CLASS COMPONENTS

A class component in React is a fundamental building block for creating user interface components. It is defined as a JavaScript class that extends the **React.Component** base class provided by the React library. ***Class components are used to encapsulate the behavior, state, and rendering logic of a component in a more object-oriented manner.***

KEY CHARACTERISTICS OF CLASS COMPONENTS

- **Class Declaration:** Class components are declared using the class keyword and extend the **React.Component** class. This means they inherit certain lifecycle methods and features from React.

- **State Management:** Class components have a built-in state management system using the **this.state** object. State can be initialized and modified using this.setState(), and component re-renders are triggered when the state changes.

- **Lifecycle Methods:** Class components have access to a variety of lifecycle methods, such as **componentDidMount, componentDidUpdate, and componentWillUnmount,** which allow developers to hook into different stages of the component's lifecycle.

- **Render Method:** The render method is required in class components, and it returns the JSX that defines the component's UI.

- **Class Properties:** Class components can define their own instance properties and methods, allowing for the encapsulation of behavior and data within the component class.

HOW TO CREATE CLASS COMPONENT

Creating a class component in React involves defining a JavaScript class that extends the React.Component class and includes methods to manage the component's behavior, state, and rendering.

1. **Import React and React.Component:** In your component file, import the React library and the `Component` class:
   import React, { Component } from 'react';

2. **Create the Class Component** Define a JavaScript class that extends the `Component` class. This class will serve as your React component. Give your component a meaningful name.
   ```
   import React, { Component } from 'react';
     class MyComponent extends Component{
       // Component code goes here
     }
   ```
   Or
   ```
   import Reactfrom 'react';
   class MyComponent extends React.Component{}
   ```

3. **Constructor and State:** Define a constructor method inside your class. This is where you can initialize the component's state using **this.state**. State is used to store and manage data that can change over time.
   e.g.
   ```
   constructor(props) {
      super(props);
   this.state = {  count: 0}; }
   ```

4. **Render Method**: Define a render() method in your class component. This method should return JSX, which describes the component's user interface. The render method is required in all class components
   e.g.
   ```
   render() {
       return (
   <div>
   <p>Count: {this.state.count}</p>
   <button onClick={incrementCount }>Increment</button>
   </div>
       );
     }
   ```

5. **Handling Events:** To handle events, such as button clicks, you can define event handlers as methods in your class.
   ```
   incrementCount() {
   this.setState({ count: this.state.count + 1 });
     }
   ```
   You can then attach this event handler to the button's `onClick` attribute in your JSX.
6. **Export the Component:** At the end of your component file, export your class component so it can be used in other parts of your application:
   e.g.
   ```
   export default MyComponent;
   ```

e.g.-1

```
 import React, {Component} from 'react';
 class Counter extends Component {
 constructor(props) {
 super(props);
this.state = {
    count: 0
  };
 }
incrementCount() {
this.setState(prevState => ({
    count: prevState.count + 1
  }));
 }

render() {
  return (
<div>
<h2>Counter: {this.state.count}</h2>
<button onClick={() =>this.incrementCount()}>Increment</button>
</div>
  );
 }
}
export default Counter;
```

e.g.-2.A

```
import React, { Component } from 'react';
class Toggle extends Component {
  constructor(props) {
    super(props);
this.state = {
isToggleOn: false
    };
  }
handleClick() {
console.log("handleClick start-")
    console.log(this)
console.log("handleClick start-")
this.setState(prevState => ({
isToggleOn: !prevState.isToggleOn
    }));
  }
render() {
    return (
<button onClick={()=>{this.handleClick()}}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
</button>
    );  } }
export default Toggle;
```

e.g.-2.B

```
import React, { Component } from 'react';
class Toggle extends Component {
  constructor(props) {
    super(props);
this.state = {
isToggleOn: false
    };
this.handleClick = this.handleClick.bind(this);
  }

handleClick() {
this.setState(prevState => ({
isToggleOn: !prevState.isToggleOn
    }));
  }

render() {
    return (
<button onClick={this.handleClick}>
       {this.state.isToggleOn ? 'ON' : 'OFF'}
</button>
    );
  }
}
export default Toggle;
```

e.g.-3

```
import React, { Component } from 'react';
class TodoList extends Component {
  constructor(props) {
    super(props);
this.state = {
      todos: [],
newTodo: ""
      };
  }
addTodo() {
    const { todos, newTodo } = this.state;
    if (newTodo !== "") {
this.setState({
        todos: [...todos, newTodo],
newTodo: ''
      });
    }
  }
  handleChange(event) {
this.setState({
newTodo: event.target.value
    });
  }
render() {
    return (
<div>
<input
          type="text"
          value={this.state.newTodo}
onChange={event =>this.handleChange(event)}
        />
<button onClick={() =>this.addTodo()}>Add</button>
<ul>
          {this.state.todos.map((todo, index) => (
<li key={index}>{todo}</li>
          ))}
</ul>
</div>
    );
  }
}
export default TodoList;
```

e.g.-4

```
import React, { Component } from 'react';
class Form extends Component {
  constructor(props) {
    super(props);
    this.state = {
      name: '',
      email: ''
    };
  }
  handleChange(event) {
this.setState({
      [event.target.name]: event.target.value
    });
  }
  handleSubmit(event) {
  event.preventDefault();
    const { name, email } = this.state;
    // Perform form submission or data handling logic
console.log('Name:', name);
console.log('Email:', email);
  }
render() {
    return (
<form onSubmit={event =>this.handleSubmit(event)}>
<label>     Name:<input     type="text"   name="name"   value={this.state.name}   onChange={event
=>this.handleChange(event)} /></label><br />
<label>     Email:<input   type="email"   name="email"     value={this.state.email}   onChange={event
=>this.handleChange(event)} /></label><br />
<button type="submit">Submit</button></form>
    );
  }
}
export default Form;
```

e.g.-5

```
import React, { Component } from 'react';
class Timer extends Component {
  constructor(props) {
    super(props);
this.state = {
      seconds: 0
    };
this.timerInterval = null;
  }

componentDidMount() {
this.timerInterval = setInterval(() => {
this.setState(prevState => ({
        seconds: prevState.seconds + 1
      }));
    }, 1000);
  }
componentWillUnmount() {
clearInterval(this.timerInterval);
  }

render() {
    return (
<div>
<h2>Timer: {this.state.seconds} seconds</h2>
</div>
    ); }
}
export default Timer;
```

LIFECYCLE METHODS IN CLASS COMPONENTS

- COMPONENTDIDMOUNT: THIS METHOD IS CALLED IMMEDIATELY AFTER THE COMPONENT IS RENDERED AND ADDED TO THE DOM. IT'S A GOOD PLACE TO PERFORM INITIAL SETUP. THINK OF IT AS THE MOMENT WHEN YOUR COMPONENT *"COMES TO LIFE."*

e.g.

```
import React, { Component } from'react';
class MyComponent extends Component {
  constructor(props){
    super(props)
    console.log("construction of the component is going on")
  }
  componentDidMount() {
      console.log("component done")
  }
  render() {
    return<div>My Component</div>;
  }
}
```

```
exportdefault MyComponent
```

E.G.1

```
import React, { Component } from'react';
class MyComponent extends Component {
  componentDidMount() {
    console.log('Component has mounted!');
    // Perform any initialization or API calls here
    // Example: Fetch data from an API
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then(response =>response.json())
      .then(data => {
        console.log('Fetched data:', data);
        // Update component state with the fetched data
        this.setState({ data });
      })
      .catch(error => {
        console.log.error('Error fetching data:', error);
      });
  }
  render() {
    return<div>My Component</div>;
  }
}
exportdefault MyComponent
```

E.G.-2

```
import React, { Component } from'react';
class MyComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {
      mydata: ""
    };
  }
  componentDidMount() {
    console.log('Component has mounted!');
    // Perform any initialization or API calls here
    // Example: Fetch data from an API
    fetch('https://jsonplaceholder.typicode.com/posts')
      .then(response =>response.json())
      .then(data => {
        console.log('Fetched data:', data);
        // Update component state with the fetched data
        for (leti = 0; i<data.length; i++) {
          const object = data[i];
          this.setState(prevState => ({
            mydata: prevState.mydata + object.title + <br />
            })
            );
        }
      })
      .catch(error => {
        console.error('Error fetching data:', error);
      });
  }
  render() {
    return (
      <div>
          {this.state.mydata}
          <br/>
      </div>
      )
  }
}
exportdefault MyComponent;
```

- **COMPONENTDIDUPDATE: THIS METHOD IS CALLED WHENEVER THE COMPONENT'S PROPS OR STATE CHANGE. IT'S LIKE A NOTIFICATION THAT SOMETHING HAS BEEN UPDATED. YOU CAN USE IT TO PERFORM ACTIONS BASED ON THE CHANGES, SUCH AS MAKING ADDITIONAL API CALLS OR UPDATING THE COMPONENT'S UI.**

  Syntax

  ```
  componentDidUpdate(prevProps, prevState) {
    // Your code here to handle component updates
    // You can access the previous props and state with prevProps and prevState
  }
  ```
- **prevProps:** This parameter holds the previous props of the component before the update.
- **prevState:** This parameter holds the previous state of the component before the upda

E.G.

```
import React, { Component } from 'react';
class Counter extends Component {
  constructor(props) {
    super(props);
this.state = {
      count: 0
    };
  }

componentDidMount() {
console.log('Component has mounted!');
  }

componentDidUpdate(prevProps, prevState) {
console.log(prevState.count);
        console.log(this.state.count);
  }

incrementCount = () => {
this.setState(prevState => ({
      count: prevState.count + 1
    }));
  };

render() {
    return (
<div>
<p>Count: {this.state.count}</p>
<button onClick={this.incrementCount}>Increment</button>
</div>
    );
  }
}
export default Counter;
```

•**componentWillUnmount:** This method is called right before the component is removed from the DOM. It's your chance to clean up any resources used by the component, such as canceling timers, closing connections, or unsubscribing from event listeners. It's like saying "goodbye" to your component.

e.g.

```
import React from 'react';
class App extends React.Component {
constructor() {
super();
this.state = {
        delete: false,
     };
  }
render() {
     return (
<div>
<h1>User List</h1>
<button onClick={() =>this.setState({ delete: true })}>
             Delete Users
</button>

           {this.state.delete ===false ?<User /> :null }
</div>
     );
  }
}

class User extends React.Component {
componentWillUnmount() {
alert('Deleted User successfully');
  }
render() {
     return (
<div>
<h3>Username: Rahul</h3>
<h3>Email: rbbansal558@gmail.com</h3>
</div>
     );
  }
}
export default App;
```