Lifecycle Methods

Relevel
by Unacademy

Topics Covered

- **React components**
  What are React components

- **Class Based Components**
  What are classes in React?

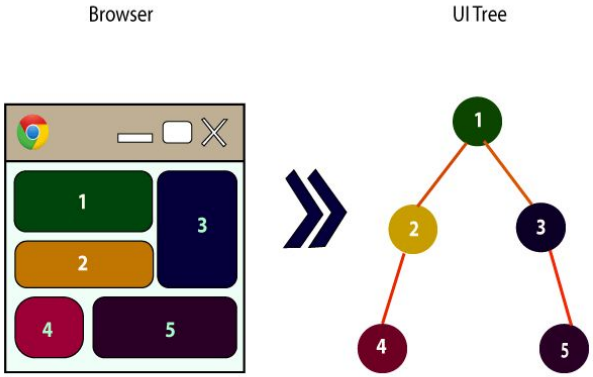
- **Functional Components**
  What are functional components and how are they different from class based components.

- **Lifecycle Methods**
  The birth, growth and death of components

- **Practice Questions**
  Assignments

React Components

- React.js is an Open Source JavaScript library for building UI.
- It follows a component based architecture, where each UI element can be composed of different reusable components..
- React has two types of components: Class based and Function based.
- Components in React have a built-in state object. The state is encapsulated data where we store assets that are persistent between component renderings.
- If a user changes state by interacting with our application, the UI may look completely different afterwards, because it's represented by this new state rather than the old state.
- To manage the behavior of the state, initializing it, updating it and modifying it as per requirements is called state management.



Class Based Components

- But in React, a class is a type of component that allows state management.
- Before Hooks, introduced in React 16.8, a class-based component was the only way to use and manage states.
- This is why a class was the only stateful component available at that time.
- Classes allowed us to effectively manage and modify the state.

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

class Unacademy extends Component{
    constructor(props){
        super(props);
    }

    render() {
        return(
            <div>
                <h1>Hello Everybody!</h1>
            </div>
        )
    }
}

export default Unacademy;
```

Class Based Components

A typical class based component contains the following:

- React component imported from React library for writing JSX.
- Component imported from React library for creating classes.
- A class component that extends the Component of React
- A constructor to initialize the props.
- A render method to paint the UI.
- An export named after the class name.
- In this example, Unacademy is the name of our class component, which renders a heading called Hello Everybody!

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

class Unacademy extends Component{
    constructor(props){
        super(props);
    }

    render() {
        return(
            <div>
                <h1>Hello Everybody!</h1>
            </div>
        )
    }
}

export default Unacademy;
```

Functional Components

- A functional component is similar to a JavaScript Function.
- It starts with the function keyword followed by a name with parentheses and curly braces.
- Prior to React Hooks being introduced in the 16.5 version of React, functional components didn't provide state management.
- Thus functional components were called stateless components.
- With the introduction of Hooks, we can pretty much do everything we can do in a class component, but much faster with less lines of code.

```
import React from 'react.'


function Unacademy (props){



        return(
            <div>
                <h1>Hello Everybody!</h1>
            </div>
        )

}

export default Unacademy;
```

Functional Components

A typical functional component contains the following:

- React component imported from reacting library for writing JSX.
- A function declaration followed by a function name should start with a capital letter.
- A parameter is passed in between parentheses, which in React is props.
- A return method contains JSX.
- An export named after the function name.

ES 6 Functional Components

- We can also write functional components following ES 6 syntax, i.e. arrow functions.
- It loses the function keyword, while everything else remains the same.

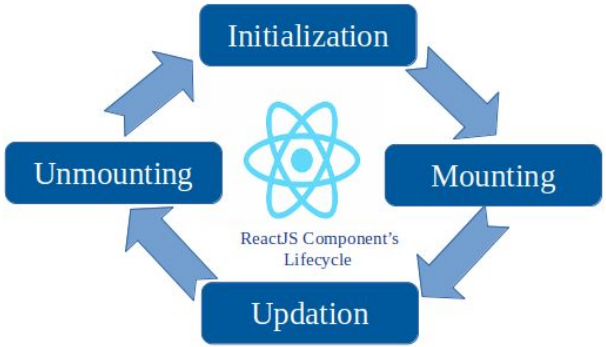
```
import React from 'react.'


const Unacademy = (props) => {



        return(
            <div>
                <h1>Hello Everybody!</h1>
            </div>
        )

}

export default Unacademy;
```

Lifecycle Methods

In React.js, the components essentially go through three primary stages:

- Mounting(Inserting elements into DOM)
- Updating(Modifying elements in DOM)
- Unmounting(Removing elements from DOM)

You can think of these as a component's birth, growth, and death.



Mounting

Whenever a React component is created and inserted into DOM, we get to see them on the browser or User Interface; they go through the following methods in the order they are mentioned here:

- constructor.
- static getDerivedStateFromProps()
- render()
- componentDidMount()

Mounting

Class-based components currently support more lifecycle methods; thus we will learn the methods via class components first.

We will understand the lifecycle methods via an example:

```
class Unacademy extends
React.Component {
  constructor() {
    super();
    this.state={}
    console.log("Constructor is
called");
  }
```

Mounting

- The getDerivedStateFromProps is the second method that is called; it is invoked right before calling the render method, both on the initial mount and subsequent updates. It must return an object to update the state; if nothing is to be updated, it must return null.

- It is a rarely used method, and is only called if state derivation is required.

```
static
getDerivedStateFromProps(props,
state) {

console.log("getDerivedStatesFro
mProps is called");
    return null;
  }
```

Mounting

- The render is the third method; it is the only essential method required for, well, basically rendering our class component.
- It is primarily used for printing the JSX in DOM.
- It is called whenever there is a change in state or props.

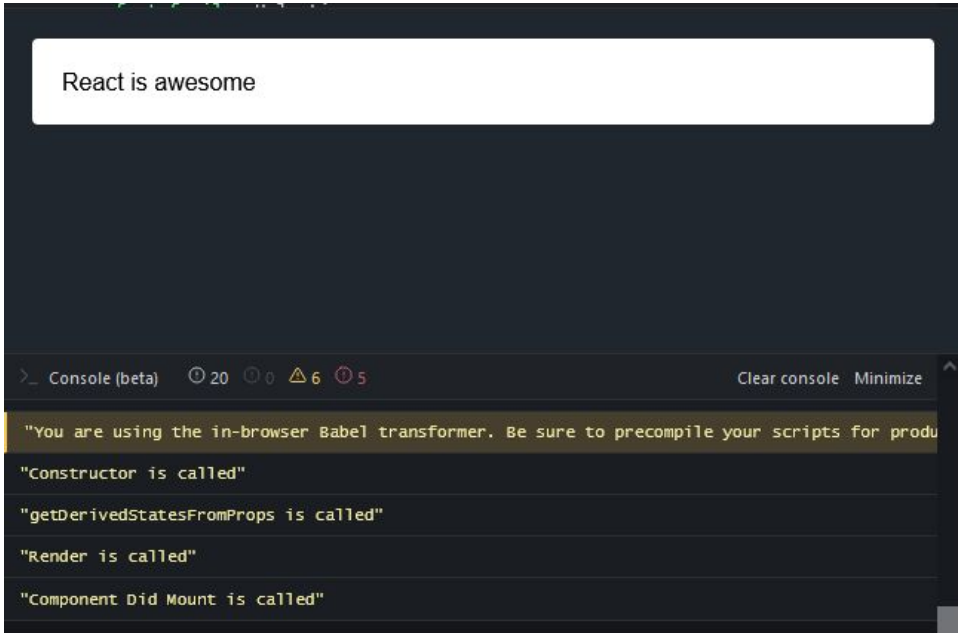
```
render() {
    console.log("Render is
called");
    return (
      <div>
        <h1>React is
awesome</h1>
      </div>
    );
  }
```

Mounting

- componentDidMount() is the last lifecycle method in the mounting stage that is invoked immediately after a component is inserted into the DOM.
- It is called only once.
- It is primarily used for making API calls.

```
componentDidMount() {
    console.log("Component Did
Mount is called");
  }
}
  }
```

Output


React is awesome
Console (beta)
20
6
5
Clear console
Minimize
"You are using the in-browser Babel transformer. Be sure to precompile your scripts for produ
"Constructor is called"
"getDerivedStatesFromProps is called"
"Render is called"
"Component Did Mount is called"

Updating

Whenever a React component is subjected to an update, caused due to changes in props or state. These methods are called in the following order when a component is being re-rendered:

- static getDerivedStateFromProps()
- shouldComponentUpdate()
- render()
- getSnapshotBeforeUpdate()
- componentDidUpdate()

Unmounting

Unmounting is the last stage of a component's lifecycle; it signifies that the component no longer exists in the DOM.

- It contains only one method, and that is componentWillUnmount().
- **componentWillUnmount()** is invoked immediately before a component is unmounted and destroyed.
- It is primarily used to clean up our code, such as cancelling API calls or cleaning up any subscriptions created in componentDidMount().

Example:

- We will take an example to learn the Updating & Unmounting phase's methods.
- **getDerivedStateFromProps** is the first method to be called whenever the state or props of a component changes.

```
class App extends React.Component {
  constructor(){
    super();
    console.log('* constructor');
    this.state = {
      count: 0,
    }
  }
  componentDidMount(){ console.log('* componentDidMount'); }
  static getDerivedStateFromProps(){ console.log('*
getDerivedStateFromProps'); }
```

Example:

- **shouldComponentUpdate()** is used for letting React know if a component's output is not affected by the current change in state or props. The default behaviour is to re-render on every state change.
- **getSnapshotBeforeUpdate()** is invoked right before the most recently rendered output is added to the DOM. It provides provision to our component to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this method will be passed to componentDidUpdate().
- **componentDidUpdate()** is the last method to be called during the updation stage; similar to componentDidMount(), it is called only once and when it is called, it signifies that the DOM has been updated.
- ComponentDidUpdate() is primarily used for operating on DOM elements after the DOM has been updated.
- **componentWillUnmount()** is invoked immediately before a component is unmounted and destroyed.
- It is primarily used to clean up our code, such as canceling API calls or cleaning up any subscriptions created in componentDidMount().

```
shouldComponentUpdate(){ console.log('*
shouldComponentUpdate'); return true; }

  getSnapshotBeforeUpdate(){ console.log('*
getSnapshotBeforeUpdate'); }

  componentDidUpdate(){ console.log('*
componentDidUpdate'); }

  componentWillUnmount(){ console.log('*
componentWillUnmount'); }
```

```
render() {
    console.log('* render');
    return (
      <div>
        <h1>React lifecycle example</h1>
        <p>View lifecycle events on the right. Click the button to trigger a component update.</p>
        <button onClicka={() => { console.log(''); this.setState({ count: this.state.count+1 }); }}>
          Trigger an update (Counter {this.state.count})
        </button>
      </div>
    );
  }
```

```
ReactDOM.render(
  <App />,
  document.getElementById('app')
);
```

Output


```
* constructor
* getDerivedStateFromProps
* render
* componentDidMount

* getDerivedStateFromProps
* shouldComponentUpdate
* render
* getSnapshotBeforeUpdate
* componentDidUpdate

* getDerivedStateFromProps
* shouldComponentUpdate
* render
* getSnapshotBeforeUpdate
* componentDidUpdate
```

Lifecycle of Functional Components

- Functional components in React do not get access to the plethora of lifecycle methods that we covered for class-based components above.
- However, with the introduction of Hooks, we have one method that helps us mimic some of the class component lifecycle methods' behaviour.
- There is one such hook called useEffect.
- useEffect works typically like componentDidMount(); it is called once after the function has finished rendering.

Syntax:

```
useEffect(() => {
        effect

    }, [ArrayOfDependencies])
```

Example

- It takes a callback as the first parameter where we can make API calls or subscriptions like we do in componentDidMount.
- It takes an array as a second parameter which stores the dependencies.
- If the array is empty, useEffect will behave exactly like componentDidMount()
- If the array contains dependencies, it will behave exactly like componentDidUpdate() i.e. it will update whenever there is any change observed to the array elements.

```
const App = (props) => {
  useEffect(() => {
    console.log('After rendering is
done')
  },[])

  return(
    <>
    <h1>React is fun</h1>
    </>
  )
}
```

useEffect Customizations

- In the following example, everytime there is a change in the car variable, useEffect will be called.

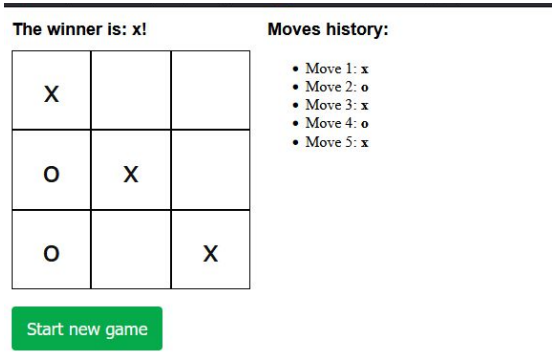
```
useEffect(() => {
        let car = “Nexon”;

    }, [car])
```

- componentWillUnmount()- To replicate the behavior of componentWillUnmount, we need to add event listeners and clean them up before the component is ejected from DOM like this:

```
useEffect(() => {
        window.addEventListener("mousemove", () => {});
        return () => {
          window.removeEventListener("mousemove", () => {})
        }
      }, []);
```

App Building : Tic Tac Toe App

1. In this lesson, we will build a Tic-Tac-toe application using React.js.
2. The app would allow the two people to play turn by turn; it will record each move of the players and determine the winner once a result is achieved.



Assignment

- Create an app using useEffect Hook that demonstrates the lifecycle methods of class components.

Thank You