React Lifecycle & State

CS568 – Web Application Development I

Computer Science Department

Maharishi International University

Maharishi International University - Fairfield, Iowa

Content

- Class-based component
- State
- Event handler
- Passing method references between components
- Lifting state up
- Updater function

Class-Based Component Example

```
class DisplayMsg extends React.Component {
   render() {
      const { text, type } = this.props; // JS, where 'props' instance variable come from?
         return ( // JSX
           <div style={{ color: type === "success" ? 'green' : 'red' }}> {text} </div>
         );
    }
}

<DisplayMsg text="Okay" type="success" />
```

constructor and super()

```
class Person {
    constructor(n) {
        this.name = n;
    }
}

class NicePerson extends Person {
    constructor(n) {
        super(n);
        console.log(this); // { name: 'Asaad' }
    }
}

const asaad = new NicePerson(`Asaad`);
```

In JavaScript, **super** refers to the parent class constructor. Therefore, it's essential to understand that you can only use this in a constructor AFTER you've called the parent constructor.

React Constructor

```
class Button extends React.Component {
    constructor(props) {
        ~~super();~~ super(props);
        console.log(props); // okay
        console.log(this.props); // undefined
    }
    ...
}
```

Even if you forget to pass **props** to **super()**, React would still set them right afterwards (outside the constructor).
But **this.props** would still be **undefined** between the **super** call and the end of your **constructor**.
It's recommended to always pass down **super(props)**, even though it isn't strictly necessary. This ensures **this.props** is set even before the **constructor** exits.

Events

We can add an event handler to any component with an "**onEvent**" property.

```
const Button = () => {
  return (
    <button onClick={() => console.log('Button clicked')}>click me</button>
  );
};
```

Why is this considered a bad code?

All DOM-related attributes (which are handled by React) need to be camel-case

Binding Event Handler Methods

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: 'React'}; // similar to defining state outside constructor
  }
  whoIsThis() {
    console.log(this.state.name);
  }
  render() {
    return (
      <button onClick={this.whoIsThis}>Hello {this.state.name}</button>
    )
  }
}
```

How to avoid this error?

Binding Event Handler Methods

- Using bind method

this.whoIsThis.bind(this)

- Using arrow function

whoIsThis = () => {….}

State of class-based component

- The **`state`** instance property is a special one because React will manage it. **`state`** is a plain JavaScript object.
- Only use the function setState to change state. This function call will trigger the re-rendering
- Do not try to mutate the state without using setState
- **`setState:`**
    - **Asynchronous update**
    - **batched**
    - Merged

Asynchronous update

```
class App extends React.Component {
  state = { value: 0 };

  action = ()=>{
    this.setState({ value: this.state.value + 1});
    console.log(this.state.value);
  }
  render() {
    return (
      <button onClick={this.action}>Click</button>
    );
  }

}
```

Batched

```
class App extends React.Component {
  state = { value: 0 };

  action = ()=>{
    this.setState({ value: this.state.value + 1});
    this.setState({ value: this.state.value + 1});
    this.setState({ value: this.state.value + 1});
    console.log(this.state.value);
  }
  render() {
    return (
      <button onClick={this.action}>Click</button>
    );
  }

}
```

Update with the latest value

```
class App extends React.Component {
  state = { value: 0 };

  action = ()=>{
    this.setState((prevState) => ({ value: prevState.value + 1}));
    this.setState((prevState) => ({ value: prevState.value + 1}));
    this.setState((prevState) => ({ value: prevState.value + 1}));
    console.log(this.state.value);
  }
  render() {
    return (
      <button onClick={this.action}>Click</button>
    );
  }

}
```

Merged

```
class App extends React.Component {
  state = { value: 0 };

  action = ()=>{
    this.setState({ value: this.state.value + 1});
    this.setState({ id: 1}, () => console.log(this.state));
  }
  render() {
    return (
      <button onClick={this.action}>Click</button>
    );
  }

}
```

Actions after updated

```
class App extends React.Component {
  state = { value: 0 };

  action = ()=>{
    this.setState({ value: this.state.value + 1},
         ()=>console.log(this.state.value));
    console.log(this.state.value);
  }
  render() {
    return (
      <button onClick={this.action}>Click</button>
      <p>{this.state.value}</p>
    );
  }
}
```

Pure Component

If a React component does not modify anything outside of its definition, we can label that component pure as well. Pure components have a better chance at being reused without any problems.

PureComponent changes the life-cycle method **`shouldComponentUpdate`** and adds some logic to automatically check whether a re-render is required for the component. This allows the Pure Component to call method render only if it detects changes in state or props. **Only a shallow check of props and state will be made. Take advantage of Immutable attributes.**

Class-base pure component example

```
class Message extends React.PureComponent{
 render(){
   console.log('rendered message')
   return <p>{this.props.msg}</p>
 }
}
```

Rendering Sibling Components

Adjacent elements can’t be rendered in React because each of them gets translated into a function call when JSX gets converted.

```
root.render(
 [
    <Comp1 />,
    <Comp2 />
 ]
);
```

```
root.render(
  <div>
    <Comp1 />
    <Comp2 />
  </div>
);
```

```
root.render(
  <React.Fragment>
    <Comp1 />
    <Comp2 />
  </React.Fragment>
);
```

Without introducing a new DOM parent node.

```
root.render(
  <>
    <Comp1 />
    <Comp2 />
  </>
);
```

The empty tag will get transpiled into the React.Fragment.

Mounting vs Unmounting

Rendering a React component in the browser for the first time is referred to as "**mounting**" and removing it from the browser is referred to as "**unmounting**".

Side Effects

- In computer science, a function or expression is said to have a side effect if, in addition to producing a value, it also modifies some state or interacts with calling functions or the outside world. For example, a function might modify a global variable, write data to a file, read data, call other side-effecting functions.
- Because understanding an effectful program requires thinking about all possible histories, side effects often make a program harder to understand.
- Side effects are essential to enable a program to interact with the outside world. However, we need to manage side effects to avoid unexpected behaviors

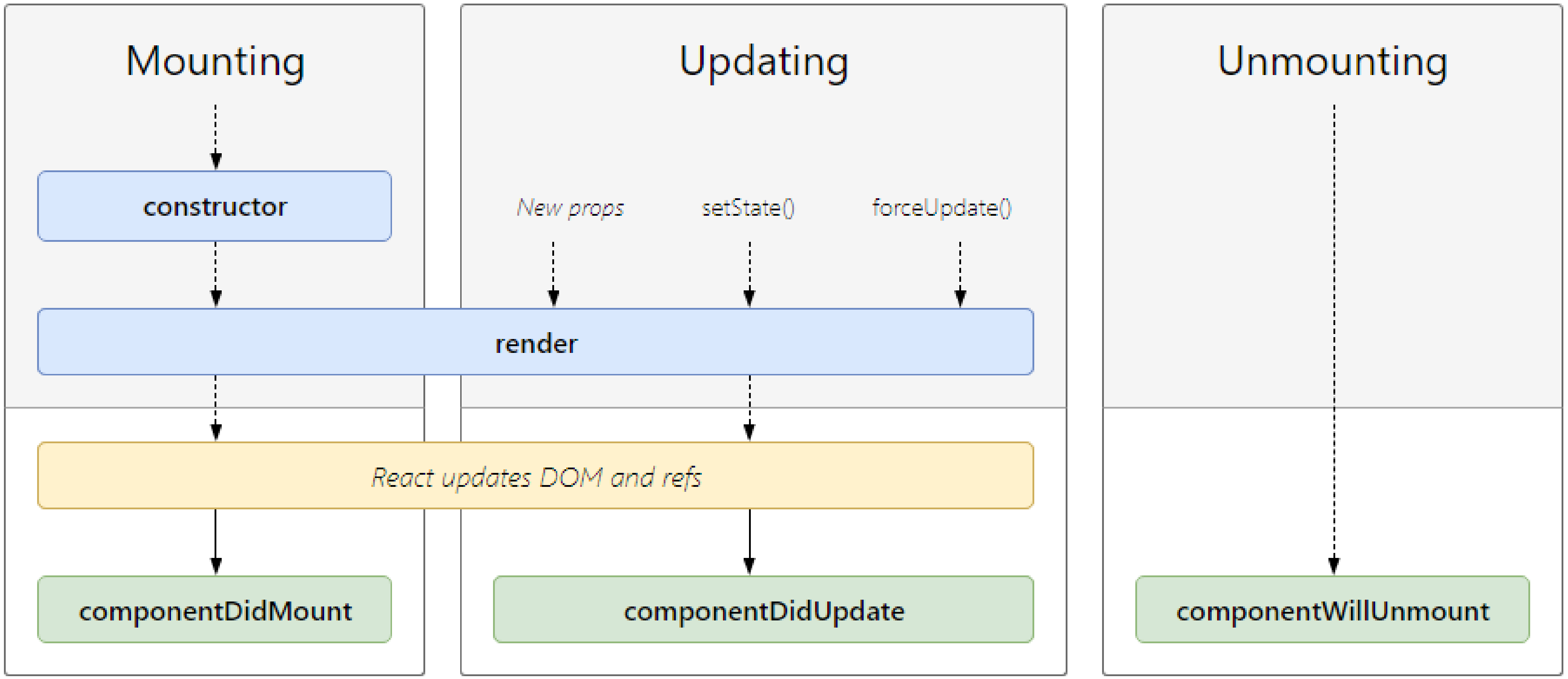
Side Effect Example

- In React, a side effect refers to any additional operation or effect that occurs in a component, beyond rendering the UI
- React app might need to change the page title. This is not something you can do directly with the React API. You need to use the DOM API for it: *document.title="new title"*
- When rendering an input form-element you might want to set focus a text box. That too has to be done with the DOM API: *element.focus()*

Component Lifecycle Hooks

- Side effects usually need to happen either before or after React render task.
- React provides **lifecycle methods** in class components to let you perform custom operations before or after the render method.
- You can do things after a component is first mounted inside a **`componentDidMount`** class method
- You can do things after a components gets an update inside a **`componentDidUpdate`** class method.
- You can do things right before a component is removed from the browser inside a **`componentWillUnmount`** class method.

Component Lifecycle Hooks



https://github.com/wojtekmaj/react-lifecycle-methods-diagram

Demo

- ComponentDidMount
- ComponentDidUpdate
- ComponentWillUnmount

Functional and Class Components

Functional component	Class Component
Stateless or presentational	Stateful
Receive props from arguments	This.props
Do not have this	Have this
Use hooks to add states and manage lifecycle	This.state Have functions to manage lifecycle
Return statement contains the rendering content	Override the function render()

Is it possible to change the data in the parent from the child component?

Yes, pass down the function as a prop in the parent that changes the data.

Summary

- Class-based component: constructor, super
- Event handler
- State: setState
- Lifecycle: Mount, update, Unmount