React Js

Components

• Question 1: What are components in React? Explain the difference between functional components and class components

In React, components are the building blocks of the user interface.

Components can be classified into two main types:

1. **Functional Components**
2. **Class Components**

Functional Components

A **functional component** is a simple JavaScript function that accepts props as an argument and returns JSX. They are easier to write and understand.
Example:

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

Class Components

A **class component** is a more traditional way of defining React components. It is a JavaScript class that extends React.Component and has a render() method to return JSX.
Example:

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

• Question 2: How do you pass data to a component using props?

Rccprops:-

Parent Class:-

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

import Hitesh from './Hitesh'


export default class App extends Component {
  /*
  rcc props-multilevel inheritance// contextapi or redux
  */
  render() {
```

```
    return (
      <div>
        <p>this is a app page.</p>
        <Hitesh fronted="yash"></Hitesh>
      </div>
    )
  }
}
```

Child Class:-

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

export default class Hitesh extends Component {
  render() {
    return (
      <div>
        <h1>{this.props.fronted}</h1>
      </div>
    )
  }
}
```

Rfcprops:-

Parent Class:-

```
import React from 'react'
import Rfcprops from './Rfcprops'
export default function App() {
  const mydata={
    frontend:"react js",
    backend:"nodejs",
    database:"mogodb"
  }
```

```
  //props data pass other compenents- username- home- welcome rfcprops
  return (
    <div>
      <Rfcprops frontend={mydata.frontend} backend={mydata.backend}
database={mydata.database}></Rfcprops>
    </div>
  )
}
```

Child Class:-

```
import React from 'react'


export default function Rfcprops(props) {
  return (
    <div>
    <p>{props.frontend}</p>
    <p>{props.backend}</p>
    <p>{props.database}</p>
    </div>
  )
}
```

• Question 3: What is the role of render() in class components?

Render() use only rcc Components.

The render() method in React class components is responsible for returning the JSX that defines the UI of the component. It determines what will be displayed on the screen.

Key Points:

- **Mandatory**: Every class component must have a render() method.
- **Pure Function**: It should not modify state or props.
- **Return Value**: Must return valid JSX or null.

Example:

```
class MyComponent extends React.Component {
  render() {
```

```
    return <h1>Hello, World!</h1>;
  }
}
```

• Task 1: • Create a functional component Greeting that accepts a name as a prop and displays "Hello, [name]!".

```
import React from "react";
import Greeting from "./Greeting";

function App() {
  return (
    <div>
      <Greeting name="Alice" />
      <Greeting name="Bob" />
    </div>
  );
}
export default App;
import React from "react";
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}
export default Greeting;
```

• Task 2: • Create a class component WelcomeMessage that displays "Welcome to React!" anda render() method.

```
import React from "react";
import WelcomeMessage from "./WelcomeMessage";

function App() {
  return (
    <div>
      <WelcomeMessage />
```

```
    </div>
  );
}
export default App;
```

Props and State

Question 1: What are props in React.js? How are props different from state?

Props data pass one Components anyother Components.

Props data not update.

Props use class and functional base Components.

State data store.

State only use class Components.

State not pass data anyother Components.

Question 2: Explain the concept of state in React and how it is used to manage componentdata.

In React, state is a built-in object used to manage dynamic data that affects how a component renders and behaves. Unlike props, which are read-only and passed from parent to child, state is local to a component and can be updated over time.

Example of State Usage in React:

Functional Component Example (Using useState Hook):

```
import React, { useState } from "react";
function Counter() {
  const [count, setCount] = useState(0); // Declare state
  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

Class Component Example:

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

```
class Counter extends Component {
  constructor() {
    super();
    this.state = { count: 0 }; // Initialize state
  }
  increment = () => {
    this.setState({ count: this.state.count + 1 }); // Update state
  };
  render() {
    return (
      <div>
        <h1>Count: {this.state.count}</h1>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

Question 3: Why is this.setState() used in class components, and how does it work?

In React class components, this.setState() is the method used to update the component's state. It triggers a re-render of the component, ensuring that the UI stays in sync with the updated state.

Why is this.setState() Used in Class Components?

In React class components, this.setState() is the method used to update the component's **state**. It triggers a re-render of the component, ensuring that the UI stays in sync with the updated state.

How this.setState() Works:

1. **Merge and Update State**:
    - this.setState() merges the new state with the existing state.
    - Only the keys specified in the new state object are updated; other state properties remain unchanged.

2. **Asynchronous Behavior**:
    - this.setState() updates the state asynchronously to optimize performance.
    - Multiple state updates may be batched together to reduce unnecessary renders.
3. **Triggers a Re-render**:
    - After the state is updated, React automatically calls the render() method to update the UI based on the new state.

Example:-

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


export default class App extends Component {
    constructor(){
       super()
       this.state={
          frontend:"reactjs"
       }
    }
     mydata= ()=>{
       this.setState({frontend:"node js"})
      }
 // state - object - this -child super - parent  constructor setstate
 render() {
   return (
     <div>
       <p>{this.state.frontend} </p>
       <button onClick={this.mydata}>okk</button>
     </div>
   )
 }
}
```

Task 1: • Create a React component UserCard that accepts name, age, and location asprops and displays them in a card format.

Usercard.js

```
import React from "react";
import "./UserCard.css"; // Optional: For styling the card
function UserCard({ name, age, location }) {
  return (
    <div className="user-card">
      <h2>{name}</h2>
      <p>Age: {age}</p>
      <p>Location: {location}</p>
    </div>
  );
}
export default UserCard;
```

App.js

```
import React from "react";
import UserCard from "./UserCard";
function App() {
  return (
    <div>
      <UserCard name="Alice" age={25} location="New York" />
      <UserCard name="Bob" age={30} location="San Francisco" />
    </div>
  );
}
export default App;
```

Task 2: • Create a Counter component with a button that increments a count value usingReact state. Display the current count on the screen.

Counter.js

```
import React, { useState } from "react";
function Counter() {
  const [count, setCount] = useState(0); // Declare state for count
```

```
  return (
    <div style={{ textAlign: "center", marginTop: "20px" }}>
      <h1>Current Count: {count}</h1>
      <button
        onClick={() => setCount(count + 1)}
        style={{
          padding: "10px 20px",
          fontSize: "16px",
          backgroundColor: "#007BFF",
          color: "white",
          border: "none",
          borderRadius: "5px",
          cursor: "pointer",
        }}
      >
        Increment
      </button>
    </div>
  );
}
export default Counter;
```

App.js

```
import React from "react";
import Counter from "./Counter";
function App() {
  return (
    <div>
      <Counter />
    </div>
  );
```

}

export default App;

Question 1: What is React.js? How is it different from other JavaScript frameworks and libraries?

React.js is a popular, open-source JavaScript library developed by Facebook for building user interfaces (UIs), especially for single-page applications. React is component-based and focuses on creating reusable UI components that manage their own state.

Key Features of React.js:

1. **Component-Based Architecture**:
    - Applications are built using reusable components, making development modular and maintainable.
2. **Virtual DOM**:
    - React uses a Virtual DOM to optimize rendering by efficiently updating only the necessary parts of the real DOM.
3. **Declarative Syntax**:
    - Developers describe what the UI should look like, and React handles updating the DOM to match this description.
4. **Unidirectional Data Flow**:
    - Data flows in one direction, making the application predictable and easier to debug.
5. **JSX (JavaScript XML)**:
    - React combines HTML and JavaScript logic in the same file using JSX for more readable and maintainable code.

Question 2: Explain the core principles of React such as the virtual DOM and componentbased architecture.

React.js is built on several key principles that enable developers to create efficient, reusable, and scalable user interfaces. Here are two of its core principles:

1. Virtual DOM

The Virtual DOM is a lightweight, in-memory representation of the actual DOM. React uses the Virtual DOM to efficiently update and render components.

How It Works:

1. **Virtual Representation**:
    - React creates a Virtual DOM tree that mirrors the structure of the actual DOM.
2. **State/Prop Changes**:

    - When the state or props of a component change, React creates a new Virtual DOM representation.
3. **Diffing Algorithm**:
    - React compares the new Virtual DOM with the previous one to calculate the minimal set of changes required.
4. **Batch Updates**:
    - Only the parts of the actual DOM that need to be updated are changed, improving performance.

Benefits:

- **Performance Optimization**:
    - Avoids direct manipulation of the actual DOM, which is slower.
- **Efficient Rendering**:
    - Reduces the number of DOM updates, ensuring smooth user interactions.

2. Component-Based Architecture

React applications are built using **components**, which are self-contained, reusable building blocks of the UI.

Key Characteristics of Components:

1. **Reusability**:
    - Components can be reused across the application, reducing redundancy.
2. **Encapsulation**:
    - Each component manages its own state and logic independently.
3. **Hierarchical Structure**:
    - Components can be nested, allowing complex UIs to be broken into smaller, manageable parts.

Types of Components:

- **Functional Components**:
    - Simplest form of components, defined as functions.
    - Example:

```
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}
```

Class Components:

- More traditional way of defining components, with lifecycle methods.
- Example:

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

Benefits:

- **Modularity**:
    - Breaking the UI into components makes it easier to develop, test, and maintain.
- **Dynamic Behavior**:
    - Components can manage their own state and respond to user interactions.

Question 3: What are the advantages of using React.js in web development?

Advantages of Using React.js in Web Development

React.js has become a go-to library for web developers due to its efficiency, flexibility, and developer-friendly features. Below are the key advantages:

---

1. Component-Based Architecture

- Reusable Components: React enables developers to build applications with reusable and self-contained components, which reduces code duplication and improves maintainability.
- Encapsulation: Components encapsulate their logic, making them modular and easier to debug.

---

2. Efficient Rendering with Virtual DOM

- Faster Updates: React's Virtual DOM minimizes direct manipulation of the actual DOM, ensuring faster rendering and better performance.
- Optimized Updates: React's diffing algorithm calculates the minimal changes required, reducing unnecessary re-renders.

---

3. Declarative UI

- Readable Code: React's declarative syntax makes it easy to describe how the UI should look at any given time.
- State-Driven Rendering: React automatically updates the UI when the underlying data (state or props) changes.

---

4. Unidirectional Data Flow

- Predictable Behavior: Data flows in one direction (from parent to child), making the application more predictable and easier to debug.
- Better Control: Helps manage the state and props efficiently, even in complex applications.

---

5. Rich Ecosystem and Tooling

- Libraries and Extensions: React integrates well with libraries like Redux, React Router, and Axios for state management, routing, and API calls.
- Developer Tools: Tools like React DevTools enhance debugging and performance profiling.

---

6. Cross-Platform Development

- React Native: React can be used to develop mobile applications with React Native, allowing for code reuse between web and mobile.

---

7. SEO-Friendly

- Server-Side Rendering (SSR): React supports SSR with libraries like Next.js, making applications faster and more SEO-friendly by rendering content on the server.

---

8. Strong Community and Ecosystem

- Large Community: React has a vast community that contributes libraries, tools, and learning resources.
- Enterprise Support: Backed by Facebook and widely adopted in the industry.

---

9. Backward Compatibility

- React maintains backward compatibility, meaning updates to the library don't require major rewrites, reducing maintenance overhead.

---

10. Flexibility

- Integration: React can be integrated into existing projects without requiring a complete rewrite.
- Framework Agnostic: Unlike frameworks like Angular, React focuses solely on the UI, allowing developers to choose their preferred tools for other aspects of the project.

Task: • Set up a new React.js project using create-react-app.

Setting Up a New React.js Project Using create-react-app

Follow these steps to set up a React.js project using the create-react-app tool:

Prerequisites:

1. **Node.js**: Ensure you have Node.js installed. You can download it from Node.js Official Website.
2. **npm or Yarn**: Installed along with Node.js. You can verify by running:

Steps to Create a React Project:

1. Create the React App:

Run the following command in your terminal:

npx create-react-app my-app

- **npx**: Ensures you're using the latest version of create-react-app.
- **my-app**: Name of your project folder. Replace it with your desired name.

2. Navigate to the Project Directory:

After the installation is complete, move into the project folder:

cd my-app

3. Start the Development Server:

Run the following command to start the React application:

npm start

This will open a new browser tab with the default React application running at http://localhost:3000.

Create a basic component that displays "Hello, React!" on the web page.

HelloReact.js

import React from "react";

```
function HelloReact() {
  return <h1>Hello, React!</h1>;
}
export default HelloReact;
```

JSX (JavaScript XML)

Question 1: What is JSX in React.js? Why is it used?

Why is JSX Used?

JSX simplifies the process of creating and managing UI components by combining the power of JavaScript and the structure of HTML.

Key Reasons for Using JSX:

1. Improved Readability:
    - JSX closely resembles HTML, making it more intuitive for developers, especially those familiar with web development.
2. Component Definition:
    - JSX provides a clear way to define React components, including their structure and behavior.

Example:

```
function Greeting() {
  return <h1>Welcome to React!</h1>;
}
```

Dynamic Content:

- JSX allows embedding JavaScript expressions using {} for dynamic content rendering.

Example:

```
const name = "John";
const element = <h1>Hello, {name}!</h1>;
```

 Better Error Messaging:

- JSX provides more descriptive error messages during compilation, making debugging easier.

 Integration with JavaScript:

- JSX seamlessly integrates with JavaScript functions, variables, and objects.

Example:

const user = { name: "Alice", age: 25 };

const element = <h1>{user.name} is {user.age} years old.</h1>;

Tooling and Ecosystem:

- JSX is supported by React's ecosystem, including development tools, IDEs, and linters, ensuring a smooth development experience.

How JSX Works:

JSX code is not directly understood by browsers. It is transpiled by tools like Babel into standard JavaScript using `React.createElement()`.

Question 2: How is JSX different from regular JavaScript? Can you write JavaScript insideJSX?

Key Differences Between JSX and Regular JavaScript:

Aspect	JSX	Regular JavaScript
Syntax	Combines HTML-like tags with JavaScript expressions.	Pure JavaScript syntax, no HTML-like tags.
Output	Transpiled to React.createElement() calls that describe UI components.	Executes directly in the browser or Node.js environment.
Ease of Use	Designed for defining UI components declaratively.	Requires manual DOM manipulation or templating for UI.
HTML Attributes	Uses camelCase for attributes (e.g., className instead of class).	Uses regular attribute names (e.g., class).
Dynamic Content	Embeds JavaScript expressions using {} inside JSX.	Requires string concatenation or template literals for dynamic content.

Yes, you can embed JavaScript expressions inside JSX using curly braces {}. This is one of the most powerful features of JSX, allowing dynamic content and logic to be incorporated directly into your UI.

Question 3: Discuss the importance of using curly braces {} in JSX expressions.

1. Embedding Dynamic Content

Curly braces are used to insert dynamic values or expressions into JSX. This allows developers to render data or perform calculations dynamically. For example:

const name = "John";

return <h1>Hello, {name}!</h1>;

2. Executing JavaScript Code

Inside curly braces, you can include any valid JavaScript expression, such as:

- Arithmetic operations: {2 + 2}
- Function calls: {formatDate(date)}
- Conditional expressions: {isLoggedIn ? 'Welcome back!' : 'Please sign in.'}

3. Dynamic Attribute Values

Curly braces can also be used to assign dynamic values to HTML attributes in JSX:

```
const isActive = true;

return <button disabled={!isActive}>Click Me</button>;
```

4. Working with Objects and Arrays

Curly braces enable rendering arrays, mapping over data, or accessing object properties dynamically:

```
const items = ['Apple', 'Banana', 'Cherry'];

return (
  <ul>
    {items.map(item => <li key={item}>{item}</li>)}
  </ul>
);
```

5. Seamless Integration of JavaScript and Markup

JSX is designed to resemble HTML while allowing JavaScript to interweave seamlessly. Curly braces bridge this gap, enabling a smooth and intuitive way to include logic and data manipulation directly in the markup.

Task: • Create a React component that renders the following JSX elements: • A heading with the text "Welcome to JSX". • A paragraph explaining JSX with dynamic data (use curly braces to insert variables).

```
import React from 'react';

const JSXIntroduction = () => {
  const dynamicText = "a syntax extension for JavaScript that allows you to write HTML-like code within JavaScript";
  return (
    <div>
      <h1>Welcome to JSX</h1>
      <p>JSX is {dynamicText}. It makes your code more readable and allows you to include JavaScript expressions seamlessly.</p>
```

```
    </div>
  );
};
export default JSXIntroduction;
```

Handling Events in React

Question 1: How are events handled in React compared to vanilla JavaScript? Explain the concept of synthetic events.

Aspect	Vanilla JavaScript	React (Synthetic Events)
Event Syntax	onclick	onClick (camelCase)
Event Binding	Inline or via addEventListener	Directly in JSX
Event Delegation	Manually implemented	Built-in with React
Cross-browser Handling	Manual	Handled by synthetic events
Event Object	Native DOM event	SyntheticEvent with native access

Question 2: What are some common event handlers in React.js? Provide examples of onClick, onChange, and onSubmit.

1. onClick

The onClick handler is triggered when an element, like a button, is clicked.

```
import React from "react";

function App() {
  const handleClick = () => {
    alert("Button clicked!");
  };

  return (
    <button onClick={handleClick}>
      Click Me
    </button>
  );
}
```

```
export default App;
```

2. onChange

The onChange handler is triggered when the value of an input field changes.

```
import React, { useState } from "react";

function App() {
  const [text, setText] = useState("");

  const handleChange = (event) => {
    setText(event.target.value);
  };

  return (
    <div>
      <input
        type="text"
        value={text}
        onChange={handleChange}
        placeholder="Type something"
      />
      <p>You typed: {text}</p>
    </div>
  );
}

export default App;
```

3. onSubmit

The onSubmit handler is triggered when a form is submitted.

```
import React, { useState } from "react";
```

```
function App() {
  const [inputValue, setInputValue] = useState("");

  const handleSubmit = (event) => {
    event.preventDefault(); // Prevents the default form submission behavior
    alert(`Form submitted with value: ${inputValue}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
        placeholder="Enter text"
      />
      <button type="submit">Submit</button>
    </form>
  );
}

export default App;
```

Question 3: Why do you need to bind event handlers in class components?

1. this Context in JavaScript

In JavaScript, the value of this depends on how a function is called. For example:

- When a function is called as a method of an object, this refers to that object.
- When a function is called independently (e.g., as a callback), this can become undefined (in strict mode) or refer to the global object.

2. React's Event Handlers

When you pass a method like this.handleClick as an event handler, it is called without being explicitly bound to the component. This results in this being undefined within the handler unless you explicitly bind it.

3. How to Bind Event Handlers

To ensure this refers to the component instance, you need to bind the event handler explicitly. There are several ways to do this:

Task 1: • Create a button in a React component that, when clicked, changes the text from"Not Clicked" to "Clicked!" using event handling.

```
import React, { useState } from "react";

function ClickButton() {
  const [text, setText] = useState("Not Clicked");
  const handleClick = () => {
    setText("Clicked!");
  };
  return (
    <div>
      <p>{text}</p>
      <button onClick={handleClick}>Click Me</button>
    </div>
  );
}

export default ClickButton;
```

Task 2: • Create a form with an input field in React. Display the value of the input field dynamically as the user types in it.

```
import React, { useState } from "react";

function DynamicInputForm() {
  const [inputValue, setInputValue] = useState("");
  const handleChange = (event) => {
    setInputValue(event.target.value); // Update the state with the input field's value
  };
  return (
```

```
    <div>
      <form>
        <label>
          Enter Text:
          <input
            type="text"
            value={inputValue}
            onChange={handleChange} // Call handleChange on every input event
          />
        </label>
      </form>
      <p>Current Value: {inputValue}</p> {/* Dynamically display input value */}
    </div>
  );
}
export default DynamicInputForm;
```

Conditional Rendering

Question 1: What is conditional rendering in React? How can you conditionally render elements in a React component?

Conditional rendering in React is the ability to render different content or components based on a condition. It works similarly to conditional statements in JavaScript (like if, else, or the ternary operator), allowing React to decide which UI elements to display based on the application state or props.

Using Ternary Operator

The ternary operator is a concise way to render one of two possible outcomes.

```
function Greeting({ isLoggedIn }) {
  return (
    <h1>
      {isLoggedIn ? "Welcome back!" : "Please log in."}
    </h1>
  );
}
```

export default Greeting;

Question 2: Explain how if-else, ternary operators, and && (logical AND) are used in JSXfor conditional rendering.

1. Using if-else Statements

Description:

- Use if-else for more complex conditions where multiple outcomes are possible.
- Since JSX expressions cannot directly use statements like if, the logic is typically handled outside the JSX or within helper functions.

Example:

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
  } else {
    return <h1>Please log in.</h1>;
  }
}
```

export default Greeting;

2. Using Ternary Operators

Description:

- A ternary operator is a concise way to render one of two possible outcomes.
- Syntax: condition ? expressionIfTrue : expressionIfFalse

Example:

```
function Greeting({ isLoggedIn }) {
  return (
    <h1>
      {isLoggedIn ? "Welcome back!" : "Please log in."}
    </h1>
  );
}
```

export default Greeting;

3. Using Logical && (Short-Circuit Evaluation)

Description:

- The logical && operator evaluates the right-hand side expression only if the left-hand side condition is true.
- Use it to render content only when a condition is true.

Example:

```
function Notification({ hasNotifications }) {
  return (
    <div>
      <h1>Dashboard</h1>
      {hasNotifications && <p>You have new notifications!</p>}
    </div>
  );
}


export default Notification;
```

• Task 1: • Create a component that conditionally displays a login or logout button based onthe user's login status.

```
import React, { useState } from "react";


function LoginControl() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);


  const handleLogin = () => setIsLoggedIn(true);
  const handleLogout = () => setIsLoggedIn(false);


  return (
    <div>
      {isLoggedIn ? (
```

```
      <button onClick={handleLogout}>Logout</button>
    ) : (
      <button onClick={handleLogin}>Login</button>
    )}
  </div>
 );
}

export default LoginControl;
```

Task 2: • Implement a component that displays a message like "You are eligible to vote" if theuser is over 18, otherwise display "You are not eligible to vote."

```
import React, { useState } from "react";

function VotingEligibility() {
 const [age, setAge] = useState(0); // Default age set to 0

 const handleChange = (event) => {
  setAge(Number(event.target.value)); // Update age state based on user input
 };

 return (
  <div>
   <label>
    Enter your age:{" "}
    <input
     type="number"
     value={age}
     onChange={handleChange} // Handle input changes
    />
   </label>
   <p>
```

```
      {age >= 18
        ? "You are eligible to vote."
        : "You are not eligible to vote."}
    </p>
  </div>
 );
}

export default VotingEligibility;
```

Lists and Keys

• Question 1: How do you render a list of items in React? Why is it important to use keyswhen rendering lists?

Rendering a list in React typically involves the map() function, which iterates over an array and returns a JSX element for each item.

```
function ItemList() {
 const items = ["Apple", "Banana", "Cherry"];

 return (
  <ul>
    {items.map((item, index) => (
      <li key={index}>{item}</li> // Render each item as a list element
    ))}
  </ul>
 );
}

export default ItemList;
```

Why Is It Important to Use Keys When Rendering Lists?

1. Efficient Updates

- React uses the **Virtual DOM** to efficiently update the UI.

- Keys help React identify which items have changed, been added, or removed, allowing it to optimize rendering.

2. Avoiding Re-rendering of Unchanged Elements

- Without keys, React might re-render entire lists unnecessarily, even if only one item changes.
- With keys, React can pinpoint the exact element that needs updating.

3. Correct Component State Preservation

- If a list item contains a stateful component (e.g., an input field), using keys ensures that the component's state is preserved across updates.

• Question 2: What are keys in React, and what happens if you do not provide a unique key?

In React, keys are special attributes used to uniquely identify elements in a list. They help React efficiently update and manage the DOM when changes occur in the list, such as adding, removing, or reordering items.

- Key is a prop that needs to be assigned to each element in a list when rendering multiple components or elements.
- Keys help React track each element, so it can re-render only the parts of the UI that have changed instead of re-rendering the entire list.

  If you do not provide a unique key in a list, React will use the index of the item as a fallback key. While this can work in some cases, it can lead to several issues:

  1. Inefficient Re-renders

  Without unique keys, React might not be able to efficiently identify which items in the list have changed. This can cause unnecessary re-renders of items that haven't changed, reducing performance.

- **Example Issue:**
  - If you remove or add items to the list, React might re-render the entire list, not just the changed items.

  2. Loss of Component State

  When keys are not unique, React may mix up components in the list, causing unexpected behavior or the loss of component state. This happens because React could mistakenly treat different components as the same, leading to mismatched states (e.g., form input values or animations).

- **Example Issue:**
  - If you have a list of components with form inputs and you remove or reorder items, the input fields may "jump" between items or retain incorrect values.

3. Incorrect or Unstable Rendering

- If the key value is not unique (such as using an index in dynamic lists where items are reordered), React might mistakenly reuse an old component for a new one, resulting in bugs or incorrect rendering.

Task 1: • Create a React component that renders a list of items (e.g., a list of fruit names). Usethe map() function to render each item in the list.

```
import React from "react";


function FruitList() {
  const fruits = ["Apple", "Banana", "Cherry", "Date", "Elderberry"];


  return (
    <ul>
      {fruits.map((fruit, index) => (
        <li key={index}>{fruit}</li> // Rendering each fruit as a list item with a key
      ))}
    </ul>
  );
}


export default FruitList;
```

Task 2: • Create a list of users where each user has a unique id. Render the user list usingReact and assign a unique key to each user

```
import React from "react";


function UserList() {
  // Array of user objects with unique id for each user
  const users = [
    { id: 1, name: "John Doe" },
    { id: 2, name: "Jane Smith" },
    { id: 3, name: "Samuel Green" },
    { id: 4, name: "Anna Brown" },
```

```
    { id: 5, name: "Emily White" }
  ];

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li> // Using user.id as the unique key for each user
      ))}
    </ul>
  );
}

export default UserList;
```

Forms in React

Question 1: How do you handle forms in React? Explain the concept of controlled components.

Controlled Components in React

A controlled component is a form element (like <input>, <textarea>, <select>) where the value of the element is controlled by React's state. In other words, React manages the form data, and the value of the form field is always stored in the component's state.

Key Features of Controlled Components:

1. State Management: The form field value is stored in the React component's state.
2. Two-Way Data Binding: The form element's value is updated through React state, and changes to the form field trigger state updates.
3. Event Handlers: You define an event handler (like onChange) to update the state when the user interacts with the form field.

How Controlled Components Work

1. State Initialization: The form field value is initialized in the component's state.
2. Handling Input Changes: When a user types in the form field, the onChange event is triggered, and the state is updated accordingly.

3. Form Submission: When the form is submitted, the state values are accessed to send or process the data.

Question 2: What is the difference between controlled and uncontrolled components in React?

Key Differences Between Controlled and Uncontrolled Components

Aspect	Controlled Components	Uncontrolled Components
Data Source	Data is stored in React state.	Data is stored in the DOM (via ref).
State Management	Form values are managed via React state.	Form values are not bound to React state.
Form Element Access	Value is updated via event handlers (e.g., onChange).	Value is accessed using ref to the DOM.
Rendering	React re-renders the component when the state changes.	No re-rendering, as the DOM handles the form data.
Complexity	More complex, but offers better control over form data.	Simpler for basic forms, but offers less control.
Two-Way Data Binding	Yes, with React managing both read and write.	No, one-way data flow (DOM handles value).

Task 1: • Create a form with inputs for name, email, and password. Use state to control the form and display the form data when the user submits it.

```
import React, { useState } from "react";

function RegistrationForm() {
  // State to hold form data
  const [formData, setFormData] = useState({
    name: "",
    email: "",
    password: ""
  });
  // Handler to update form data state
  const handleChange = (event) => {
```

```
    const { name, value } = event.target;
    setFormData((prevState) => ({
      ...prevState,
      [name]: value, // Dynamically update the state based on input field name
    }));
  };
  // Handler to display form data on form submission
  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`Name: ${formData.name}\nEmail: ${formData.email}\nPassword:
${formData.password}`);
  };
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name:
        <input
          type="text"
          name="name"
          value={formData.name} // Controlled input, linked to state
          onChange={handleChange} // Update state on input change
        />
      </label>
      <br />
      <label>
        Email:
        <input
          type="email"
          name="email"
          value={formData.email} // Controlled input, linked to state
          onChange={handleChange} // Update state on input change
```

```
        />
      </label>
      <br />
      <label>
        Password:
        <input
          type="password"
          name="password"
          value={formData.password} // Controlled input, linked to state
          onChange={handleChange} // Update state on input change
        />
      </label>
      <br />
      <button type="submit">Submit</button>
    </form>
  );
}
export default RegistrationForm;
```

Task 2: • Add validation to the form created above. For example, ensure that the email input contains a valid email address.

```
import React, { useState } from "react";

function RegistrationForm() {
  // State to hold form data and error messages
  const [formData, setFormData] = useState({
    name: "",
    email: "",
    password: "",
  });

  const [errors, setErrors] = useState({
```

```
  name: "",
  email: "",
  password: "",
});

// Validate the email format using a regular expression
const validateEmail = (email) => {
  const emailPattern = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;
  return emailPattern.test(email);
};

// Handler to update form data state
const handleChange = (event) => {
  const { name, value } = event.target;
  setFormData((prevState) => ({
    ...prevState,
    [name]: value, // Dynamically update the state based on input field name
  }));
};

// Handler for form submission
const handleSubmit = (event) => {
  event.preventDefault();

  // Validate inputs
  let isValid = true;
  let validationErrors = {};

  // Name validation
  if (formData.name === "") {
```

```
    validationErrors.name = "Name is required.";
    isValid = false;
   }

   // Email validation
   if (formData.email === "") {
    validationErrors.email = "Email is required.";
    isValid = false;
   } else if (!validateEmail(formData.email)) {
    validationErrors.email = "Please enter a valid email address.";
    isValid = false;
   }

   // Password validation
   if (formData.password === "") {
    validationErrors.password = "Password is required.";
    isValid = false;
   }

   if (isValid) {
    // Form is valid, show an alert with the form data
    alert(`Name: ${formData.name}\nEmail: ${formData.email}\nPassword:
${formData.password}`);
   } else {
    // Set validation errors if any
    setErrors(validationErrors);
   }
  };

  return (
   <form onSubmit={handleSubmit}>
```

```
      <label>
        Name:
        <input
          type="text"
          name="name"
          value={formData.name} // Controlled input, linked to state
          onChange={handleChange} // Update state on input change
        />
      </label>
      {errors.name && <p style={{ color: "red" }}>{errors.name}</p>} {/* Display error if name is invalid */}
      <br />

      <label>
        Email:
        <input
          type="email"
          name="email"
          value={formData.email} // Controlled input, linked to state
          onChange={handleChange} // Update state on input change
        />
      </label>
      {errors.email && <p style={{ color: "red" }}>{errors.email}</p>} {/* Display error if email is invalid */}
      <br />

      <label>
        Password:
        <input
          type="password"
          name="password"
```

```
          value={formData.password} // Controlled input, linked to state
          onChange={handleChange} // Update state on input change
        />
      </label>
      {errors.password && <p style={{ color: "red" }}>{errors.password}</p>} {/* Display error if password is invalid */}
      <br />

      <button type="submit">Submit</button>
    </form>
  );
}

export default RegistrationForm;
```

Lifecycle Methods (Class Components)

Question 1: What are lifecycle methods in React class components? Describe the phases of a component's lifecycle.

Lifecycle Methods in React Class Components

In React, lifecycle methods are special methods that get called at different stages of a component's existence, such as when it is created, updated, or destroyed. These methods allow developers to run specific code at various points in the component's life.

There are three main phases in a component's lifecycle:

1. Mounting: The phase when the component is being created and inserted into the DOM.
2. Updating: The phase when a component is re-rendering as a result of changes to either its props or state.
3. Unmounting: The phase when the component is being removed from the DOM.

Each phase has a set of lifecycle methods that can be called automatically by React at specific points.

---

1. Mounting Phase

The mounting phase occurs when the component is first created and inserted into the DOM. The lifecycle methods associated with this phase are:

- constructor(props):
    - Called when the component is created.
    - Used to initialize state and bind event handlers.

static getDerivedStateFromProps(props, state):

- Called before every render, both during the initial mount and during updates.
- Allows state to be updated in response to prop changes.
- Returns an object to update state or null to indicate no state change.

render():

- This method is required in all class components and returns the JSX that represents the component's UI.
- It is called every time the component is about to render.

componentDidMount():

- Called immediately after the component is inserted into the DOM.
- Ideal for making API calls or setting up subscriptions.

2. Updating Phase

The **updating phase** occurs when a component’s state or props change. React automatically re-renders the component when this happens. The lifecycle methods associated with this phase are:

- **static getDerivedStateFromProps(props, state)**:
    - Called in the updating phase, just like in the mounting phase, whenever the component’s props change.
- **shouldComponentUpdate(nextProps, nextState)**:
    - Determines if a component should re-render when new props or state are received.
    - Returns true (re-render) or false (skip re-render).
    - Used for performance optimization.

 render():

- Called again during updates to reflect changes in props or state.
- Same as in the mounting phase, it returns the component's JSX.

 getSnapshotBeforeUpdate(prevProps, prevState):

- Called right before the changes from the render are applied to the DOM.

- It allows you to capture information (e.g., scroll position) from the DOM before it is changed.
- Returns a value that will be passed to componentDidUpdate().

  - **componentDidUpdate(prevProps, prevState, snapshot)**:
- Called immediately after the component's updates are flushed to the DOM.
- Ideal for handling side effects (e.g., fetching new data when props change).
- The snapshot argument is the value returned by getSnapshotBeforeUpdate().

3. Unmounting Phase

The **unmounting phase** occurs when a component is being removed from the DOM. The lifecycle method associated with this phase is:

- **componentWillUnmount()**:
  - Called immediately before the component is removed from the DOM.
  - Used for cleanup tasks such as invalidating timers or canceling network requests.

Question 2: Explain the purpose of componentDidMount(), componentDidUpdate(),and componentWillUnmount().

1. componentDidMount()

Purpose:

componentDidMount() is called once immediately after the component has been rendered to the screen. This is the ideal place to initiate any side effects such as network requests (API calls), setting up subscriptions, or triggering animations that need to be executed once the component is added to the DOM.

Common Use Cases:

- Fetching data from an API when the component is first rendered.
- Setting up event listeners or subscriptions (e.g., WebSocket connections, custom events).
- Triggering animations or transitions after the component is mounted.

2. componentDidUpdate(prevProps, prevState, snapshot)

Purpose:

componentDidUpdate() is called **after every update** to the component. It runs after the render method is called, but **only if the component's state or props have changed**. This method is useful for performing side effects based on changes in the component, such as responding to prop or state changes, making additional API calls, or performing other updates in the DOM.

Parameters:

- prevProps: The previous props before the update.
- prevState: The previous state before the update.
- snapshot: The value returned by getSnapshotBeforeUpdate() (if used).

Common Use Cases:

- Responding to prop or state changes (e.g., fetching new data when a prop changes).
- Performing DOM updates or calculations based on updated state or props.
- Triggering animations or transitions after a state change.

3. componentWillUnmount()

Purpose:

componentWillUnmount() is called **immediately before the component is removed from the DOM**. This method is used for cleanup tasks, such as invalidating timers, canceling network requests, or removing event listeners or subscriptions to avoid memory leaks when the component is no longer needed.

Common Use Cases:

- Clearing timers or intervals that were set during the component's lifecycle (e.g., setInterval()).
- Removing event listeners (e.g., window.addEventListener()).
- Cancelling network requests or cleaning up subscriptions (e.g., WebSocket connections).

Task 1: • Create a class component that fetches data from an API when the component mounts using componentDidMount(). Display the data in the component.

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


class UserList extends Component {
 constructor(props) {
  super(props);
  // Initialize state to store the fetched data and loading/error state
  this.state = {
   users: [],       // To store the list of users
   isLoading: true, // To indicate if data is still being fetched
   error: null,     // To store any error that occurs during the fetch
  };
```

```
}

// Fetch data when the component mounts
componentDidMount() {
  // Fetch data from the API
  fetch("https://jsonplaceholder.typicode.com/users")
    .then((response) => response.json()) // Convert the response to JSON
    .then((data) => {
      // Update the state with the fetched data
      this.setState({ users: data, isLoading: false });
    })
    .catch((error) => {
      // If an error occurs during fetch, update the error state
      this.setState({ error: error.message, isLoading: false });
    });
}

render() {
  const { users, isLoading, error } = this.state;

  // Display loading message while fetching data
  if (isLoading) {
    return <div>Loading...</div>;
  }

  // Display error message if there's an error
  if (error) {
    return <div>Error: {error}</div>;
  }
```

```
    // Display the list of users
    return (
      <div>
        <h1>User List</h1>
        <ul>
          {users.map((user) => (
            <li key={user.id}>
              {user.name} - {user.email}
            </li>
          ))}
        </ul>
      </div>
    );
  }
}


export default UserList;
```

Task 2: • Implement a component that logs a message to the console when it updates using componentDidUpdate(). Log another message when the component unmounts using componentWillUnmount().

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


class UpdateAndUnmountLogger extends Component {
  constructor(props) {
    super(props);
    // Initial state with a simple counter
    this.state = {
      counter: 0,
    };
  }
```

```
  // Log a message every time the component updates
  componentDidUpdate(prevProps, prevState) {
    console.log("Component updated!");
    console.log("Previous counter value:", prevState.counter);
    console.log("Current counter value:", this.state.counter);
  }

  // Log a message when the component is about to unmount
  componentWillUnmount() {
    console.log("Component will unmount!");
  }

  // Method to handle counter increment
  incrementCounter = () => {
    this.setState((prevState) => ({ counter: prevState.counter + 1 }));
  };

  render() {
    return (
      <div>
        <h1>Counter: {this.state.counter}</h1>
        <button onClick={this.incrementCounter}>Increment Counter</button>
      </div>
    );
  }
}

export default UpdateAndUnmountLogger;
```