Web Application Development:

The Component Lifecycle

Introduction

- Keeping track of everything components do sometimes can be tough.
- To help with this, React provides us with lifecycle methods.
- Unsurprisingly, lifecycle methods are special methods that automatically get called as our component goes about its business.
- They notify us of important milestones in a component's life, and we can use these notifications to simply pay attention or change what our component is about to do.

Lifecycle Methods

- componentWillMount
- componentDidMount
- componentWillUnmount
- componentWillUpdate
- componentDidUpdate
- shouldComponentUpdate
- componentWillReceiveProps
- componentDidCatch

Coding example

```
class CounterParent extends React.Component {
constructor(props) {
super(props);

console.log("constructor: Default state time!");

this.state = {
count: 0
};

this.increase = this.increase.bind(this);
}

increase() {
this.setState({
count: this.state.count + 1
});
}

componentWillUpdate(newProps, newState) {
console.log("componentWillUpdate: Component is about to update!");
}

componentDidUpdate(currentProps, currentState) {
console.log("componentDidUpdate: Component just updated!");
}
```

```
componentWillMount() {
console.log("componentWillMount: Component is about to mount!");
}

componentDidMount() {
console.log("componentDidMount: Component just mounted!");
}

componentWillUnmount() {
console.log("componentWillUnmount: Component is about to be removed
from the DOM!");
}

shouldComponentUpdate(newProps, newState) {
console.log("shouldComponentUpdate: Should component update?");

if (newState.count < 5) {
console.log("shouldComponentUpdate: Component should update!");
return true;
} else {
ReactDOM.unmountComponentAtNode(destination);
console.log("shouldComponentUpdate: Component should not update!");
return false;
}
}

componentWillReceiveProps(newProps) {
console.log("componentWillReceiveProps: Component will get new
props!");
}

render() {
var backgroundStyle = {
padding: 50,
border: "#333 2px dotted",
width: 250,
height: 100,
borderRadius: 10,
textAlign: "center"
};

return (
<div style={backgroundStyle}>
<Counter display={this.state.count} />
<button onClick={this.increase}>
+
</button>
```

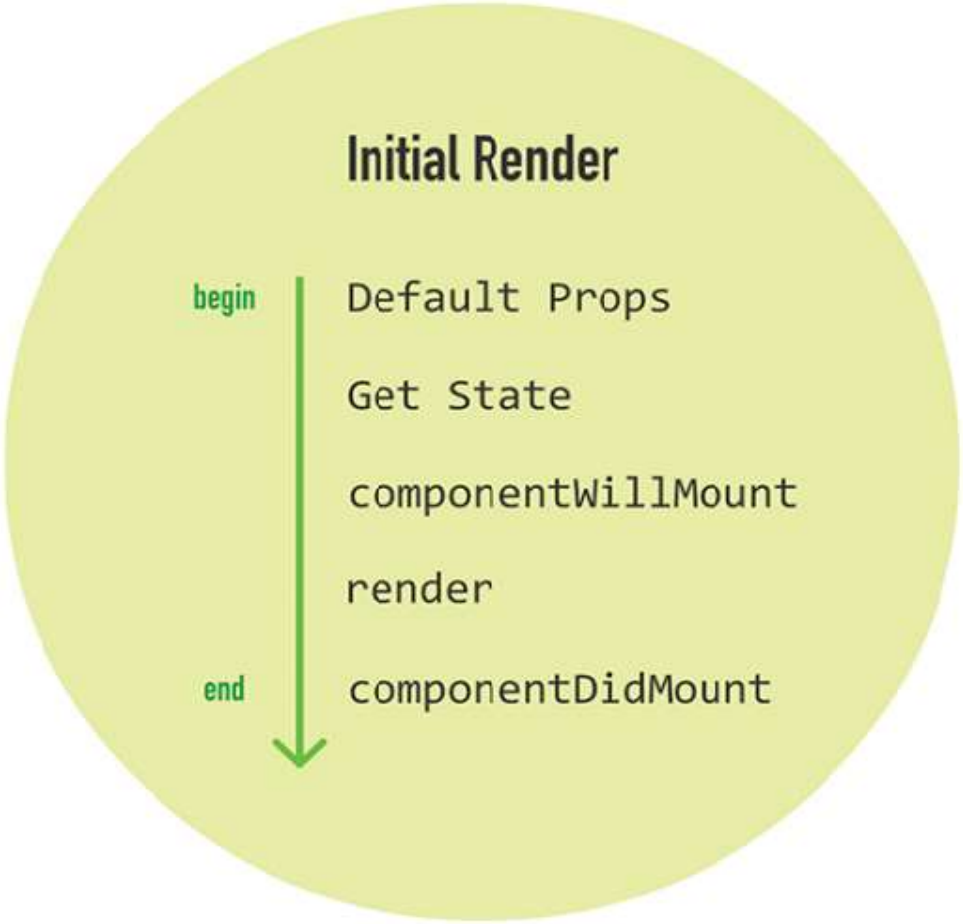
```
</div>
);
}
}

console.log("defaultProps: Default prop time!");
CounterParent.defaultProps = {

};
```

The Initial Rendering Phase

- When your component is about to start its life and make its way to the DOM, the following lifecycle methods get called



Getting the Default Props

- This property on the component allows you to specify the default value of this.props.
- If we wanted to set a name property on our CounterParent component, it could look as follows:

```
CounterParent.defaultProps = { name:    "Iron Man" };
```

Getting the Default State

- This step happens inside your component's constructor.
- You get the chance to specify the default value of this.state as part of your component's creation:

```
constructor(props) {
super(props);
console.log("constructor: Default state time!");
this.state = {
count: 0
};
this.increase = this.increase.bind(this);
}
```

- Notice that we're defining our state object and initializing it with a count property whose value is 0.

Getting the Default State

- **componentWillMount**
  - This is the last method that gets called before your component gets rendered to the DOM.
  - There is an important point to note here: If you call setState inside this method, your component will not re-render.
- **Render**
  - This one should be very familiar to you by now.
  - Every component must have this method defined, and it is responsible for returning some JSX.
  - If you do not want to render anything, simply return null or false.

Getting the Default State

- **componentDidMount**
  - This method gets called immediately after your component renders and gets placed on the DOM.
  - At this point, you can safely perform any DOM querying operations without worrying about whether your component has made it.
  - If you have any code that depends on your component being ready, you can specify all of that code here as well.
  - With the exception of the render method, all of these lifecycle methods can fire only once.
  - That's quite different from the methods you see next.

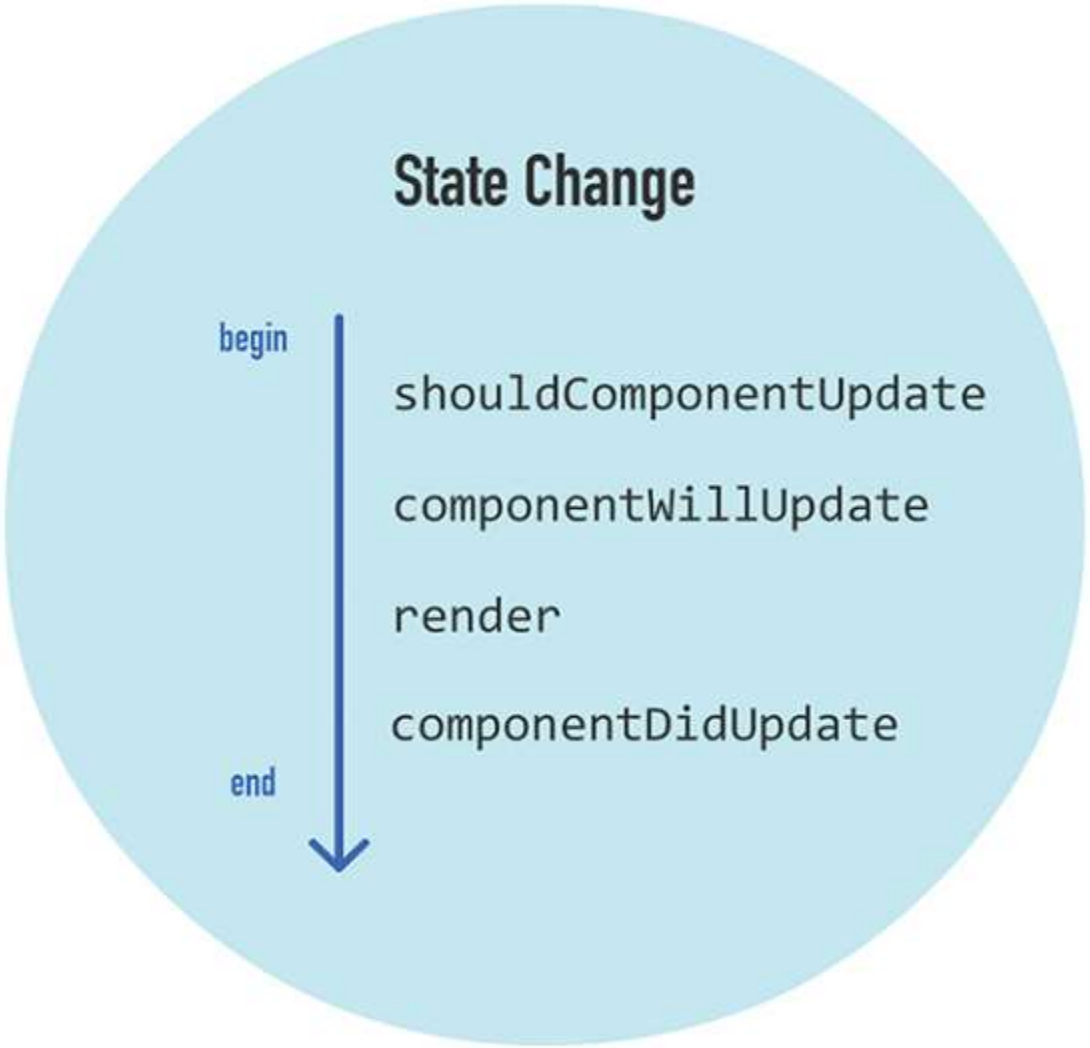
Getting the Default State

- **The Updating Phase**
  - After your components get added to the DOM, they can potentially update and re-render when a prop or state change occurs.
  - During this time, a different collection of lifecycle methods gets called

Dealing with State Changes

- First, let's look at a state change.
- As we mentioned earlier, when a state change occurs, your component calls its render method again.
- Any components that rely on the output of this component also get their render methods called.
- This is done to ensure that the component is always displaying the latest version of itself.
- All of that is true, but it's only a partial representation of what happens.

Dealing with State Changes



Dealing with State Changes

- **shouldComponentUpdate**
  - Sometimes you don’t want your component to update when a state change occurs.
  - This method allows you to control this updating behavior. If you use this method and return a true value, the component will update.
  - If this method returns a false value, this component will skip updating.
  - That probably sounds a bit confusing, so take a look at a simple snippet

Dealing with State Changes

```
shouldComponentUpdate(newProps, newState)
{
        console.log("shouldComponentUpdate: Should component update?");
        if (newState.count < 5)
        {
                console.log("shouldComponentUpdate: Component should
                update!");
                return true;
        }
        else
        {
                ReactDOM.unmountComponentAtNode(destination);
                console.log("shouldComponentUpdate: Component should
                not update!");
                return false;
        }
}
```

Dealing with State Changes

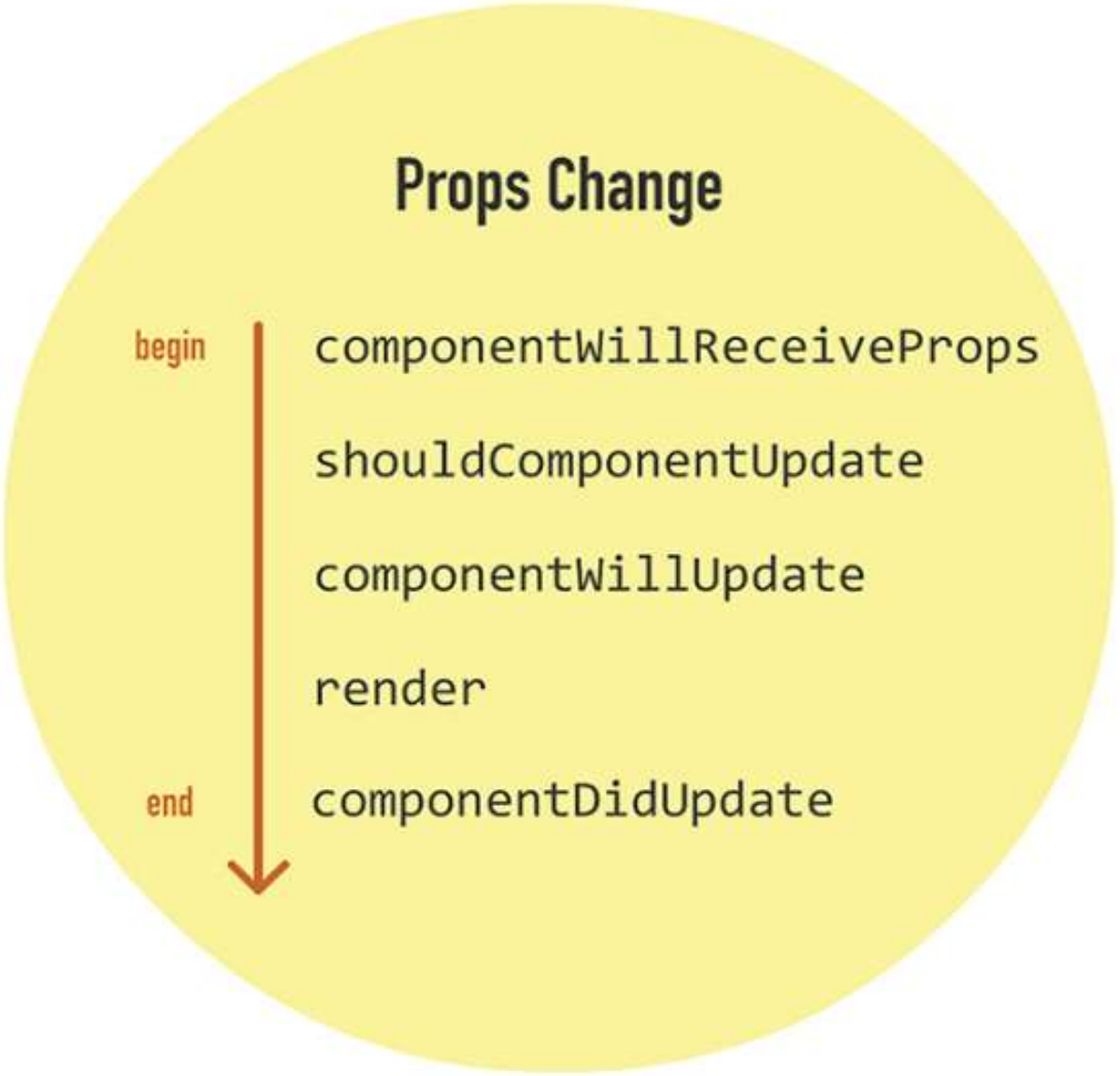
- This method gets called with two arguments, which we named newProps and newState.
- In this snippet of code, we check whether the new value of our id state property is less than or equal to 2.
- If the value is less than or equal to 2, we return true to indicate that this component should update.
- If the value is not less than or equal to 2, we return false to indicate that this component should not update.

Dealing with State Changes

- **componentWillUpdate**
  - This method gets called just before your component is about to update.
  - Nothing too exciting happens here.
  - One point to note is that you cannot change your state by calling this.setState from this method.
- **render**
  - If you didn't override the update via shouldComponentUpdate, the code inside render gets called again to ensure that your component displays itself properly.
- **componentDidUpdate**
  - This method gets called after your component updates and the render method has been called. If you need to execute any code after the update takes place, this is the place to stash it.

Dealing with Prop Changes

- The other time your component updates is when its prop value changes after it has been rendered into the DOM.



Dealing with Prop Changes

- The only new method here is componentWillReceiveProps.
- This method receives one argument, and this argument contains the new prop value that is about to be assigned to it.
- You saw the rest of the lifecycle methods when looking at state changes, so let’s not revisit them.
- Their behavior is identical when dealing with a prop change.

The Unmounting Phase

- The last phase to look at is when your component is about to be destroyed and removed from the DOM.
- Only one lifecycle method is active here, and that is componentWillUnmount.
- You perform cleanup-related tasks here, such as removing event listeners and stopping timers.
- After this method gets called, your component is removed from the DOM and you can say goodbye to it.

The Unmounting Phase