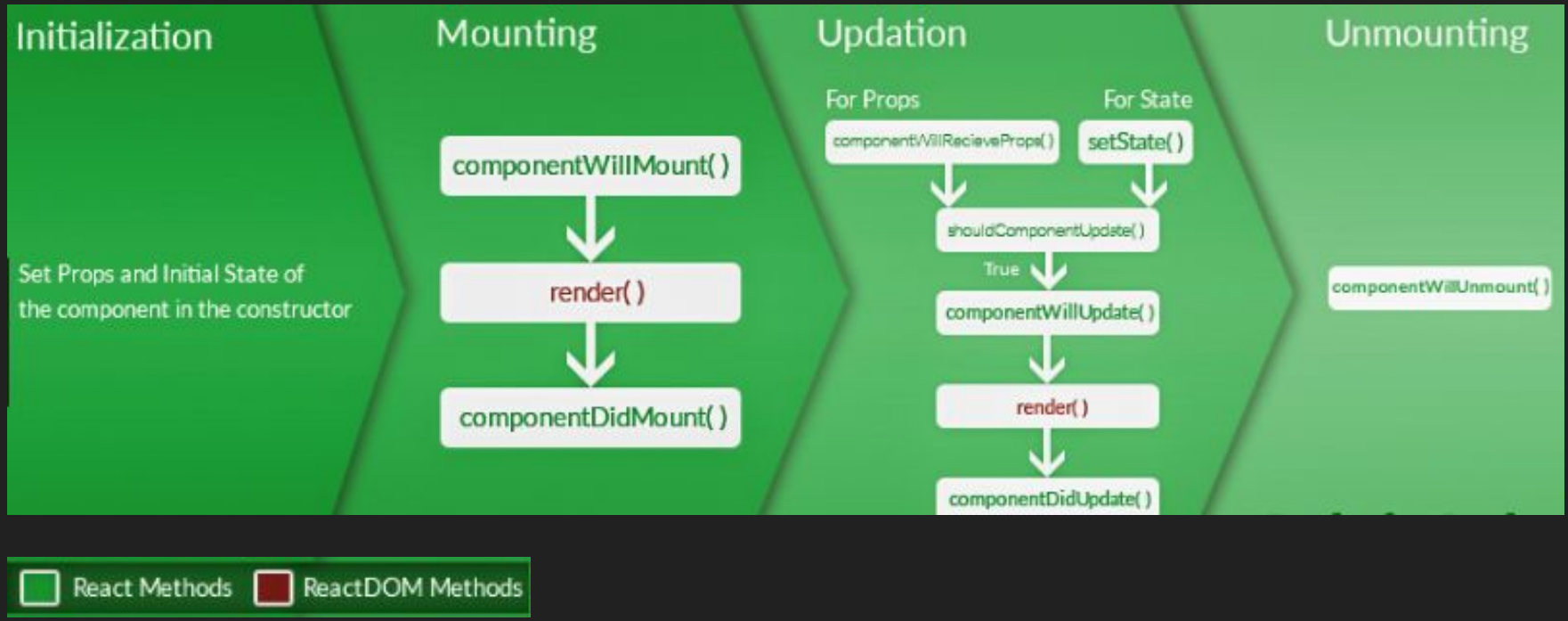
https://youtu.be/iYz2OKWO09U

Component Lifecycle



- React class components allow you to override lifecycle methods if you need to perform operations at particular times in a components lifecycle.
- When it has just been created or immediately after it has been inserted into the DOM.
- Lifecycle methods are useful when you are implementing a React component to wrap an API. *ie. create a React component for a JQuery plugin, you would use the lifecycle methods to initialize the jQuery plugin*

Component Lifecycle cont.



```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
```

- After Clock component has been inserted into the DOM, React will call componentDidMount() lifecycle method. It will setup a timer to call the component's tick() method
- If the Clock component is removed from the DOM, React calls the componentWillUnmount() lifecycle method so the timer is stopped.

```
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```