Chapter 6: React Component Lifecycle

Lifecycle methods are to be used to run code and interact with your component at different points in the components life. These methods are based around a component Mounting, Updating, and Unmounting.

Section 6.1: Component Creation

When a React component is created, a number of functions are called:

- If you are using `React.createClass` (ES5), 5 user defined functions are called
- If you are using `class Component extends React.Component` (ES6), 3 user defined functions are called

`getDefaultProps()` (ES5 only)

This is the **first** method called.

Prop values returned by this function will be used as defaults if they are not defined when the component is instantiated.

In the following example, `this.props.name` will be defaulted to `Bob` if not specified otherwise:

```
getDefaultProps() {
  return {
    initialCount: 0,
    name: 'Bob'
  };
}
```

`getInitialState()` (ES5 only)

This is the **second** method called.

The return value of `getInitialState()` defines the initial state of the React component. The React framework will call this function and assign the return value to `this.state`.

In the following example, `this.state.count` will be intialized with the value of `this.props.initialCount`:

```
getInitialState() {
  return {
    count : this.props.initialCount
  };
}
```

`componentWillMount()` (ES5 and ES6)

This is the **third** method called.

This function can be used to make final changes to the component before it will be added to the DOM.

```
componentWillMount() {
  ...
}
```

`render()` (ES5 and ES6)

This is the **fourth** method called.

The `render()` function should be a pure function of the component's state and props. It returns a single element

which represents the component during the rendering process and should either be a representation of a native DOM component (e.g. `<p />`) or a composite component. If nothing should be rendered, it can return `null` or `undefined`.

This function will be recalled after any change to the component's props or state.

```
render() {
  return (
    <div>
      Hello, {this.props.name}!
    </div>
  );
}
```

`componentDidMount()` (ES5 and ES6)

This is the **fifth** method called.

The component has been mounted and you are now able to access the component's DOM nodes, e.g. via `refs`.

This method should be used for:

- Preparing timers
- Fetching data
- Adding event listeners
- Manipulating DOM elements

```
componentDidMount() {
  ...
}
```

ES6 Syntax

If the component is defined using ES6 class syntax, the functions `getDefaultProps()` and `getInitialState()` cannot be used.

Instead, we declare our `defaultProps` as a static property on the class, and declare the state shape and initial state in the constructor of our class. These are both set on the instance of the class at construction time, before any other React lifecycle function is called.

The following example demonstrates this alternative approach:

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

    this.state = {
      count: this.props.initialCount
    };
  }

  upCount() {
    this.setState((prevState) => ({
      count: prevState.count + 1
    }));
  }

  render() {
    return (
      <div>
```

```
        Hello, {this.props.name}!<br />
        You clicked the button {this.state.count} times.<br />
        <button onClick={this.upCount}>Click here!</button>
      </div>
    );
  }
}

MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Replacing `getDefaultProps()`

Default values for the component props are specified by setting the `defaultProps` property of the class:

```
MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Replacing `getInitialState()`

The idiomatic way to set up the initial state of the component is to set `this.state` in the constructor:

```
constructor(props){
  super(props);

  this.state = {
    count: this.props.initialCount
  };
}
```

Section 6.2: Component Removal

`componentWillUnmount()`

This method is called **before** a component is unmounted from the DOM.

It is a good place to perform cleaning operations like:

- Removing event listeners.
- Clearing timers.
- Stopping sockets.
- Cleaning up redux states.

```
componentWillUnmount(){
  ...
}
```

An example of removing attached event listener in `componentWillUnMount`

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

export default class SideMenu extends Component {
```

```
  constructor(props) {
    super(props);
    this.state = {
        ...
      };
    this.openMenu = this.openMenu.bind(this);
    this.closeMenu = this.closeMenu.bind(this);
  }

  componentDidMount() {
    document.addEventListener("click", this.closeMenu);
  }

  componentWillUnmount() {
    document.removeEventListener("click", this.closeMenu);
  }

  openMenu() {
    ...
  }

  closeMenu() {
    ...
  }

  render() {
    return (
      <div>
        <a
          href      = "javascript:void(0)"
          className = "closebtn"
          onClick   = {this.closeMenu}
        >
          ×
        </a>
        <div>
          Some other structure
        </div>
      </div>
    );
  }
}
```

Section 6.3: Component Update

`componentWillReceiveProps(nextProps)`

This is the **first function called on properties changes**.

When **component's properties change**, React will call this function with the **new properties**. You can access to the old props with *this.props* and to the new props with *nextProps*.

With these variables, you can do some comparison operations between old and new props, or call function because a property change, etc.

```
componentWillReceiveProps(nextProps){
  if (nextProps.initialCount && nextProps.initialCount > this.state.count){
    this.setState({
      count : nextProps.initialCount
    });
```

```
  }
}
```

shouldComponentUpdate(nextProps, nextState)

This is the **second function called on properties changes and the first on state changes**.

By default, if another component / your component change a property / a state of your component, **React** will render a new version of your component. In this case, this function always return true.

You can override this function and **choose more precisely if your component must update or not**.

This function is mostly used for **optimization**.

In case of the function returns **false**, the **update pipeline stops immediately**.

```
componentShouldUpdate(nextProps, nextState){
  return this.props.name !== nextProps.name ||
    this.state.count !== nextState.count;
}
```

componentWillUpdate(nextProps, nextState)

This function works like `componentWillMount()`. **Changes aren't in DOM**, so you can do some changes just before the update will perform.

/!\ : you cannot use **this.setState()**.

```
componentWillUpdate(nextProps, nextState){}
```

render()

There's some changes, so re-render the component.

componentDidUpdate(prevProps, prevState)

Same stuff as `componentDidMount()` : **DOM is refreshed**, so you can do some work on the DOM here.

```
componentDidUpdate(prevProps, prevState){}
```

Section 6.4: Lifecycle method call in different states

This example serves as a complement to other examples which talk about how to use the lifecycle methods and when the method will be called.

This example summarize Which methods (componentWillMount, componentWillReceiveProps, etc) will be called and in which sequence will be different for a component **in different states**:

When a component is initialized:

1. getDefaultProps
2. getInitialState
3. componentWillMount
4. render
5. componentDidMount

When a component has state changed:

1. shouldComponentUpdate

2. componentWillUpdate
3. render
4. componentDidUpdate

When a component has props changed:

1. componentWillReceiveProps
2. shouldComponentUpdate
3. componentWillUpdate
4. render
5. componentDidUpdate

When a component is unmounting:

1. componentWillUnmount

Section 6.5: React Component Container

When building a React application, it is often desirable to divide components based on their primary responsibility, into Presentational and Container components.
Presentational components are concerned only with displaying data - they can be regarded as, and are often implemented as, functions that convert a model to a view. Typically they do not maintain any internal state.
Container components are concerned with managing data. This may be done internally through their own state, or by acting as intermediaries with a state-management library such as Redux. The container component will not directly display data, rather it will pass the data to a presentational component.

```
// Container component
import React, { Component } from 'react';
import Api from 'path/to/api';

class CommentsListContainer extends Component {
    constructor() {
        super();
        // Set initial state
        this.state = { comments: [] }
    }

    componentDidMount() {
        // Make API call and update state with returned comments
        Api.getComments().then(comments => this.setState({ comments }));
    }

    render() {
        // Pass our state comments to the presentational component
        return (
            <CommentsList comments={this.state.comments} />;
        );
    }
}

// Presentational Component
const CommentsList = ({ comments }) => (
    <div>
        {comments.map(comment => (
            <div>{comment}</div>
        )}
    </div>
);
```

```
CommentsList.propTypes = {
    comments: React.PropTypes.arrayOf(React.PropTypes.string)
}
```