Class Component and State | Cheat Sheet

Concepts in Focus

1. Components

There are two ways to write React Components.

They are:

- Functional Components
- Class Components

1.1 Functional Components

These are JavaScript functions that take props as a parameter if necessary and return react element (JSX).

```
const Welcome = () => <h1>Hello, User</h1>;

export default Welcome;
```

1.2 Class Components

These components are built using an ES6 class.

To define a React Class Component,

- Create an ES6 class that extends `React.Component` .
- Add a single empty method to it called `render()` .

1.2.1 extends

The

`extends` keyword is used to inherit methods and properties from the `React.Component` .

1.2.2 render()

The

`render()` method is the only required method in a class component. It returns the JSX element.

Syntax:

```
import { Component } from "react";

class MyComponent extends Component {
  render() {
    return JSX;
  }
}
```

Use

`this.props` in the `render()` body to access the props in a class component.

```
class Welcome extends Component {
 render() {
   const { name } = this.props
   return <h1>Hello, {name}</h1>
 }
}
```

Note

The component name should always be in the pascal case.

2. React Events

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

1. React events are named using **camelCase**, rather than **lowercase**.

Example:

HTML	JSX
onclick	onClick
onblur	onBlur
onchange	onChange

2. With JSX, you pass a **function** as the event handler rather than a **string**.

Example:

```
<button onclick="activateLasers()">Activate Lasers</button>
```

```
<button onClick={activateLasers}>Activate Lasers</button>
```

We should not call the function when we add an event in JSX.

```
class MyComponent extends Component {
  handleClick = () => {
    console.log("clicked")
  }
  render() {
    return <button onClick={this.handleClick()}>Click Me</button>
  }
}
```

In the above function, the

`handleClick` is called instead of passed as a reference.

```
class MyComponent extends Component {
  handleClick = () => {
    console.log("clicked")
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

In the above function, the

`handleClick` is passed as a reference. So, the function is not being called every time the component renders.

Providing Arrow Functions

To not change the context of

`this` , we have to pass an arrow function to the event.

```
class MyComponent extends Component {
  handleClick() {
    console.log(this)  // undefined
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

```
class MyComponent extends Component {
  handleClick = () => {
    console.log(this)  // MyComponent {...}
  }
  render() {
```

```
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

3. State

The state is a JS object in which we store the component's data that changes over time.

When the state object changes, the component re-renders.

Intialising State:

```
class Counter extends Component {
  state = { count: 0 }
  render() {
    const { count } = this.state;
    return <p className="count">{count}</p>;
  }
}
```

3.1 Updating State

We can update the state by using

`setState()` . We can provide function/object as an argument to set the state.

Providing Function as an Argument:

Syntax:

`this.setState( prevState => ({... }) )`

Here the previous state is sent as a parameter to the callback function.

```
onIncrement = () => {
  this.setState((prevState) =>
    console.log(`previous state value ${prevState.count}`)
  )
}
```

3.2 State Updates are Merged

State updates are merged. It means that when you update only one key-value pair in the state object, it will not affect the other key-value pai object.

```
// For example let's say your state is as followed:
state = { key1 : value1, key2 : value2 }

// if you use this.setState such as :
this.setState((prevState) => ({ prevState.key1 : value3 }))

// your new state will be :
state = { key1 : value3, key2 : value2 }
```

3.3 Functional Components vs Class Components

Functional Components	Class Components
Renders the UI based on props	Renders the UI based on props and state

Use Class Components whenever the state is required. Otherwise, use the Functional components.

4. Counter Application

File: src/App.js

```
import Counter from "./components/Counter";

const App = () => {
  return <Counter />
}

export default App;
```

File: src/components/Counter/index.js

```
import { Component } from "react"

import "./index.css"

class Counter extends Component {
  state = { count: 0 }
  onIncrement = () => {
    this.setState((prevState) => ({ count: prevState.count + 1 }))
  }
  onDecrement = () => {
    this.setState((prevState) => ({ count: prevState.count - 1 }))
  }
  render() {
    const { count } = this.state
    return (
      <div className="container">
        <h1 className="count">Count {count}</h1>
        <button className="button" onClick={this.onIncrement}>
          Increase
        </button>
        <button className="button" onClick={this.onDecrement}>
          Decrease
        </button>
```

```
      </div>
    )
  }
}

export default Counter
```

Submit Feedback

Notes Discussions