React Conditional Rendering

In React, we can create multiple components which encapsulate behavior that we need. After that, we can render them depending on some conditions or the state of our application. In other words, based on one or several conditions, a component decides which elements it will return. In React, conditional rendering works the same way as the conditions work in JavaScript. We use JavaScript operators to create elements representing the current state, and then React Component update the UI to match them.

From the given scenario, we can understand how conditional rendering works. Consider an example of handling a **login/logout** button. The login and logout buttons will be separate components. If a user logged in, render the **logout component** to display the logout button. If a user not logged in, render the **login component** to display the login button. In React, this situation is called as **conditional rendering**.

There is more than one way to do conditional rendering in React. They are given below.

- if
- ternary operator
- logical && operator
- switch case operator
- Conditional Rendering with enums

if

It is the easiest way to have a conditional rendering in React in the render method. It is restricted to the total block of the component. IF the condition is **true**, it will return the element to be rendered. It can be understood in the below example.

Example

```
function UserLoggin(props) {
  return <h1>Welcome back!</h1>;
}
function GuestLoggin(props) {
  return <h1>Please sign up.</h1>;
}
function SignUp(props) {
  const isLoggedIn = props.isLoggedIn;
```

```
  if (isLoggedIn) {
    return <UserLogin />;
  }
  return <GuestLogin />;
}

ReactDOM.render(
  <SignUp isLoggedIn={false} />,
  document.getElementById('root')
);
```

Logical && operator

This operator is used for checking the condition. If the condition is **true**, it will return the element **right** after **&&**, and if it is **false**, React will **ignore** and skip it.

Syntax

```
{
   condition &&
   // whatever written after && will be a part of output.
}
```

We can understand the behavior of this concept from the below example.

If you run the below code, you will not see the **alert** message because the condition is not matching.

```
('javatpoint' == 'JavaTpoint') && alert('This alert will never be shown!')
```

If you run the below code, you will see the **alert** message because the condition is matching.

```
(10 > 5) && alert('This alert will be shown!')
```

Example

```
import React from 'react';
import ReactDOM from 'react-dom';
// Example Component
function Example()
{
   return(<div>
         {
            (10 > 5) && alert('This alert will be shown!')
         }
         </div>
      );
}
```

You can see in the above output that as the condition **(10 > 5)** evaluates to true, the alert message is successfully rendered on the screen.

Ternary operator

The ternary operator is used in cases where two blocks alternate given a certain condition. This operator makes your if-else statement more concise. It takes **three** operands and used as a shortcut for the if statement.

Syntax

```
condition ?  true : false
```

If the condition is **true**, **statement1** will be rendered. Otherwise, **false** will be rendered.

Example

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
```

```
    <div>
      Welcome {isLoggedIn ? 'Back' : 'Please login first'}.
    </div>
  );
}
```

Switch case operator

Sometimes it is possible to have multiple conditional renderings. In the switch case, conditional rendering is applied based on a different state.

Example

```
function NotificationMsg({ text}) {
  switch(text) {
    case 'Hi All':
      return <Message: text={text} />;
    case 'Hello JavaTpoint':
      return <Message text={text} />;
    default:
      return null;
  }
}
```

Conditional Rendering with enums

An **enum** is a great way to have a multiple conditional rendering. It is more **readable** as compared to switch case operator. It is perfect for **mapping** between different **state**. It is also perfect for mapping in more than one condition. It can be understood in the below example.

Example

```
function NotificationMsg({ text, state }) {
  return (
    <div>
```

```
      {{
        info: <Message text={text} />,
        warning: <Message text={text} />,
      }[state]}
    </div>
  );
}
```

Conditional Rendering Example

In the below example, we have created a **stateful** component called **App** which maintains the login control. Here, we create three components representing Logout, Login, and Message component. The stateful component App will render either or depending on its current **state**.

```
import React, { Component } from 'react';
// Message Component
function Message(props)
{
   if (props.isLoggedIn)
      return <h1>Welcome Back!!!</h1>;
   else
      return <h1>Please Login First!!!</h1>;
}
// Login Component
function Login(props)
{
  return(
        <button onClick = {props.clickInfo}> Login </button>
    );
}
// Logout Component
function Logout(props)
{
   return(
        <button onClick = {props.clickInfo}> Logout </button>
    );
```

```
}
class App extends Component{
  constructor(props)
  {
     super(props);
     this.handleLogin = this.handleLogin.bind(this);
      this.handleLogout = this.handleLogout.bind(this);
    this.state = {isLoggedIn : false};
  }
  handleLogin()
  {
     this.setState({isLoggedIn : true});
  }
  handleLogout()
  {
     this.setState({isLoggedIn : false});
  }
  render(){
     return(
        <div>
     <h1> Conditional Rendering Example </h1>
           <Message isLoggedIn = {this.state.isLoggedIn}/>
           {
              (this.state.isLoggedIn)?(
              <Logout clickInfo = {this.handleLogout} />
              ) : (
              <Login clickInfo = {this.handleLogin} />
              )
           }
        </div>
        );
  }
}
export default App;
```

Output: