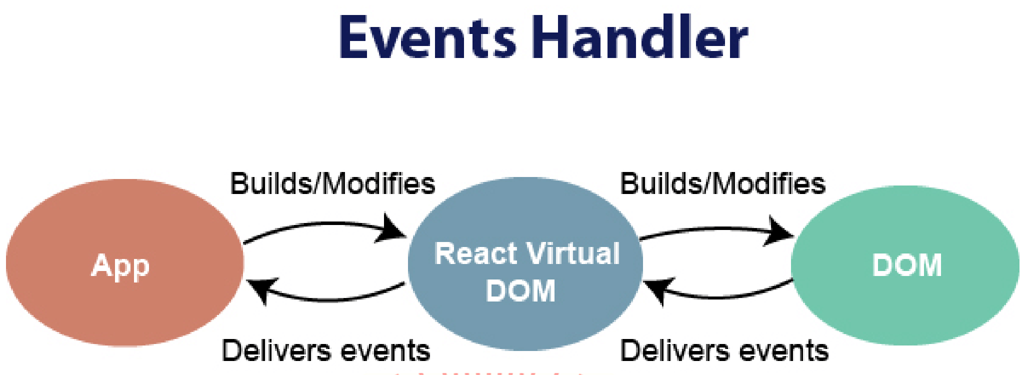
Event Handling: Event Handling and Binding event handlers,
An event is an action that could be triggered as a result of the user action or system generated event. For example, a mouse click, loading of a web page, pressing a key, window resizing, and other interactions are called events.

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.



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

React events are named as camelCase instead of lowercase.
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 Geetanjali
</button>
```

Event declaration in React:

```
<button onClick={showMessage}>
Hello Geetanjali
</button>
```

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>
```

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 which defines according to the W3C spec.

Now let us see how to use Events in React.

Example

In the below example, we have used only one component and added 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;
```

Rendering: Conditional Rendering and List 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 Components 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 is not logged in, render the login component to display the login button. In React, this situation is called 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**

With 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.
}
```

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

```
('geetanjali' == 'Geetanjali') && 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';
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 is 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 Geetanjali':
         return <Message text={text} />;
      default:
         return null;
   }
}
```

Conditional Rendering with enums

An enum is a great way to have multiple conditional rendering. It is more readable as compared to switch case operators. 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;
```

List and keys,

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 multiplies 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);
OP:
[5, 10, 15, 20, 25]
```

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 <ul> </ul> 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('root')
);
```

React Lists

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) =>
      <li>{myList}</li>
   );
   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;
```

Index as Key Anti-pattern

A key is a unique identifier. In React, it is used to identify which items have changed, updated, or deleted from the Lists. It is useful when we dynamically create components or when the users alter the lists. It also helps to determine which components in a collection need to be re-rendered instead of re-rendering the entire set of components every time.

Keys should be given inside the array to give the elements a stable identity. The best way to pick a key as a string that uniquely identifies the items in the list. It can be understood with the below example.

Example

```
const stringLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];
const updatedLists = stringLists.map((strList) => {
   <li key={strList.id}> {strList} </li>;
});
```

If there are no stable IDs for rendered items, you can assign the item index as a key to the lists. It can be shown in the below example.

Example

```
const stringLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];
const updatedLists = stringLists.map((strList, index) => {
   <li key={index}> {strList} </li>;
});
```

Note: It is not recommended to use indexes for keys if the order of the item may change in future. It creates confusion for the developer and may cause issues with the component state.

Using Keys with component

Consider you have created a separate component for ListItem and extracting ListItem from that component. In this case, you should have to assign keys on the <ListItem /> elements in the array, not to the <li> elements in the ListItem itself. To avoid mistakes, you have to keep in mind that keys only make sense in the context of the surrounding array. So, anything you are returning from map() function is recommended to be assigned a key.

Example: Incorrect Key usage

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

function ListItem(props) {
   const item = props.item;
   return (
      // Wrong! No need to specify the key here.
      <li key={item.toString()}>
         {item}
      </li>
   );
}
function NameList(props) {
   const myLists = props.myLists;
   const listItems = myLists.map((strLists) =>
      // The key should have been specified here.
      <ListItem item={strLists} />
   );
   return (
      <div>
         <h2>Incorrect Key Usage Example</h2>
         <ol>{listItems}</ol>
      </div>
   );
}
```

```
const myLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];
ReactDOM.render(
   <NameList myLists={myLists} />,
   document.getElementById('app')
);
export default App;
```

In the given example, the list is rendered successfully. But it is not a good practice that we had not assigned a key to the map() iterator.

React Keys
Example: Correct Key usage
To correct the above example, we should have to assign a key to the map() iterator.

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

function ListItem(props) {
   const item = props.item;
   return (
      // No need to specify the key here.
      <li> {item} </li>
   );
}
function NameList(props) {
   const myLists = props.myLists;
   const listItems = myLists.map((strLists) =>
      // The key should have been specified here.
      <ListItem key={myLists.toString()} item={strLists} />
   );
   return (
      <div>
         <h2>Correct Key Usage Example</h2>
         <ol>{listItems}</ol>
      </div>
   );
}
const myLists = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];
ReactDOM.render(
   <NameList myLists={myLists} />,
   document.getElementById('app')
);
export default App;
```

Uniqueness of Keys among Siblings

We had discussed that keys assignment in arrays must be unique among their siblings. However, it doesn't mean that the keys should be globally unique. We can use the same set of keys in producing two different arrays. It can be understood in the below example.

Example

```
import React from 'react';
import ReactDOM from 'react-dom';
function MenuBlog(props) {
  const titlebar = (
    <ol>
      {props.data.map((show) =>
        <li key={show.id}>
          {show.title}
        </li>
      )}
    </ol>
  );
  const content = props.data.map((show) =>
    <div key={show.id}>
      <h3>{show.title}: {show.content}</h3>
    </div>
  );
  return (
    <div>
      {titlebar}
      <hr />
      {content}
    </div>
  );
}
const data = [
  { id: 1, title: 'First', content: 'Welcome to Geetanjali!!' },
  { id: 2, title: 'Second', content: 'It is the best ReactJS Tutorial!!' },
  { id: 3, title: 'Third', content: 'Here, you can learn all the ReactJS topics!!' }
];
ReactDOM.render(
  <MenuBlog data={data} />,
  document.getElementById('app')
);
export default App;
```

Introduction: Basic form handling
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.

- Uncontrolled component
- 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;
```

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

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 the 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 the setState() method. This makes component have better control over the form elements and data.

A controlled component takes its current value through props and notifies the changes through callbacks like an onChange event. A parent component "controls" this change 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;
```

Handling Multiple Inputs in Controlled Component

If you want to handle multiple controlled input elements, add a name attribute to each element, and then the handler function decided what to do based on the value of event.target.name.

Example

```
import React, { Component } from 'react';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      personGoing: true,
      numberOfPersons: 5
    };
    this.handleInputChange = this.handleInputChange.bind(this);
  }
  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;
    this.setState({
      [name]: value
    });
  }
  render() {
    return (
      <form>
        <h1>Multiple Input Controlled Form Example</h1>
        <label>
          Is Person going:
          <input
```

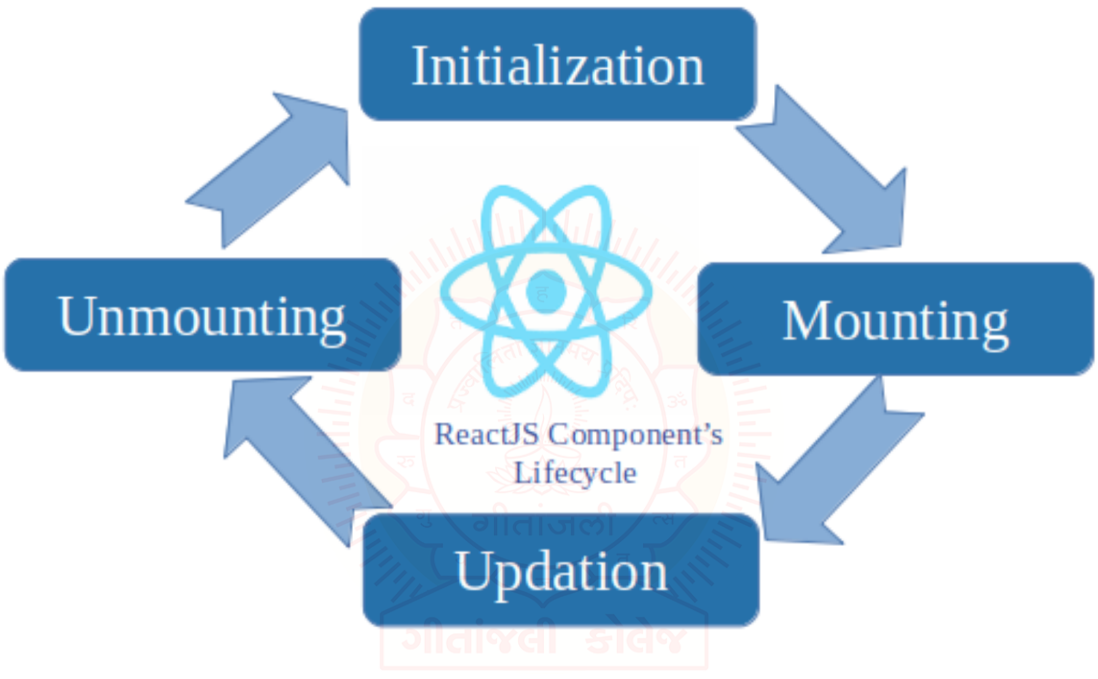
```
              name="personGoing"
              type="checkbox"
              checked={this.state.personGoing}
              onChange={this.handleInputChange} />
          </label>
          <br />
          <label>
            Number of persons:
            <input
              name="numberOfPersons"
              type="number"
              value={this.state.numberOfPersons}
              onChange={this.handleInputChange} />
          </label>
        </form>
      );
    }
}
export default App;
```

Difference table between **controlled** and **uncontrolled component**.

Sr. No	**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.

Components: Components Life Cycle Methods,
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 are called at various points during a component's life. The lifecycle of the component is divided into four phases. They are:

- Initial Phase
- Mounting Phase
- Updating Phase
- 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.

componentWillRecieveProps()
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.

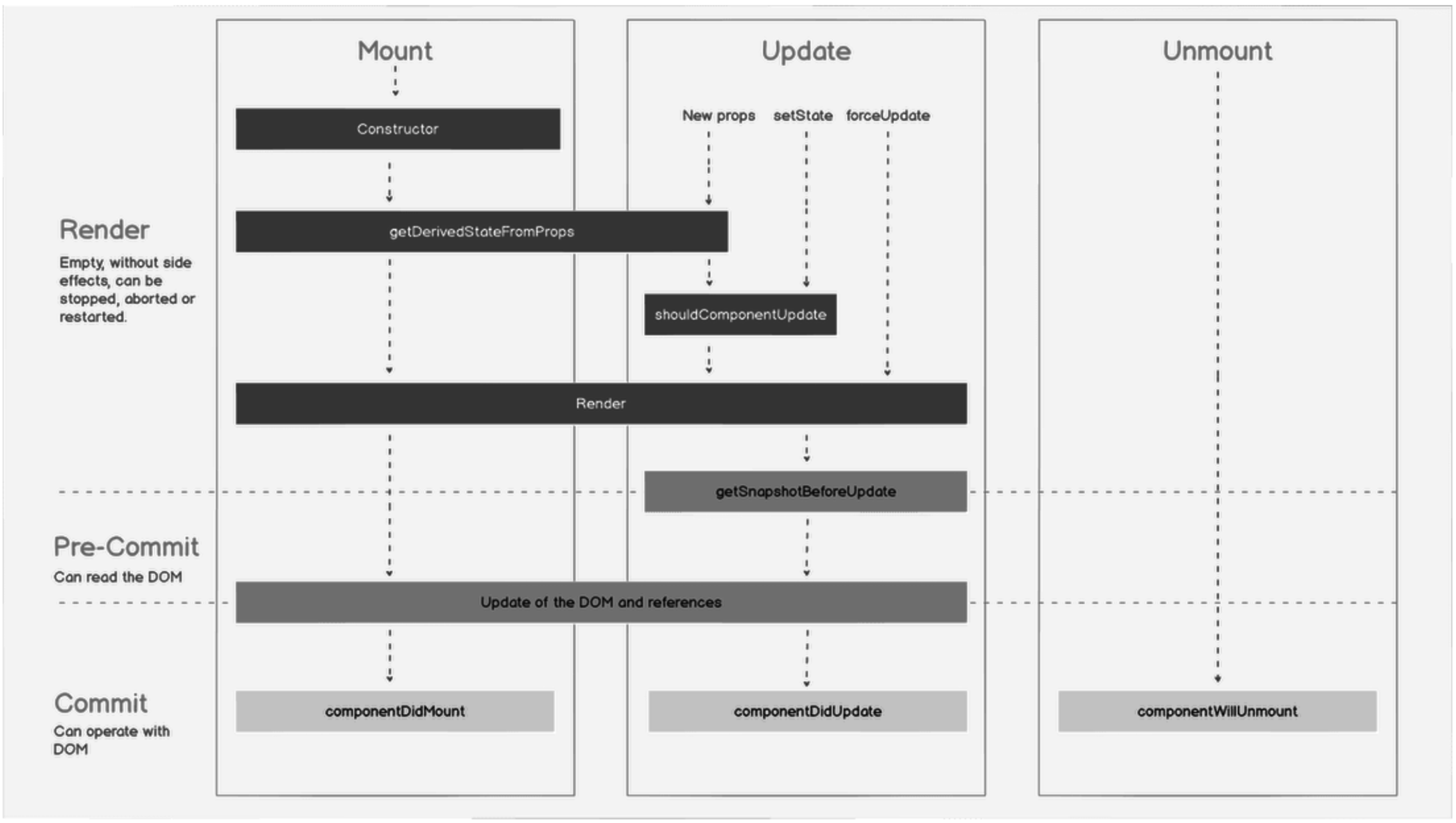
componentWillUpdate()
It is invoked just before the component updating occurs. Here, you can't change the component state by invoking this.setState() method. It will not be called, if shouldComponentUpdate() returns false.

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.

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 listeners, 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: "Geetanjali" };
      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;
```

Pure Components

Generally, In ReactJS, we use shouldComponentUpdate() Lifecycle method to customize the default behavior and implement it when the React component should re-render or update itself.

Prerequisite:

- ReactJS Components
- ReactJS Components – Set 2

Now, ReactJS has provided us with a Pure Component. If we extend a class with Pure Component, there is no need for shouldComponentUpdate() Lifecycle Method. ReactJS Pure Component Class compares current state and props with new props and states to decide whether the React component should re-render itself or Not.

In simple words, If the previous value of state or props and the new value of state or props is the same, the component will not re-render itself. Since Pure Components restricts the re-rendering when there is no use of re-rendering of the component. Pure Components are Class Components which extend React.PureComponent.

Example: Program to demonstrate the creation of Pure Components.

```
import React from 'react';

export default class Test extends React.PureComponent {
   render() {
      return <h1>Welcome to Geetanjali</h1>;
   }
}
```

Extending React Class Components with Pure Components ensures the higher performance of the Component and ultimately makes your application faster, While in the case of Regular Component, it will always re-render either value of State and Props changes or not.

While using Pure Components, Things to be noted are that, In these components, the Value of State and Props are Shallow Compared (Shallow Comparison) and It also takes care of "shouldComponentUpdate" Lifecycle method implicitly.

So there is a possibility that if these State and Props Objects contain nested data structure then Pure Component's implemented shouldComponentUpdate will return false and will not update the whole subtree of Children of this Class Component. So in Pure Component, the nested data structure doesn't work properly.

In this case, State and Props Objects should be simple objects and Child Elements should also be Pure, which means to return the same output for the same input values at any instance.

Fragments
A common pattern in React is for a component to return multiple elements. Fragments let you group a list of children without adding extra nodes to the DOM.

There is also a new short syntax for declaring them.

```
render() {
   return (
      <React.Fragment>
         <ChildA />
         <ChildB />
         <ChildC />
      </React.Fragment>
   );
}
```

Motivation
A common pattern is for a component to return a list of children. Take this example React snippet:

```
class Table extends React.Component {
   render() {
      return (
         <table>
            <tr>
               <Columns />
            </tr>
         </table>
      );
   }
}
```

<Columns /> would need to return multiple <td> elements in order for the rendered HTML to be valid. If a parent div was used inside the render() of <Columns />, then the resulting HTML will be invalid.

```
class Columns extends React.Component {
   render() {
      return (
         <div>
            <td>Hello</td>
            <td>World</td>
         </div>
      );
   }
}
results in a < Table /> output of:
```

```
<table>
  <tr>
    <div>
      <td>Hello</td>
      <td>World</td>
    </div>
  </tr>
</table>
```

Fragments solve this problem.

Usage

```
class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Hello</td>
        <td>World</td>
      </React.Fragment>
    );
  }
}
```

which results in a correct <Table /> output of:

```
<table>
  <tr>
    <td>Hello</td>
    <td>World</td>
  </tr>
</table>
```

Short Syntax

There is a new, shorter syntax you can use for declaring fragments. It looks like empty tags: You can use <></> the same way you'd use any other element except that it doesn't support keys or attributes.

Keyed Fragments

Fragments declared with the explicit <React.Fragment> syntax may have keys. A use case for this is mapping a collection to an array of fragments — for example, to create a description list:

```
class Columns extends React.Component {
  render() {
    return (
      <>
```

```
            <td>Hello</td>
            <td>World</td>
        </>
    );
  }
}
```

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Without the `key`, React will fire a key warning
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
    </dl>
  );
}
```

key is the only attribute that can be passed to Fragment. In the future, we may add support for additional attributes, such as event handlers.