

React State And Props

React State

- The state is an updatable structure that is used to contain data or information about the component.
- The state in a component can change over time. The change in state over time can happen as a response to user action or system event.
- A component with the state is known as stateful components.
- It is the heart of the react component which determines the behavior of the component and how it will render.
- They are also responsible for making a component dynamic and interactive.

React State

- A state must be kept as simple as possible.
- It can be set by using the **setState()** method and calling setState() method triggers UI updates.
- A state represents the component's local state or information.
- It can only be accessed or modified inside the component or by the component directly.
- To set an initial state before any interaction occurs, we need to use the **getInitialState()** method.

Defining State

- To define a state, you have to first declare a default set of values for defining the component's initial state.
- To do this, add a class constructor which assigns an initial state using `this.state`.
- The '**this.state**' property can be rendered inside **render()** method.

Example

- The below sample code shows how we can create a stateful component using ES6 syntax.

```
import React, { Component } from 'react';  
class App extends React.Component {  
  constructor() {  
    super();  
    this.state = { displayBio: true };  
  }  
}
```

```
render() {  
  const bio = this.state.displayBio ? (  
    <div>  
      <p><h3>welcome to the world of react </h3></p>  
    </div>  
  ) : null;  
  return (  
    <div>  
      <h1> Welcome to reactjs!! </h1>  
      { bio }  
    </div>  
  );  
}  
}  
export default App;
```

React State

- To set the state, it is required to call the `super()` method in the constructor. It is because `this.state` is uninitialized before the `super()` method has been called.

Changing the State

- We can change the component state by using the `setState()` method and passing a new state object as the argument.
- Now, create a new method `toggleDisplayBio()` in the above example and bind `this` keyword to the `toggleDisplayBio()` method otherwise we can't access `this` inside `toggleDisplayBio()` method.

`this.toggleDisplayBio = this.toggleDisplayBio.bind(this);`

Example

- In this example, we are going to add a **button** to the **render()** method. Clicking on this button triggers the `toggleDisplayBio()` method which displays the desired output.

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor() {
    super();
    this.state = { displayBio: false };
    console.log('Component this', this);
    this.toggleDisplayBio = this.toggleDisplayBio.bind(this);
  }
  toggleDisplayBio(){
    this.setState({displayBio: !this.state.displayBio});
  }
}
```

```
render() {  
  return (  
    <div>  
      <h1>Welcome to Reactjs!!</h1>  
      {  
        this.state.displayBio ? (  
          <div>  
            <p><h4>welcome to the world of react js</h4>
```

</p>

<button onClick={this.toggleDisplayBio}> Show Less </button>

</div>

): (

<div>

<button onClick={this.toggleDisplayBio}> Read More </button>

</div>

)

}

</div>

)

}

}

export default App;

React Props

- Props stand for "**Properties**."
- They are **read-only** components.
- It is an object which stores the value of attributes of a tag and work similar to the HTML attributes.
- It gives a way to pass data from one component to other components.
- It is similar to function arguments.
- Props are passed to the component in the same way as arguments passed in a function.

React Props

- Props are **immutable** so we cannot modify the props from inside the component.
- Inside the components, we can add attributes called props.
- These attributes are available in the component as **this.props** and can be used to render dynamic data in our render method.
- When you need immutable data in the component, you have to add props to **ReactDOM.render()** method in the **main.js** file of your ReactJS project and used it inside the component in which you need. It can be explained in the below example.

Example

App.js

```
import React, { Component } from 'react';
class App extends React.Component {
  render() {
    return (
      <div>
        <h1> Welcome to { this.props.name } </h1>
        <p> <h4> welcome to the world of reactjs </h4> </p>
      </div>
    );
  }
}
export default App;
```

Main.js

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';
```

```
import App from './App.js';
```

```
ReactDOM.render(<App name = "ReactJS!!" />, document.getElementById('app'));
```

Default Props

- It is not necessary to always add props in the `ReactDOM.render()` element. You can also set **default** props directly on the component constructor. It can be explained in the below example.

Example

App.js

```
import React, { Component } from 'react';
class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Default Props Example</h1>
        <h3>Welcome to {this.props.name}</h3>
        <p>welcome to the world of ReactJs</p>
      </div>
    );
  }
}
App.defaultProps = {
  name: "ReactJs"
}
export default App;
```

Main.js

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';
```

```
import App from './App.js';
```

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

State and Props

- It is possible to combine both state and props in your app.
- You can set the state in the parent component and pass it in the child component using props.
- It can be shown in the below example.

Example

App.js

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

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

    this.state = {
      name: "ReactJs",
    }
  }
  render() {
    return (
      <div>
        <JTP jtpProp = {this.state.name}/>
      </div>
    );
  }
}
```

- **class JTP extends** React.Component {
- render() {
- **return** (
- <div>
- <h1>State & Props Example</h1>
- <h3>Welcome to {**this**.props.jtpProp}</h3>
- <p>Welcome to the world of RactJS</p>
- </div>
-);
- }
- }
- **export default** App;

Main.js

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';
```

```
import App from './App.js';
```

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

React Props Validation

- Props are an important mechanism for passing the **read-only** attributes to React components.
- The props are usually required to use correctly in the component.
- If it is not used correctly, the components may not behave as expected. Hence, it is required to use **props validation** in improving react components.

React Props Validation

- Props validation is a tool that will help the developers to avoid future bugs and problems.
- It is a useful way to force the correct usage of your components.
- It makes your code more readable.
- React components used special property **PropTypes** that help you to catch bugs by validating data types of values passed through props, although it is not necessary to define components with propTypes. However, if you use propTypes with your components, it helps you to avoid unexpected bugs.

Validating Props

- **App.propTypes** is used for props validation in react component.
- When some of the props are passed with an invalid type, you will get the warnings on JavaScript console.
- After specifying the validation patterns, you will set the App.defaultProps.

Syntax:

```
class App extends React.Component {  
    render() {}  
}  
Component.propTypes = { /*Definition */};
```

ReactJS Props Validator

SN	PropType	Description
1.	PropTypes.any	The props can be of any data type.
2.	PropTypes.array	The props should be an array.
3.	PropTypes.bool	The props should be a boolean.
4.	PropTypes.func	The props should be a function.
5.	PropTypes.number	The props should be a number.
6.	PropTypes.object	The props should be an object.
7.	PropTypes.string	The props should be a string.
8.	PropTypes.symbol	The props should be a symbol.
9.	PropTypes.instanceOf	The props should be an instance of a particular JavaScript class.
10.	PropTypes.isRequired	The props must be provided.
11.	PropTypes.element	The props must be an element.
12.	PropTypes.node	The props can render anything: numbers, strings, elements or an array (or fragment) containing these types.
13.	PropTypes.oneOf()	The props should be one of several types of specific values.
14.	PropTypes.oneOfType([PropTypes.string, PropTypes.number])	The props should be an object that could be one of many types.

Example

- Here, we are creating an App component which contains all the props that we need.
- In this example, **App.propTypes** is used for props validation.
- For props validation, you must have to add this line:

import PropTypes from 'prop-types' in App.js file.

App.js

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
class App extends React.Component {
  render() {
    return (
      <div>
        <h1>ReactJS Props validation example</h1>
        <table>
          <tr>
            <th>Type</th>
            <th>Value</th>
            <th>Valid</th>
          </tr>
```

<tr>

<td>Array</td>

<td>{**this**.props.propArray}</td>

<td>{**this**.props.propArray ? "true" : "False"}</td>

</tr>

<tr>

<td>Boolean</td>

<td>{**this**.props.propBool ? "true" : "False"}</td>

<td>{**this**.props.propBool ? "true" : "False"}</td>

</tr>

<tr>

<td>Function</td>

<td>{**this**.props.propFunc(5)}</td>

<td>{**this**.props.propFunc(5) ? "true" : "False"}</td>

</tr>

<tr>

<td>String</td>

<td>{**this**.props.propString}</td>

<td>{**this**.props.propString ? "true" : "False"}</td>

</tr>

<tr>

```
<tr>
  <td>Number</td>
  <td>{this.props.propNumber}</td>
  <td>{this.props.propNumber ? "true" : "False"}</td>
</tr>
</table>
</div>
);
}
}
App.propTypes = {
  propArray: PropTypes.array.isRequired,
  propBool: PropTypes.bool.isRequired,
  propFunc: PropTypes.func,
  propNumber: PropTypes.number,
  propString: PropTypes.string,
}
```



```
App.defaultProps = {  
  propArray: [1,2,3,4,5],  
  propBool: true,  
  propFunc: function(x){return x+5},  
  propNumber: 1,  
  propString: "JavaTpoint",  
}  
export default App;
```

Main.js

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';
```

```
import App from './App.js';
```

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

Output:



A screenshot of a web browser window. The address bar shows 'localhost:8080'. The page title is 'ReactJS Props validation example'. Below the title is a table with three columns: 'Type', 'Value', and 'Valid'. The table contains five rows of data, all showing 'true' in the 'Valid' column.

Type	Value	Valid
Array	12345	true
Boolean	true	true
Function	10	true
String	JavaTpoint	true
Number	1	true

ReactJS Custom Validators

- ReactJS allows creating a custom validation function to perform custom validation. The following argument is used to create a custom validation function.
- **props:** It should be the first argument in the component.
- **propName:** It is the propName that is going to validate.
- **componentName:** It is the componentName that are going to validated again.

Example

```
var Component = React.createClass({  
  App.propTypes = {  
    customProp: function(props, propName, componentName) {  
      if (!item.isValid(props[propName])) {  
        return new Error('Validation failed!');  
      }  
    }  
  }  
})
```

State Vs. Props

SN	Props	State
1.	Props are read-only.	State changes can be asynchronous.
2.	Props are immutable.	State is mutable.
3.	Props allow you to pass data from one component to other components as an argument.	State holds information about the components.
4.	Props can be accessed by the child component.	State cannot be accessed by child components.
5.	Props are used to communicate between components.	States can be used for rendering dynamic changes with the component.
6.	Stateless component can have Props.	Stateless components cannot have State.
7.	Props make components reusable.	State cannot make components reusable.
8.	Props are external and controlled by whatever renders the component.	The State is internal and controlled by the React Component itself.

React Constructor

- The constructor is a method used to initialize an object's state in a class.
- It automatically called during the creation of an object in a class.
- The concept of a constructor is the same in React. The constructor in a React component is called before the component is mounted.
- When you implement the constructor for a React component, you need to call **super(props)** method before any other statement.
- If you do not call super(props) method, **this.props** will be undefined in the constructor and can lead to bugs.

Syntax

```
Constructor(props){  
  super(props);  
}
```

- In React, constructors are mainly used for two purposes:
 - 1.It used for initializing the local state of the component by assigning an object to this.state.
 - 2.It used for binding event handler methods that occur in your component.

React Constructor

- Note: If you neither initialize state nor bind methods for your React component, there is no need to implement a constructor for React component.
- You cannot call **setState()** method directly in the **constructor()**.
- If the component needs to use local state, you need directly to use '**this.state**' to assign the initial state in the constructor.
- The constructor only uses `this.state` to assign initial state, and all other methods need to use `set.state()` method.

Example

App.js

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

```
class App extends Component {  
  constructor(props){  
    super(props);  
    this.state = {  
      data: 'www.reactjs.com'  
    }  
    this.handleEvent = this.handleEvent.bind(this);  
  }  
  handleEvent(){  
    console.log(this.props);  
  }  
}
```

```
render() {  
  return (  
    <div className="App">  
      <h2>React Constructor Example</h2>  
      <input type ="text" value={this.state.data} />  
      <button onClick={this.handleEvent}>Please Click</button>  
    </div>  
  );  
}  
}  
export default App;
```

Main.js

```
import React from 'react';
```

```
import ReactDOM from 'react-dom';
```

```
import App from './App.js';
```

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

The most common question related to the constructor are:

1. Is it necessary to have a constructor in every component?

- No, it is not necessary to have a constructor in every component. If the component is not complex, it simply returns a node.

```
class App extends Component {  
  render () {  
    return (  
      <p> Name: { this.props.name }</p>  
    );  
  }  
}
```

2. Is it necessary to call `super()` inside a constructor?

- Yes, it is necessary to call `super()` inside a constructor. If you need to set a property or access `'this'` inside the constructor in your component, you need to call `super()`.

```
class App extends Component {  
  constructor(props){  
    this.fName = "Jhon"; // 'this' is not allowed before super()  
  }  
  render () {  
    return (  
      <p> Name: { this.props.name }</p>  
    );  
  }  
}
```

When you run the above code, you get an error saying **'this' is not allowed before super()**. So if you need to access the props inside the constructor, you need to call `super(props)`.

1) The constructor is used to initialize state.

```
class App extends Component {  
  constructor(props){  
    // here, it is setting initial value for 'inputTextValue'  
    this.state = {  
      inputTextValue: 'initial value',  
    };  
  }  
}
```

2) Using 'this' inside constructor

```
class App extends Component {  
  constructor(props) {  
    // when you use 'this' in constructor, super() needs to be called first  
  
    super();  
  
    // it means, when you want to use 'this.props' in constructor, call it as  
    below  
    super(props);  
  }  
}
```


3) Initializing third-party libraries

```
class App extends Component {  
  constructor(props) {  
  
    this.myBook = new MyBookLibrary();  
  
    //Here, you can access props without using 'this'  
    this.Book2 = new MyBookLibrary(props.environment);  
  }  
}
```

4) Binding some context(this) when you need a class method to be passed in props to children.

```
class App extends Component {
```

```
  constructor(props) {
```

```
    // when you need to 'bind' context to a function
```

```
    this.handleFunction = this.handleFunction.bind(this);
```

```
  }
```

```
}
```

React Component API

- ReactJS component is a top-level API. It makes the code completely individual and reusable in the application. It includes various methods for:
 - Creating elements
 - Transforming elements
 - Fragments

React Component API

- the three most important methods available in the React component API.
 1. `setState()`
 2. `forceUpdate()`
 3. `findDOMNode()`

setState()

- This method is used to update the state of the component.
- This method does not always replace the state immediately.
- Instead, it only adds changes to the original state.
- It is a primary method that is used to update the user interface(UI) in response to event handlers and server responses.

setState()

- Syntax
- **this.setState(object newState[, function callback]);**
- In the above syntax, there is an optional **callback** function which is executed once `setState()` is completed and the component is re-rendered.

Example

```
import React, { Component } from 'react';  
import PropTypes from 'prop-types';  
class App extends React.Component {  
  constructor() {  
    super();  
    this.state = {  
      msg: "Welcome to reactjs"  
    };  
    this.updateSetState = this.updateSetState.bind(this);  
  }  
  updateSetState() {  
    this.setState({ msg:"Its a best ReactJS tutorial" });  
  }  
}
```

```
render() {  
  return (  
    <div>  
      <h1>{this.state.msg}</h1>  
      <button onClick = {this.updateSetState}>SET STATE</button>  
    </div>  
  );  
}  
}  
  
export default App;
```


Main.js

- **import** React from 'react';
- **import** ReactDOM from 'react-dom';
- **import** App from './App.js';

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

forceUpdate()

- This method allows us to update the component manually.
- Syntax

```
Component.forceUpdate(callback);
```

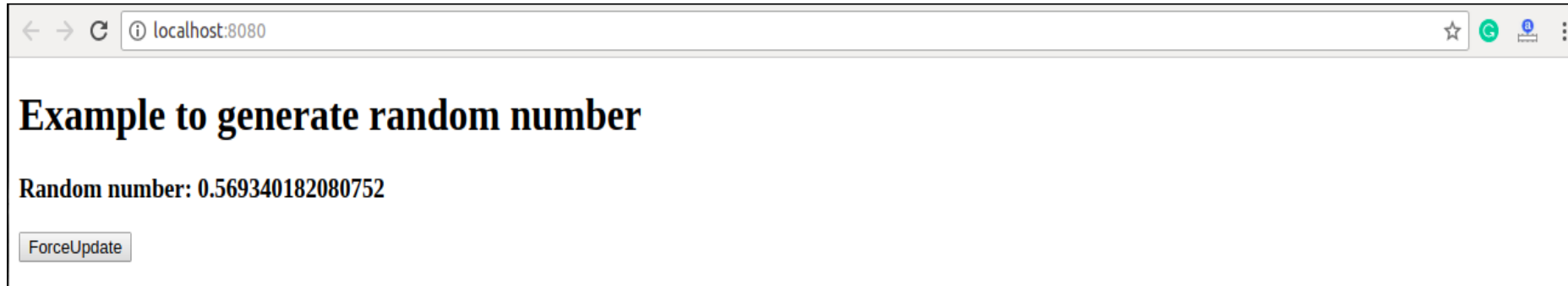
Example

App.js

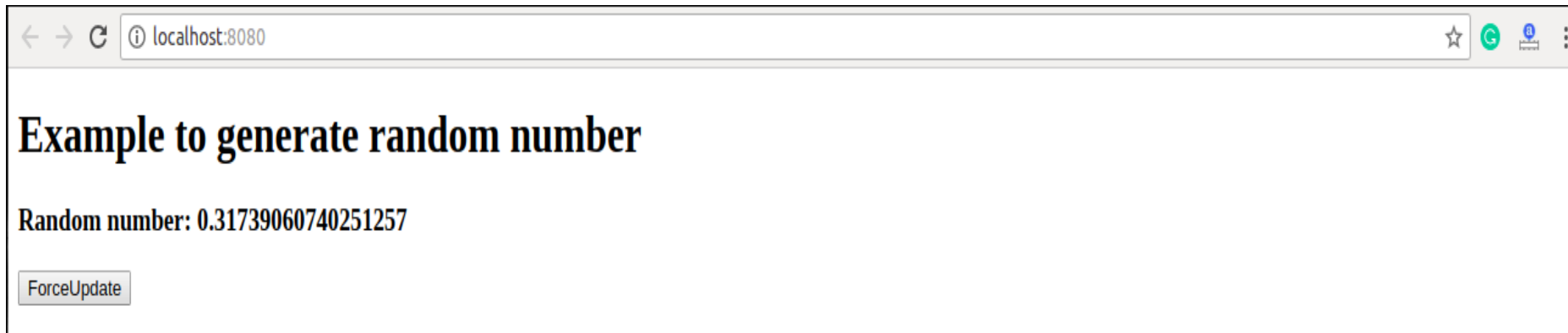
```
import React, { Component } from 'react';  
class App extends React.Component {  
  constructor() {  
    super();  
    this.forceUpdateState = this.forceUpdateState.bind(this);  
  }  
  forceUpdateState() {  
    this.forceUpdate();  
  };  
};
```

```
render() {  
  return (  
    <div>  
      <h1>Example to generate random number</h1>  
      <h3>Random number: {Math.random()}</h3>  
      <button onClick = {this.forceUpdateState}>ForceUpdate</button>  
    </div>  
  );  
}  
}  
  
export default App;
```

Output:



Each time when you click on **ForceUpdate** button, it will generate the **random** number. It can be shown in the below image.



findDOMNode()

- For DOM manipulation, you need to use **ReactDOM.findDOMNode()** method.
- This method allows us to find or access the underlying DOM node.
- Syntax

```
ReactDOM.findDOMNode(component);
```

Example

- For DOM manipulation, first, you need to import this line: **import ReactDOM** from '**react-dom**' in your **App.js** file.

- **App.js**

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

```
import ReactDOM from 'react-dom';
```

```
class App extends React.Component {
```

```
  constructor() {
```

```
    super();
```

```
    this.findDomNodeHandler1 = this.findDomNodeHandler1.bind(this);
```

```
    this.findDomNodeHandler2 = this.findDomNodeHandler2.bind(this);
```

```
  };
```

```
findDOMNodeHandler1() {  
    var myDiv = document.getElementById('myDivOne');  
    ReactDOM.findDOMNode(myDivOne).style.color = 'red';  
}  
  
findDOMNodeHandler2() {  
    var myDiv = document.getElementById('myDivTwo');  
    ReactDOM.findDOMNode(myDivTwo).style.color = 'blue';  
}
```



```
render() {  
  return (  
    <div>  
      <h1>ReactJS Find DOM Node Example</h1>  
      <button onClick = {this.findDomNodeHandler1}>FIND_DOM_NODE1  
</button>  
      <button onClick = {this.findDomNodeHandler2}>FIND_DOM_NODE2  
</button>  
      <h3 id = "myDivOne">JTP-NODE1</h3>  
      <h3 id = "myDivTwo">JTP-NODE2</h3>  
    </div>  
  );  
}  
}  
export default App;
```

Output:



Once you click on the **button**, the color of the node gets changed.

React Component Life-Cycle

- In ReactJS, every component creation process involves various lifecycle methods. These lifecycle methods are termed as component's lifecycle. These lifecycle methods are not very complicated and called at various points during a component's life. The lifecycle of the component is divided into **four phases**.
- They are:
 1. Initial Phase
 2. Mounting Phase
 3. Updating Phase
 4. Unmounting Phase

1. Initial Phase

- It is the **birth** phase of the lifecycle of a ReactJS component. Here, the component starts its journey on a way to the DOM. In this phase, a component contains the default Props and initial State. These default properties are done in the constructor of a component. The initial phase only occurs once and consists of the following methods.
- **getDefaultProps()**
It is used to specify the default value of this.props. It is invoked before the creation of the component or any props from the parent is passed into it.
- **getInitialState()**
It is used to specify the default value of this.state. It is invoked before the creation of the component.

2. Mounting Phase

- In this phase, the instance of a component is created and inserted into the DOM. It consists of the following methods.
- **componentWillMount()**
This is invoked immediately before a component gets rendered into the DOM. In the case, when you call **setState()** inside this method, the component will not **re-render**.
- **componentDidMount()**
This is invoked immediately after a component gets rendered and placed on the DOM. Now, you can do any DOM querying operations.
- **render()**
This method is defined in each and every component. It is responsible for returning a single root **HTML node** element. If you don't want to render anything, you can return a **null** or **false** value.

3. Updating Phase

- It is the next phase of the lifecycle of a react component. Here, we get new **Props** and change **State**.
- This phase also allows to handle user interaction and provide communication with the components hierarchy.
- The main aim of this phase is to ensure that the component is displaying the latest version of itself.
- Unlike the Birth or Death phase, this phase repeats again and again.
- This phase consists of the following methods.

Methods-

- **componentWillReceiveProps()**

It is invoked when a component receives new props. If you want to update the state in response to prop changes, you should compare `this.props` and `nextProps` to perform state transition by using **`this.setState()`** method.

- **shouldComponentUpdate()**

It is invoked when a component decides any changes/updation to the DOM. It allows you to control the component's behavior of updating itself. If this method returns `true`, the component will update. Otherwise, the component will skip the updating.

Methods-

- **componentWillUpdate()**

It is invoked just before the component updating occurs. Here, you can't change the component state by invoking **this.setState()** method. It will not be called, if **shouldComponentUpdate()** returns false.

- **render()**

It is invoked to examine **this.props** and **this.state** and return one of the following types: React elements, Arrays and fragments, Booleans or null, String and Number. If **shouldComponentUpdate()** returns false, the code inside **render()** will be invoked again to ensure that the component displays itself properly.

Methods-

- **componentDidUpdate()**

It is invoked immediately after the component updating occurs. In this method, you can put any code inside this which you want to execute once the updating occurs. This method is not invoked for the initial render.

4. Unmounting Phase

- It is the final phase of the react component lifecycle. It is called when a component instance is **destroyed** and **unmounted** from the DOM. This phase contains only one method and is given below.
- **componentWillUnmount()**
This method is invoked immediately before a component is destroyed and unmounted permanently. It performs any necessary **cleanup** related task such as invalidating timers, event listener, canceling network requests, or cleaning up DOM elements. If a component instance is unmounted, you cannot mount it again.

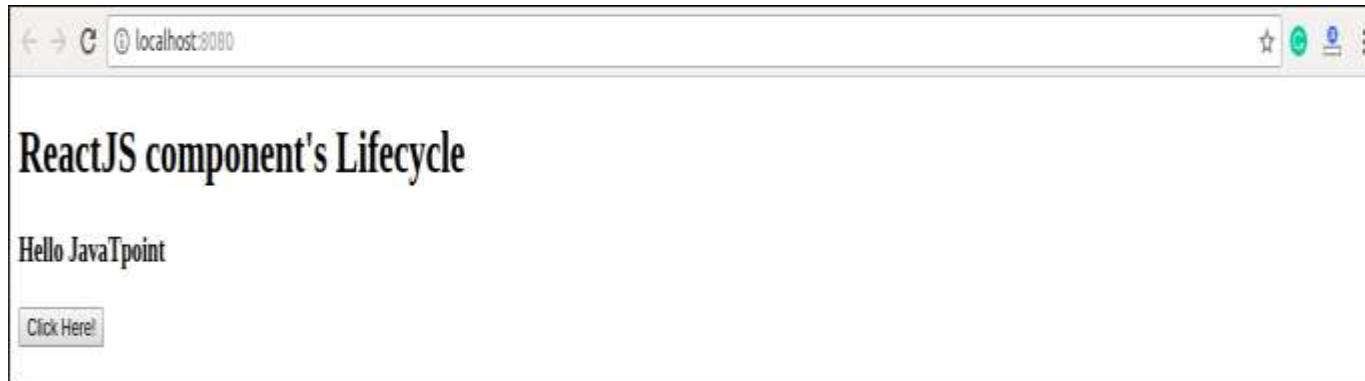
Example

- **import** React, { Component } from 'react';
- **class** App **extends** React.Component {
- constructor(props) {
- **super**(props);
- **this.state** = {hello: "JavaTpoint"};
- **this.changeState** = **this.changeState.bind(this)**
- }
- render() {
- **return** (
- <div>
- <h1>ReactJS component's Lifecycle</h1>
- <h3>Hello {**this.state.hello**}</h3>
- <button onClick = {**this.changeState**}>Click Here!</button>
- </div>
-);
- }

- `componentWillMount() {`
- `console.log('Component Will MOUNT!')`
- `}`
- `componentDidMount() {`
- `console.log('Component Did MOUNT!')`
- `}`
- `changeState(){`
- `this.setState({hello:"All!!- Its a great reactjs tutorial."});`
- `}`
- `componentWillReceiveProps(newProps) {`
- `console.log('Component Will Recieve Props!')`
- `}`

- `shouldComponentUpdate(newProps, newState) {`
- **`return true;`**
- `}`
- `componentWillUpdate(nextProps, nextState) {`
- `console.log('Component Will UPDATE!');`
- `}`
- `componentDidUpdate(prevProps, prevState) {`
- `console.log('Component Did UPDATE!')`
- `}`
- `componentWillUnmount() {`
- `console.log('Component Will UNMOUNT!')`
- `}`
- `}`
- `export default App;`

Output:



When you click on the **Click Here** Button, you get the updated result which is shown in the below screen



React Forms

- Forms are an integral part of any modern web application.
- It allows the users to interact with the application as well as gather information from the users.
- Forms can perform many tasks that depend on the nature of your business requirements and logic such as authentication of the user, adding user, searching, filtering, booking, ordering, etc.
- A form can contain text fields, buttons, checkbox, radio button, etc.

Creating Form

- React offers a stateful, reactive approach to build a form. The component rather than the DOM usually handles the React form. In React, the form is usually implemented by using controlled components.
- There are mainly two types of form input in React.
 1. Uncontrolled component
 2. Controlled component

Uncontrolled component

- The uncontrolled input is similar to the traditional HTML form inputs.
- The DOM itself handles the form data.
- Here, the HTML elements maintain their own state that will be updated when the input value changes.
- To write an uncontrolled component, you need to use a ref to get form values from the DOM.
- In other words, there is no need to write an event handler for every state update.
- You can use a ref to access the input field value of the form from the DOM.

Example

- In this example, the code accepts a field **username** and **company name** in an uncontrolled component.

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

```
class App extends React.Component {
```

```
  constructor(props) {
```

```
    super(props);
```

```
    this.updateSubmit = this.updateSubmit.bind(this);
```

```
    this.input = React.createRef();
```

```
  }
```

```
  updateSubmit(event) {
```

```
    alert('You have entered the UserName and CompanyName successfully.');
```

```
    event.preventDefault();
```

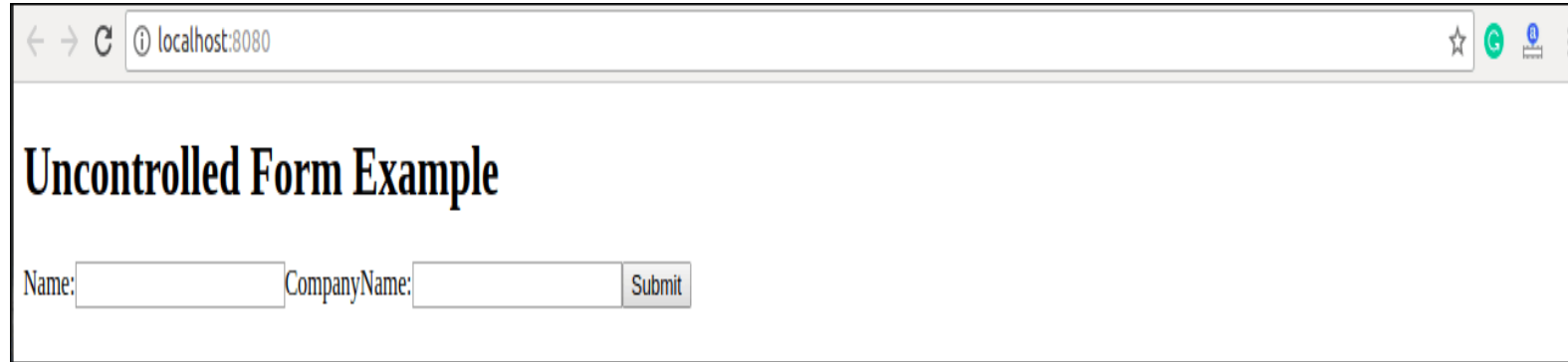
```
  }
```

```
render() {  
  return (  
    <form onSubmit={this.updateSubmit}>  
      <h1>Uncontrolled Form Example</h1>  
      <label>Name:  
        <input type="text" ref={this.input} />  
      </label>  
      <label>  
        CompanyName:  
        <input type="text" ref={this.input} />  
      </label>  
      <input type="submit" value="Submit" />  
    </form>  
  );  
}  
}
```

export default App;

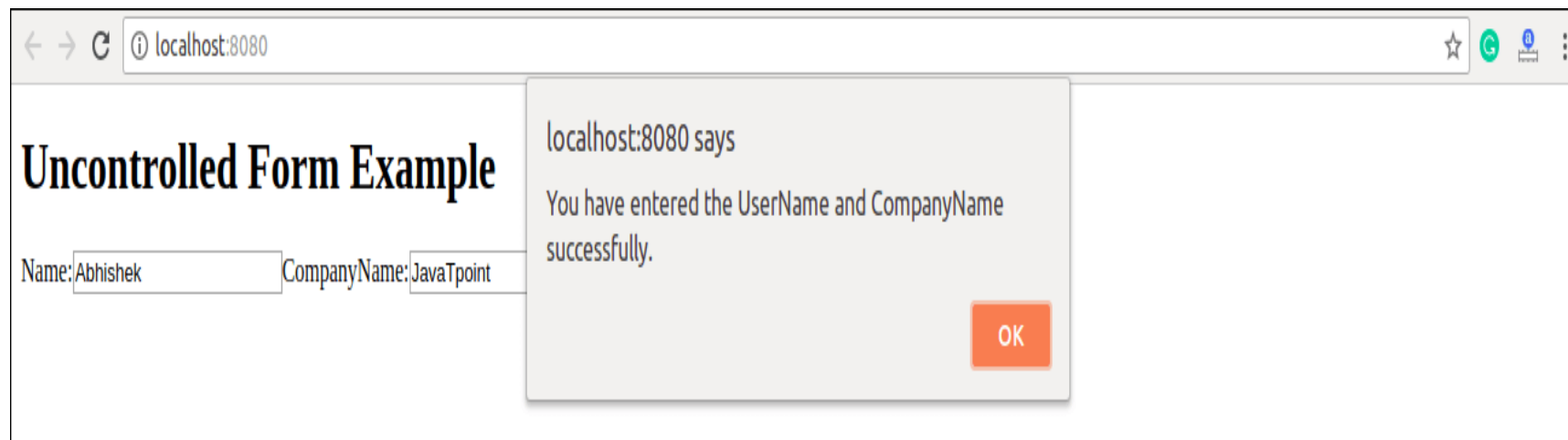
Output

- When you execute the above code, you will see the following screen.



A screenshot of a web browser window with the address bar showing 'localhost:8080'. The page title is 'Uncontrolled Form Example'. Below the title, there is a form with two input fields: 'Name:' followed by an empty text box, and 'CompanyName:' followed by an empty text box. To the right of the 'CompanyName' field is a 'Submit' button.

After filling the data in the field, you get the message that can be seen in the below screen.



A screenshot of a web browser window showing the same 'Uncontrolled Form Example' page. The 'Name' field is now filled with 'Abhishek' and the 'CompanyName' field is filled with 'JavaTpoint'. A modal dialog box is displayed over the form, titled 'localhost:8080 says'. The message inside the dialog reads: 'You have entered the UserName and CompanyName successfully.' There is an 'OK' button at the bottom right of the dialog box.

Controlled Component

- In HTML, form elements typically maintain their own state and update it according to the user input.
- In the controlled component, the input form element is handled by the component rather than the DOM.
- Here, the mutable state is kept in the state property and will be updated only with **setState()** method.
- Controlled components have functions that govern the data passing into them on every **onChange event**, rather than grabbing the data only once, e.g., when you click a **submit button**. This data is then saved to state and updated with `setState()` method. This makes component have better control over the form elements and data.

Controlled Component

- A Controlled component takes its current value through **props** and notifies the changes through **callbacks** like an onChange event.
- A parent component "controls" this changes by handling the callback and managing its own state and then passing the new values as props to the controlled component.
- It is also called as a "dumb component."

Example

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

```
class App extends React.Component {
```

```
  constructor(props) {
```

```
    super(props);
```

```
    this.state = {value: ''};
```

```
    this.handleChange = this.handleChange.bind(this);
```

```
    this.handleSubmit = this.handleSubmit.bind(this);
```

```
  }
```

```
  handleChange(event) {
```

```
    this.setState({value: event.target.value});
```

```
  }
```

```
  handleSubmit(event) {
```

```
    alert('You have submitted the input successfully: ' + this.state.value);
```

```
    event.preventDefault();
```

```
  }
```

```
render() {  
  return (  
    <form onSubmit={this.handleSubmit}>  
      <h1>Controlled Form Example</h1>  
      <label>  
        Name:  
        <input type="text" value={this.state.value} onChange={this.handleChange} />  
  
      </label>  
      <input type="submit" value="Submit" />  
    </form>  
  );  
}  
}  
export default App;
```

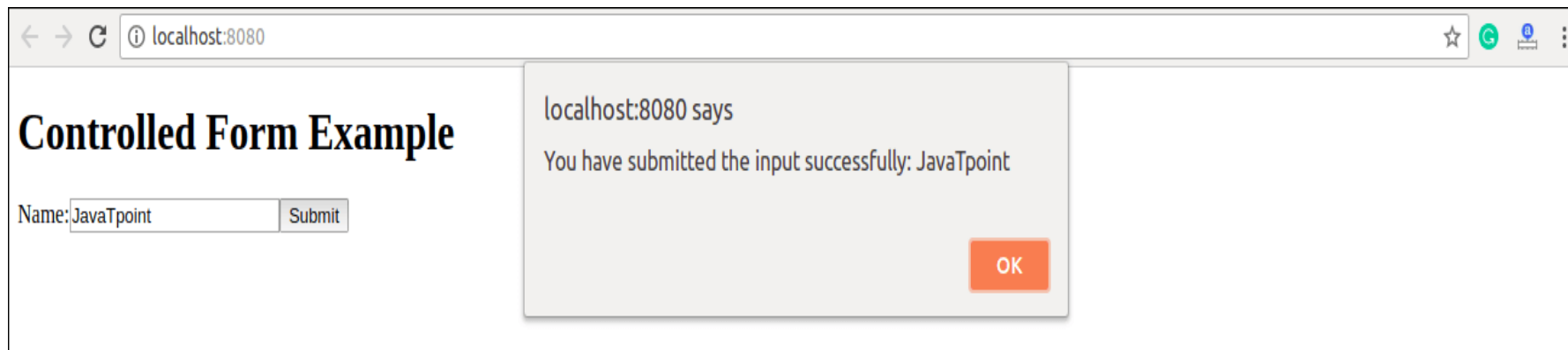

Output

- When you execute the above code, you will see the following screen.



A screenshot of a web browser window. The address bar shows 'localhost:8080'. The page title is 'Controlled Form Example'. Below the title, there is a form with a label 'Name:' followed by an empty text input field and a 'Submit' button.

After filling the data in the field, you get the message that can be seen in the below screen



A screenshot of a web browser window. The address bar shows 'localhost:8080'. The page title is 'Controlled Form Example'. Below the title, there is a form with a label 'Name:' followed by an input field containing the text 'JavaTpoint' and a 'Submit' button. A modal dialog box is displayed over the form, showing the message: 'localhost:8080 says You have submitted the input successfully: JavaTpoint' with an 'OK' button.

Difference between controlled and uncontrolled component

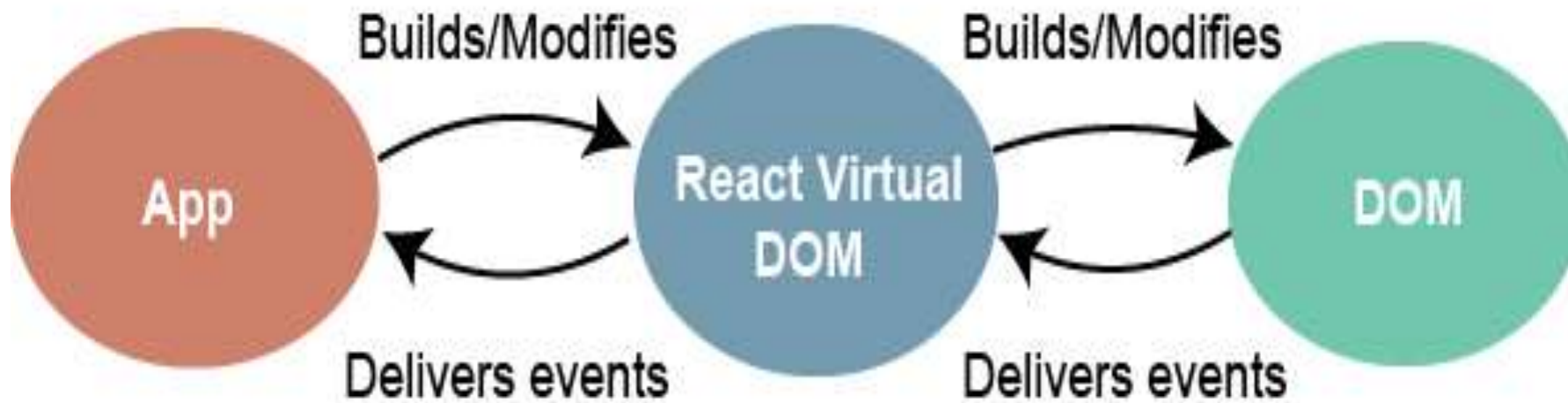
SN	Controlled	Uncontrolled
1.	It does not maintain its internal state.	It maintains its internal states.
2.	Here, data is controlled by the parent component.	Here, data is controlled by the DOM itself.
3.	It accepts its current value as a prop.	It uses a ref for their current values.
4.	It allows validation control.	It does not allow validation control.
5.	It has better control over the form elements and data.	It has limited control over the form elements and data.

React Events

- An event is an action that could be triggered as a result of the user action or system generated event.
- For example, a mouse click, loading of a web page, pressing a key, window resizes, and other interactions are called events.
- React has its own event handling system which is very similar to handling events on DOM elements.
- The react event handling system is known as Synthetic Events.
- The synthetic event is a cross-browser wrapper of the browser's native event.

React Events

Events Handler



React Events

Handling events with react have some syntactic differences from handling events on DOM.

These are:

- 1) React events are named as **camelCase** instead of **lowercase**.
- 2) With JSX, a function is passed as the **event handler** instead of a **string**.
 - For example:
 - **Event declaration in plain HTML:**

```
<button onclick="showMessage()">  
    Hello JavaTpoint  
</button>
```

React Events

- **Event declaration in React:**

```
<button onClick={showMessage}>
```

```
  Hello JavaTpoint
```

```
</button>
```

React Events

3) In react, we cannot return **false** to prevent the **default** behavior. We must call **preventDefault** event explicitly to prevent the default behavior.

For example:

- In plain HTML, to prevent the default link behavior of opening a new page, we can write:

```
<a href="#" onclick="console.log('You had clicked a Link.');" return false">  
  Click_Me  
</a>
```

React Events

- In React, we can write it as:

```
function ActionLink() {  
  function handleClick(e) {  
    e.preventDefault();  
    console.log('You had clicked a Link.');  }  
  return (  
    <a href="#" onClick={handleClick}>  
      Click_Me  
    </a>  
  );  
}
```

In the above example, e is a **Synthetic Event**

Example

- In the below example, we have used only one component and adding an onChange event. This event will trigger the **changeText** function, which returns the company name.

```
import React, { Component } from 'react';  
class App extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {  
      companyName: "  
    };  
  }  
}
```

```
changeText(event) {  
  this.setState({  
    companyName: event.target.value  
  });  
}  
render() {  
  return (  
    <div>  
      <h2>Simple Event Example</h2>  
      <label htmlFor="name">Enter company name: </label>  
      <input type="text" id="companyName" onChange={ this.changeText.bind(this) } />  
      <h4>You entered: { this.state.companyName }</h4>  
    </div>  
  );  
}  
}  
export default App;
```

Output

- When you execute the above code, you will get the following output.



A screenshot of a web browser window. The address bar shows 'localhost:8080'. The page title is 'Simple Event Example'. Below the title, there is a label 'Enter company name:' followed by an empty text input field. Below the input field, there is a label 'You entered:'.

After entering the name in the textbox, you will get the output as like below screen.



A screenshot of a web browser window, similar to the one above. The address bar shows 'localhost:8080'. The page title is 'Simple Event Example'. Below the title, there is a label 'Enter company name:' followed by a text input field containing the text 'www.javatpoint.com'. Below the input field, there is a label 'You entered:' followed by the text 'www.javatpoint.com'.

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.

React Conditional Rendering

- 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**.

React 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

React Conditional Rendering

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


Example continue

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

2) 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.  
}
```

2) Logical && operator

- 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>  
  );  
}
```

3) 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>  
  );  
}
```

4)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;  
  }  
}
```

5)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>  
  );  
}
```

Preventing Component from Rendering

- Sometimes it might happen that a component hides itself even though another component rendered it.
- To do this (prevent a component from rendering), we will have to return **null** instead of its render output.
- It can be understood in the below example:

Example

- In this example, the is rendered based on the value of the prop called **displayMessage**. If the prop value is false, then the component does not render.

```
import React from 'react';
import ReactDOM from 'react-dom';
function Show(props)
{
  if(!props.displayMessage)
    return null;
  else
    return <h3>Component is rendered</h3>;
}
```

```
ReactDOM.render(  
  <div>  
    <h1>Message</h1>  
    <Show displayMessage = {true} />  
  </div>,  
  document.getElementById('app')  
);
```

Output:



React Lists

- Lists are used to display data in an ordered format and mainly used to display menus on websites.
- In React, Lists can be created in a similar way as we create lists in JavaScript. Let us see how we transform Lists in regular JavaScript.
- The `map()` function is used for traversing the lists.
- In the below example, the `map()` function takes an array of numbers and multiply their values with 5. We assign the new array returned by `map()` to the variable `multiplyNums` and log it.

Example

```
var numbers = [1, 2, 3, 4, 5];  
const multiplyNums = numbers.map((number)=>{  
  return (number * 5);  
});  
console.log(multiplyNums);
```

Output

```
[5,10,15,20,25]
```

React Lists

- Now, let us see how we create a list in React.
- To do this, we will use the `map()` function for traversing the list element, and for updates, we enclosed them between **curly braces** `{}`.
- Finally, we assign the array elements to `listItems`.
- Now, include this new list inside ` ` elements and render it to the DOM.

Example

```
import React from 'react';  
import ReactDOM from 'react-dom';  
  
const myList = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];  
const listItems = myList.map((myList)=>{  
  return <li>{myList}</li>;  
});  
ReactDOM.render(  
  <ul> {listItems} </ul>,  
  document.getElementById('app')  
>);  
export default App;
```

Output



Rendering Lists inside components

- In the previous example, we had directly rendered the list to the DOM. But it is not a good practice to render lists in React.
- In React, we had already seen that everything is built as individual components.
- Hence, we would need to render lists inside a component. We can understand it in the following code.

Example

- **import** React from 'react';
- **import** ReactDOM from 'react-dom';
-
- function NameList(props) {
- **const** myLists = props.myLists;
- **const** listItems = myLists.map((myList) =>
- {myList}
-);

```
return (  
  <div>  
    <h2>Rendering Lists inside component</h2>  
    <ul>{listItems}</ul>  
  </div>  
);  
}  
  
const myLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];  
ReactDOM.render(  
  <NameList myLists={myLists} />,  
  document.getElementById('app')  
);  
  
export default App;
```

Output

