Mastering React Fundamentals

CS571 – Mobile Application Development

Maharishi University of Management

Department of Computer Science

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

Frameworks

Frameworks are usually large codebase and full of features.

When working with a framework, many smart design decisions are already made for you, which gives you a clear path to focus on writing good application-level logic.

A framework usually wants you to code everything a certain way.

React

React is a JavaScript **library**, not a framework. It is not a complete solution, and you will often need to use third-party libraries with React to create any solution.

React is a small library that focuses on just one thing and on doing that thing extremely well. **Building User Interfaces**.

Imperative vs Declarative

Imperative programming outlines a precise series of steps to get to what you want (**how**), like JS.

Declarative programming just declares **what** you want, like HTML.

Is React Imperative or Declarative?

React is Declarative

The browser APIs aren't easy to work with.

React allows us to declare/describe what we want, and the library will take care of the DOM manipulation.

Breaking a complex problem into discrete components, so we can reuse these components.

Working with the DOM?

It is advised to avoid traversing the DOM tree as much as possible. Any operation on the DOM is done in the same single thread that's responsible for everything else that's happening in the browser, including reactions to user events like typing, scrolling, resizing, etc.

Any expensive operation on the DOM means a slow and janky experience for the user. It is extremely important that your applications do the absolute minimum operations and batch them where possible.

Imperative Example

```
const SLIDE = {
  title: 'Slide Title',
  bullets: ['Item1', 'Item2', 'Item3']
}

function createSlide(slide) {
  const slideElement = document.createElement('div')

  const title = document.createElement('h1');
  title.innerHTML = slide.title;
  slideElement.appendChild(title);

  const bullets = document.createElement('ul');
  slide.bullets.forEach(bullet => {
    const bulletElement = document.createElement('li');
    bulletElement.innerHTML = bullet;
    bullets.appendChild(bulletElement);
  })
  slideElement.appendChild(bullets)

  return slideElement
}
```

Writing React

JSX is XML-like syntax extension of JavaScript, **transpiles to JavaScript**.

You can still write React without JSX using `React.createElement()`.

All Lowercase tags are treated as HTML/SVG tags.

All Uppercase tags are treated as custom components.

Components are just functions, return a node (something React can render).

Declarative Example

```
const SLIDE = {
  title: 'Slide Title',
  bullets: ['Item1', 'Item2', 'Item3']
}

function createSlide(slide) {
  return (
    <div>
      <h1>{slide.title}</h1>
      <ul>
        {slide.bullets.map(bullet => <li>{bullet}</li>)}
      </ul>
    </div>
  )
}
```

Declarative Example

```
function createSlides(slides) {
  return (
    <div>
       {slides.map(slide =>{
            <div>
              <h1>{slide.title}</h1>
              <ul>
                {slide.bullets.map(bullet => <li>{bullet}</li>)}
              </ul>
            </div>
       })}
  </div>
)}
```

Declarative Example

```
function createSlides(slides) {
  return (
    <div>
       {slides.map(slide => <Slide slide={slide} />)}
    </div>
)}


const Slide = ({slide}) => (
  <div>
    <h1>{slide.title}</h1>
    <ul>
      {slide.bullets.map(bullet => <li>{bullet}</li>)}
    </ul>
  </div>
)
```

React Component

The basic form of a React component is plain-old JavaScript function.

We call the function with some input (props) and it gives us some output (UI).

We can **reuse** a single component in multiple UIs and components can contain other components.

React Component Name

The first letter of React Component has to be a **capital letter**, this is a requirement since we will be dealing with a mix of HTML elements and React elements.

A JSX compiler (like **Babel**) will consider all names that start with a lowercase letter as names of HTML elements.

Component Patterns

- Container
- Presentational
- Higher order components (HOC’s)

props

Just like HTML elements can be assigned attributes like **`id`** or **`title`**, a React component may also receive a list of attributes when it gets rendered.

In React, the list of attributes received by a React component is known as **`props`**. A React function component receives this list as its **first argument**.

Unlike HTML attributes that receive only strings as value, **`props`** can be any JS value.

It is common to use object destructuring whenever we use **`props`** (or **`state`**).

Example

```
function Button (props) {
   console.log(props); // {label: "Save"}
   return <button type="submit">{props.label}</button>;
}

ReactDOM.render(<Button label="Save" />, mountNode);
```

Example

```
function Button ({label}) {
   return <button type="submit">{label}</button>;
}

ReactDOM.render(<Button label="Save" />, mountNode);
```

Example

```
const data = [
  { href: "http://miu.edu/mwp", src: "mwp.png" },
  { href: "http://miu.edu/mwa", src: "mwa.png" },
];
const Courses = ({ courses }) => {
  return (
    <div>
      {courses.map(course => <ClickableImage {...course} />)}
    </div>
  );
};

<Courses courses={data} />
```

What does **course** *represent?*

Spread Props

If we have an object and we want to pass all its key/value pairs as props, we can use the spread operator and React will spread the object key/value pairs as props

```
const obj = { firstname: 'Asaad', lastname: 'Saad'}

<Comp firstname={obj.firstname} lastname={obj.lastname} />

<Comp {...obj} />
```

Expressions in JSX, JS vs. JSX!

You can include a JavaScript expression using a pair of curly brackets anywhere within JSX.

Only expressions can be included inside these **{}** curly brackets. You cannot include a regular if-statement but a ternary expression is okay. Anything that returns a value is okay. You can always put any code in a function, make it return something, and call that function within the curly brackets.

JavaScript variables are also expressions, so when the component receives a list of **`props`** you can use these **`props`** inside curly brackets.

JavaScript object literals are also expressions. Sometimes we use a JavaScript object inside curly brackets, which makes it look like double curly brackets.

Expressions Example

```
const ErrorDisplay = ({ message }) => (
  <div style={{ color:'red', backgroundColor:'yellow' }}>
    {message}
  </div>
);

ReactDOM.render(
  <ErrorDisplay
    message="Something went wrong!"
  />, mountNode);
```

The **`style`** attribute above is a special one. We use an object as its value and that object defines the styles, we use camel-case property names and string values. React translates these style objects into inline CSS style attributes.

Creating Components Using class

We define a class that extends **`React.Component`**. A class-based React component has to at least define an **instance method** named **`render()`**. This method returns an element that represents the output from the component.

We may use **`this.props`** inside the rendered JSX. Every component gets a special **instance property** named **`props`** that holds all values passed to that component's element when it was instantiated.

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. It's very important to understand that you can't use **`this`** in a constructor until 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.

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.**

Reusable Pure Components

```
const ClickableImage = ({ href, src }) => {
 return (
   <a href={href}>
      <img src={src} />
   </a>
 );
};

<ClickableImage href="http://miu.edu" src="miu.png" />
```

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>
    )
  }
}
```

State Object

The **`state`** instance property is a special one because React will manage it.

You can only change it through **`setState()`** and React will trigger a render cycle.

`setState()` calls are **batched** and run **asynchronously**, we can pass an object to be **merged**, or a function that receives the previous state.

How State works?

```
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>
    );
  }

}
```

How State works?

```
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>
    );
  }

}
```

How State works?

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

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

}
```

How State works?

```
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>
    );
  }
}
```

React is Performant

The process by which React syncs changes in app state to the DOM is called **Reconciliation Algorithm**

- Reconstructs the virtual DOM
- Diffs the virtual DOM against the DOM
- Only makes the changes needed

It is important to remember that the reconciliation algorithm is an implementation detail. React could rerender the whole app on every action; the end result would be the same. Just to be clear, rerender in this context means calling render for all components, it doesn't mean React will unmount and remount them. It will only apply the differences.

Tree Reconciliation Algorithm

When we tell React to render a tree of elements in the browser, it first generates a virtual representation of that tree and keeps it around in memory for later. Then it'll proceed to perform the DOM operations that will make the tree show up in the browser.

When we tell React to update the tree of elements it previously rendered, it generates a new virtual representation of the updated tree. Now React has 2 versions of the tree in memory.

To render the updated tree in the browser, React does not discard what has already been rendered. Instead, it will compare the 2 virtual versions of the tree that it has in memory, compute the differences between them, figure out what sub-trees in the main tree need to be updated, and only update these sub-trees in the browser.

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.

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

```
ReactDOM.render(
  <div>
    <Comp1 />
    <Comp2 />
  </div>,
  mountNode
);
```

```
ReactDOM.render(
  <React.Fragment>
    <Comp1 />
    <Comp2 />
  </React.Fragment>,
  mountNode
);
```

Without introducing a new DOM parent node.

```
ReactDOM.render(
  <>
    <Comp1 />
    <Comp2 />
  </>,
  mountNode
);
```

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

Controlled Inputs

```
class App extends React.Component {
        state = {text: ''};

        handleChange = (event) => {
            const element = event.target;
            this.setState({text: element.value});
        };
        render(){
           return (
              <div>
                <input value={this.state.text} onChange={this.handleChange} />
                <div>{this.state.text}</div>
              </div>
            );
        }
    };
```

Tip: Always use the **`onSubmit`** event handler to handle a form submission (also useful when users hit **`enter`**) we can then **`preventDefault()`** and read the form values through refs or controlled elements.

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.

Side Effect Example

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.

When rendering an input form-element you might want to auto-focus a text box. That too has to be done with the DOM API.

Component Lifecycle Hooks

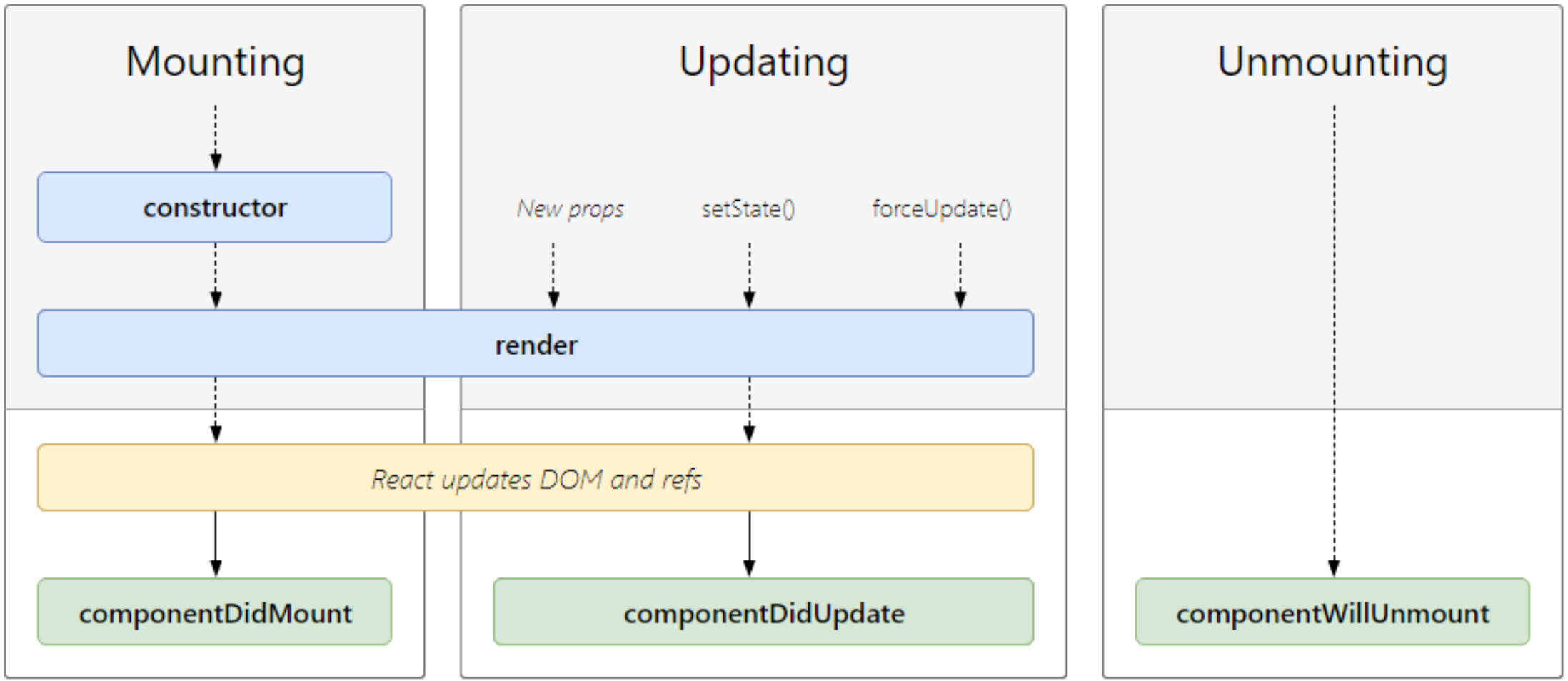
Side effects usually need to happen either before or after React render task.

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, and you can do things right before a component is removed from the browser inside a **`componentWillUnmount`** class method.

https://reactjs.org/docs/react-component.html

Component Lifecycle Hooks



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

Common Mistakes: Functions vs. Classes

```
class Numbers extends React.Component {
  const array = [];

  // ...
}


const Numbers = (props) => {
  const array = [];

  // ...
};
```

Common Mistakes: Passing Numbers

You can pass a prop value with a string:

```
<Greeting name="MSD" />
```

If you need to pass a numeric value, don’t use strings:

```
<Greeting counter="7" />
```

Instead, use curly brackets to pass an actual numeric value:

```
<Greeting counter={7} />
```

Common Mistakes: Curly Brackets vs. Parentheses

```
return {
  something;
};

return (
  something;
);
```

Common Mistakes: Implicit Return

```
this.setState(prevState => { answer: 42 });

this.setState(prevState => { return ({ answer: 42 }) });
this.setState(prevState => ({ answer: 42 }));
```

Setting an Object key dynamically

```
var str = 'name';
var obj1 = {str: 'mike'} // {str: 'mike'}
Var obj2 = {[str]: 'mike'} // {name: 'mike'}
```