Component Lifecycle

CS568 – Web Application Development I

Assistant Professor Umur Inan

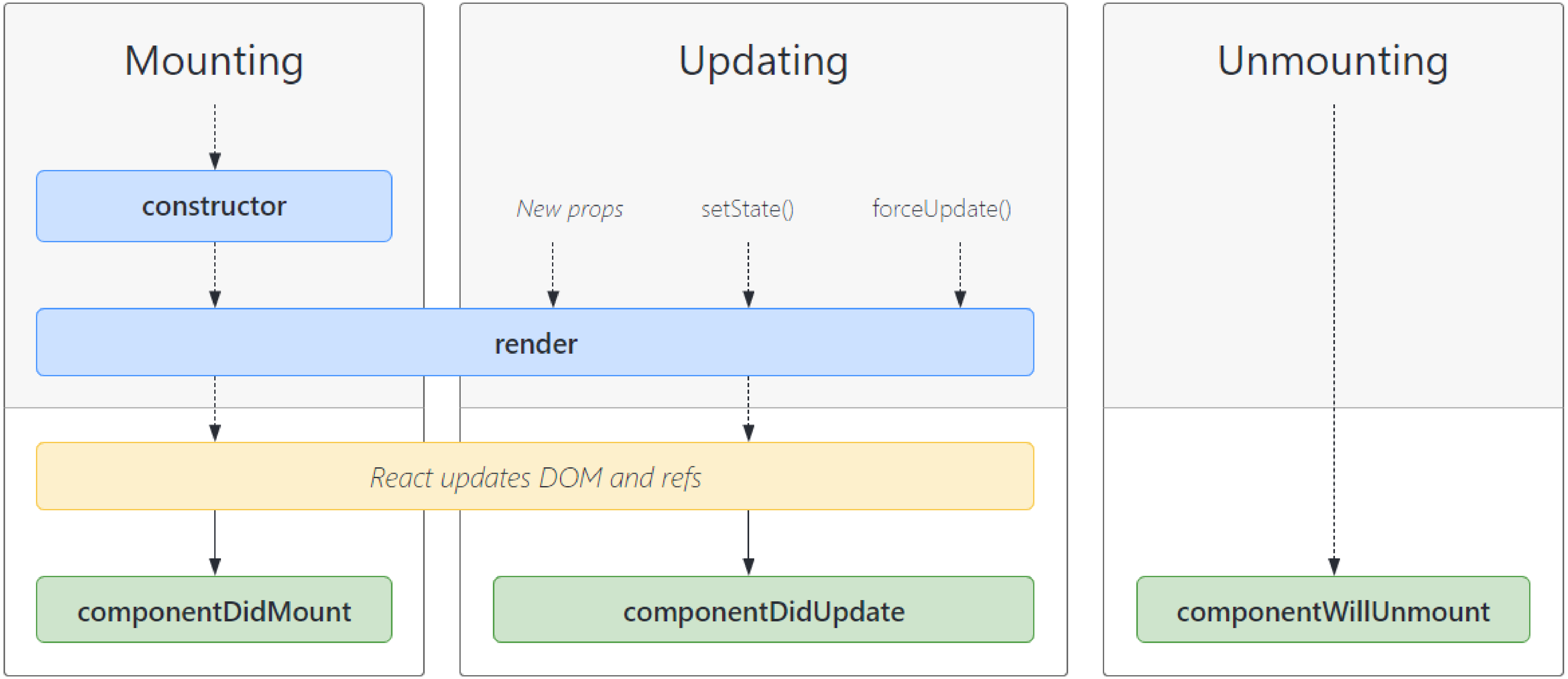
Computer Science Department

Maharishi International University

Maharishi International University - Fairfield, Iowa

Content

- Component lifecycle phases
  - Render phase
  - Commit phase
- Most common lifecycle methods
  - ComponentDidMount
  - ComponentDidUpdate
  - ComponentWillUnmount
- Other lifecycle methods



Refer: Common lifecycles

Component Lifecycle

Only available in Class-Based Components

- constructor
- getDerivedStateFromProps
- getSnapshotBeforeUpdate
- componentWillUnmount
- componentDidMount
- componentDidCatch
- shouldComponentUpdate
- componentDidUpdate
- render

Component Lifecycle - Creation

- Invoking Order

constructor(props)	Call super(props)
getDerivedStateFromProps(props,state)	When the props has changed, you can sync the state
render()	Prepare JSX code
componentDidMount	Can be used for making HTTP requests

Component Lifecycle - Update

- Invoking Order

getDerivedStateFromProps(props,state)

Performance

shouldComponentUpdate(nextProps,nextState)

May cancel update process

render()

getSnapshotBeforeUpdate(prevProps, prevState)

Last minute DOM operations

componentDidUpdate()

Component lifecycle phases

1. Render phase – Pure and has no side effects. Can be called multiple times by React. Must be stable and return the same result.
    - Constructor
    - Render
2. Commit phase – Run side effects. Will be called once.
    - ComponentDidMount
    - ComponentDidUpdate
    - ComponentWillUnmount

Side effects

Side effects are an action that impinges on the outside world. Examples:

- Making asynchronous API calls for data
- Updating global variables from inside a function
- Working with timers like setInterval or setTimeout
- Measuring the width or height or position of elements in the DOM
- Logging messages to the console or other service
- Setting or getting values in local storage

Render

Only required method in class.

It examines this.props and this.state and returns one of the following:

- React elements,
- Strings and numbers (rendered as text nodes in the DOM),
- Portals,
- null or Booleans (Render nothing).

The render() function should be **pure**, meaning that it does not modify component state, it returns the same result each time it's invoked.

componentDidMount

- This is invoked immediately after a component is mounted (inserted into the DOM tree).
- Here is the right place to initialize the DOM node. For example, **load data from the back-end service**.
- Here is the right place to set up any subscriptions (don't forget to unsubscribe in componentWillUnmount())
- You may call setState() immediately in componentDidMount() (Most times, developers do that!). It will trigger an **extra rendering**, but it will happen before the browser updates the screen. This guarantees that even though the render() will be called twice in this case, **the user won't see the intermediate state**. Use this pattern with caution because it often causes performance issues.

componentDidUpdate

- componentDidUpdate() is invoked immediately after updating occurs. This method is not called for the initial render.
- You may call setState() immediately in componentDidUpdate() but note that it must be wrapped **in a condition** like in the example below, or you'll cause an infinite loop.
- Don't copy props into state that causes bugs! Directly use props.

```
componentDidUpdate(prevProps) {
  // Typical usage (don't forget to compare props):
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

componentWillUnmount

- componentWillUnmount() is invoked immediately before a component is unmounted and destroyed.
- Perform any necessary **cleanup** in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in componentDidMount().

componentDidCatch

- There are two ways to catch JavaScript errors anywhere in their child component tree, **getDerivedStateFromError** and **componentDidCatch**. With these lifecycle methods, you can display a **fallback UI** instead of a broken component.
- This lifecycle is invoked after an error has been thrown by a descendant component. It receives two parameters:
  - error - The error that was thrown.
  - info - An object with a componentStack key containing information about which component threw the error.
- componentDidCatch() is called during the “commit” phase, so side-effects are permitted. It should be used for things like **logging errors** (info).
- A class component becomes an **Error Boundary** if it defines either (or both) of the lifecycle methods static getDerivedStateFromError() or componentDidCatch().

shouldComponentUpdate for Optimization

- Since ‘students’ component is a child element of App component when something is changed in App, students component is re-rendered.
- How can we avoid this?
- render() will not be invoked if shouldComponentUpdate() returns false!

shouldComponentUpdate for Optimization

```
//students.js
shouldComponentUpdate(nextProps,nextState){
       if(nextProps.students !== this.props.students){
           return true; //re-render
       }
       return false; //do not re-render
   }
```

Pure Components

- Instead of implementing shouldComponentUpdate method we can extend to PureComponent**.**
- Pure components prevent from re-rendering if props or state is the same.
- **It takes care of “shouldComponentUpdate” implicitly.**
- Pure Components are more performant.
- State and Props are shallowly compared.
- Shallow comparison checks the reference (address) not the value.

Unnecessary render

React re-renders when there is change in state and props.

When the parent component renders, React will recursively render all its child components, regardless of whether their props have changed or not. In other words, child components go through the render phase but not the commit phase. It is also known as **unnecessary render**.

There are techniques to prevent from unnecessary renders. One of them is to implement the should component update or extend to PureComponent.

Auxiliary Component

- It is higher order component.
- It is wrapper component.
- It does not render any html code.
- It is there to satisfy React restriction.

Auxiliary Component

```
//auxiliary.js


const Aux = props=>props.children
```

```
//auxiliary.js


const MyComponent =()=>{
  return(){
    <Aux>
      <p>....</p>
      <p>....</p>
    </Aux>
  }
}
```