Handling Events in React

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

**Event Handling in React vs. Vanilla JS**

In **vanilla JavaScript**, you use addEventListener to attach events:

button.addEventListener("click", () => alert("Clicked!"));

In **React**, you handle events directly in JSX using **camelCase** event names and functions:

<button onClick={() => alert("Clicked!")}>Click me</button>

**What Are Synthetic Events?**

React wraps native events in **Synthetic Events** to provide **cross-browser consistency**. These behave like regular events (event.target, event.preventDefault()), but work the same way across all browsers.

function handleSubmit(e) {

e.preventDefault(); // Synthetic event

}

**Summary:**

React simplifies and standardizes event handling using **synthetic events**, making it more reliable and easier to use than vanilla JS.

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

**Common React Event Handlers**

**✅ onClick – Handles clicks:**

<button onClick={() => alert("Clicked!")}>Click Me</button>

**✅ onChange – Tracks input changes:**

<input type="text" onChange={(e) => console.log(e.target.value)} />

**✅ onSubmit – Handles form submission:**

<form onSubmit={(e) => { e.preventDefault(); alert("Submitted!"); }}>

<button type="submit">Submit</button>

</form>

These use **React's Synthetic Events**, which behave like native events but work consistently across browsers.

**Why do you need to bind event handlers in class components?**

**Why Bind Event Handlers in Class Components?**

**In React class components, you need to bind event handlers so that this refers to the component instance.**

**Without binding, this inside the handler is undefined when the method is used as a callback.**

**Example Fix:**

**Bind in the constructor:**

**this.handleClick = this.handleClick.bind(this);**

**Or use an arrow function:**

**handleClick = () => {**

**// 'this' is automatically bound**

**};**

**Binding ensures this works correctly inside event handlers in class components.**

Conditional Rendering

**What is conditional rendering in React? How can you conditionally render elements in a React component?**

**Conditional rendering** means displaying elements or components **based on a condition** (like state or props). It lets you control what gets shown in the UI.

**✅ Using if statements:**

if (isLoggedIn) {

return <h1>Welcome back!</h1>;

} else {

return <h1>Please log in.</h1>;

}

**✅ Using ternary operator:**

<p>{isLoggedIn ? "Welcome!" : "Please sign in."}</p>

**✅ Using && (for one condition):**

{isLoggedIn && <button>Logout</button>}

React lets you **conditionally render** elements using JavaScript logic like if, ? :, and &&, giving you flexible control over what appears in the UI.

**Explain how if-else, ternary operators, and && (logical AND) are used in JSX for conditional rendering.**

React uses standard JavaScript syntax to conditionally render elements inside JSX. Here are the common ways:

**if-else Statement**

Used **outside** JSX, typically before the return statement.

if (isLoggedIn) {

return <h1>Welcome!</h1>;

} else {

return <h1>Please log in.</h1>;

}

**Ternary Operator (? :)**

Used **inside** JSX for inline conditions.

<h1>{isLoggedIn ? "Welcome!" : "Login"}</h1>

**Logical AND (&&)**

Renders something **only if** the condition is true.

{isLoggedIn && <button>Logout</button>}

Use if-else for complex logic, the ternary operator for inline conditions, and && for simple one-way checks in JSX.

Forms in React

**How do you handle forms in React? Explain the concept of controlled components.**

**Handling Forms in React**

In React, forms are handled by linking **input elements to state**, so you can control and respond to user input.

**Controlled Components**

A **controlled component** is a form element (like <input>) whose value is **controlled by React state**.

function MyForm() {

const [name, setName] = useState("");

const handleChange = (e) => setName(e.target.value);

return (

<form>

<input type="text" value={name} onChange={handleChange} />

<p>Your name is: {name}</p>

</form>

);

}

**Why Use Controlled Components?**

* Keeps form data in sync with component state
* Makes it easier to validate, reset, or manipulate inputs

Controlled components link form inputs to state, giving you full control over user input and making form handling more predictable.

**What is the difference between controlled and uncontrolled components in React?**

**Controlled Components** are form elements whose values are controlled by React state.

const [input, setInput] = useState("");

<input value={input} onChange={(e) => setInput(e.target.value)} />

* React manages the input value
* Easier to validate and handle data

**Uncontrolled Components** use the DOM to manage state, often with ref.

const inputRef = useRef();

<input ref={inputRef} />

* React does not control the value
* Useful for simple or non-interactive forms

Controlled = React handles input state

Uncontrolled = DOM handles input state  
Controlled components are preferred for more dynamic and interactive forms.

Lifecycle Methods (Class Components)

**What are lifecycle methods in React class components? Describe the phases of a component’s lifecycle.**

**Lifecycle methods** are special functions in React **class components** that run at specific points during a component’s life—like when it's **created**, **updated**, or **removed** from the DOM.

**Lifecycle Phases:**

1. **Mounting** (Component is added to the DOM):
   * constructor()
   * componentDidMount()
2. **Updating** (Props or state change):
   * shouldComponentUpdate()
   * componentDidUpdate()
3. **Unmounting** (Component is removed from the DOM):
   * componentWillUnmount()

Lifecycle methods help you **run code at specific stages** of a class component’s life—for setup, updates, or cleanup.

**Explain the purpose of componentDidMount(), componentDidUpdate(), and componentWillUnmount().**

**componentDidMount()**

* Runs **once** after the component is added to the DOM.
* Great for **fetching data**, setting up **timers**, or initializing **subscriptions**.

componentDidMount() {

console.log("Component mounted");

}

**componentDidUpdate(prevProps, prevState)**

* Runs **after every update** (when props or state change).
* Useful for **reacting to changes**, like updating the DOM or making API calls.

componentDidUpdate(prevProps) {

if (prevProps.id !== this.props.id) {

// Fetch new data

}

}

**componentWillUnmount()**

* Runs **right before** the component is removed from the DOM.
* Used for **cleanup** like removing timers or unsubscribing from events.

componentWillUnmount() {

clearInterval(this.timer);

}

These methods let you **hook into the component’s lifecycle** to perform actions when it mounts, updates, or unmounts.

State Management (Redux, Redux-Toolkit or Recoil)

**What is Redux, and why is it used in React applications? Explain the core concepts of actions, reducers, and the store.**

**How does Recoil simplify state management in React compared to Redux?**