Module 18 – Reactjs for Full Stack

5. Handling Events in React

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

In React, event handling is similar to vanilla JavaScript, but there are key differences due to how React works under the hood.

Feature	Vanilla JavaScript	React
Syntax	element.addEventListener('click', handler)	<button onClick={handler}>Click</button>
Binding	Manual (e.g., this context in classes)	Can be auto-bound in arrow functions or using class field syntax
Event Object	Native DOM Event	Synthetic Event (React wrapper)
Event Removal	element.removeEventListener	Handled via virtual DOM reconciliation

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

React provides several event handlers that correspond to native DOM events. These handlers are written in camelCase and passed directly in JSX.

1. onClick

The onClick event handler is triggered when a user clicks on an element like a button.

- **Usage:**
  - Often used for buttons or interactive UI elements.
  - Calls a function when the element is clicked.

2. onChange

The onChange handler is used to capture input changes in form elements like <input>, <textarea>, or <select>.

- **Usage:**
  - Useful for real-time form input handling (e.g., storing user input in state).

- Captures every keystroke or selection.

3. onSubmit

The onSubmit event is used with <form> elements to handle the form's submission logic.

- **Usage:**
  - Prevents default form submission using e.preventDefault().
  - Validates and processes form data.

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

In React class components, you need to bind event handlers to ensure that the this keyword inside the handler refers to the correct context — the component instance.

Reason:

In JavaScript, the value of this inside a method depends on how the method is called. When you pass a class method as a callback (like an event handler), it loses its original this context, so it won't refer to the component anymore unless you bind it.

Without Binding:

This will throw an error or log undefined.

```
class MyComponent extends React.Component {
 handleClick() {
  console.log(this); // undefined or wrong context
 }
 render() {
  return <button onClick={this.handleClick}>Click</button>;
 }
}
```

With Binding (3 Ways):

1. Bind in Constructor (Traditional)

```
constructor() {
 super();
 this.handleClick = this.handleClick.bind(this);
}
```

2. Arrow Function as Class Field

```
handleClick = () => {
 console.log(this); // Correctly refers to the component
}
```

3. Inline Arrow Function in JSX (Not recommended often)

```
<button onClick={() => this.handleClick()}>Click</button>
```

6. Conditional Rendering

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

- Conditional rendering means showing different UI elements based on certain conditions.
- It allows React components to render content dynamically depending on state, props, or other variables.

How to Conditionally Render Elements in React?

1. Using if or if-else statements
    - Use inside the component function to decide what to return.
2. Using Ternary Operator (condition ? true : false)

- Use inline inside JSX for simple conditions.

3. Using Logical AND (&&) Operator

- Render an element only if the condition is true.

4. Using Variables to Store JSX Elements

- Assign JSX to a variable based on condition, then use it in return.

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

1. if-else in JSX

- You cannot directly use if-else inside JSX because JSX expects expressions, not statements.
- Instead, use if-else outside JSX, usually before the return statement, to decide what to render.

2. Ternary Operator (condition ? true : false)

- The ternary operator works inside JSX because it's an expression.
- It's useful for simple conditional rendering when you want to render one thing or another.

3. Logical AND (&&)

- The && operator can be used inside JSX to render something only if the condition is true.
- If the condition is false, React ignores the expression and renders nothing.

Method	Usage	Can be used inside JSX?	Suitable for
if-else	Outside JSX to decide return	No	Complex branching and multiple returns
Ternary (?:)	Inline expression in JSX	Yes	Simple condition with two outcomes
Logical AND (&&)	Inline expression in JSX	Yes	Rendering something conditionally

7. Lists and Keys

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

Handling Forms in React

- In React, form elements like <input>, <textarea>, and <select> are usually handled as controlled components.
- This means React controls the form data by keeping the form values in the component's state.
- When a user types or selects something, React updates the state via event handlers like onChange.
- This keeps the React state as the single source of truth for form data.

What are Controlled Components?

- A controlled component is a form element whose value is controlled by React state.
- The form element's value attribute is set to the state variable.
- The onChange handler updates the state when the user interacts with the element.
- This allows you to easily access or validate form data and respond to user input immediately.

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

Controlled Components

- The form data is handled and stored in React state.
- The form element's value is set by the state via the value attribute.
- User input changes are captured through onChange handlers which update the state.
- React is the single source of truth for the form data.
- Allows easy validation, conditional disabling, and instant UI updates.
- Example: <input value={stateValue} onChange={handleChange} />

Uncontrolled Components

- Form data is handled by the DOM itself, not stored in React state.
- You access the input value using refs (e.g., React.createRef()).
- The form element maintains its own internal state.
- React does not control the value; you read it only when needed (like on submit).
- Useful for simple forms or when integrating with non-React code.
- Example: <input defaultValue="initial" ref={inputRef} />

Feature	Controlled Components	Uncontrolled Components
Data source	React state	DOM (input's internal state)
Value attribute	Controlled via value prop	Uses defaultValue prop
Updates	On every input change via onChange	Accessed via refs only
Control	Full control by React	Limited control
Use cases	Complex forms needing validation	Simple forms or third-party libs

8. Forms in React

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

Handling Forms in React

- React handles forms by linking form inputs to the component's state.
- Instead of the DOM managing the form data, React keeps the data in state variables.
- User input triggers events (like onChange) that update the state.
- The current state controls what the input displays, making the UI predictable and easy to manage.

Concept of Controlled Components

- A controlled component is a form element (like <input>, <textarea>, or <select>) whose value is controlled by React state.
- The input's value attribute is set from the state.
- When the user types or changes the input, an event handler updates the state accordingly.
- This creates a two-way binding: the UI reflects the state, and the state updates based on user input.
- Controlled components make form data handling easier, enable validation, and allow dynamic form behavior.

```
function MyForm() {
 const [name, setName] = React.useState("");
 const handleChange = (e) => {
  setName(e.target.value);  };
 const handleSubmit = (e) => {
  e.preventDefault();
  alert(`Submitted name: ${name}`);  };
 return (
  <form onSubmit={handleSubmit}>
   <input type="text" value={name} onChange={handleChange} />
   <button type="submit">Submit</button>
  </form>
 );
}
```

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

Controlled Components

- The form data is managed by React state.
- The input's value is set via the value prop tied to state.
- Changes in input update the state through onChange handlers.
- React is the single source of truth for the input value.
- Allows easy validation and dynamic input control.

<input value={stateValue} onChange={handleChange} />

Uncontrolled Components

- The form data is managed by the DOM itself.
- The input's value is accessed using refs instead of state.
- Uses defaultValue for initial value, not value.
- React does not control the input's current value.
- Useful for simple or third-party form integrations.

<input defaultValue="initial" ref={inputRef} />

Feature	**Controlled Components**	**Uncontrolled Components**
Data source	React state	DOM element
Value attribute	value prop controlled by state	defaultValue (initial only)
Updates	On every input via onChange	Accessed via refs when needed
Control level	Full React control	Limited control
Use case	Complex forms, validation	Simple forms, legacy code

9. Lifecycle Methods (Class Components)

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

- Lifecycle methods are special methods in React class components that run at specific points during a component's existence.
- They allow you to perform actions like setup, cleanup, data fetching, or updating the UI at different stages.
- Lifecycle methods only exist in class components (functional components use hooks instead).

Phases of a Component's Lifecycle

1. Mounting
    - When the component is created and inserted into the DOM for the first time.
    - Common lifecycle methods:
        - constructor() — initialize state and bind methods
        - static getDerivedStateFromProps() — update state from props before rendering
        - render() — returns the JSX to display
        - componentDidMount() — runs after component is rendered; good for data fetching
2. Updating
    - When the component re-renders due to state or prop changes.
    - Common lifecycle methods:
        - static getDerivedStateFromProps() — update state based on props
        - shouldComponentUpdate() — decide whether to re-render (performance optimization)
        - render() — update the UI
        - getSnapshotBeforeUpdate() — capture some info before the DOM updates

        - componentDidUpdate() — runs after update, useful for network requests or DOM operations

3. Unmounting
    - When the component is removed from the DOM.
    - Lifecycle method:
        - componentWillUnmount() — cleanup tasks like removing timers, canceling network requests, or removing event listeners
4. Error Handling (optional)
    - Lifecycle methods to catch errors during rendering or lifecycle methods:
        - static getDerivedStateFromError()
        - componentDidCatch()

Phase	Purpose	Common Methods
Mounting	Component created and added to DOM	constructor(), render(), componentDidMount()
Updating	Component re-renders due to props/state change	shouldComponentUpdate(), render(), componentDidUpdate()
Unmounting	Component removed from DOM	componentWillUnmount()
Error	Handle errors	getDerivedStateFromError(), componentDidCatch()

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

1. componentDidMount()

- Called once immediately after the component is mounted (inserted into the DOM).
- Purpose:
    - Initialize things that require DOM nodes (e.g., fetch data from APIs).
    - Set up subscriptions, timers, or event listeners.

- Example use cases: Fetching data, starting animations.

---

2. componentDidUpdate(prevProps, prevState)

- Called after every update except the initial render (when props or state changes).
- Purpose:
    - React to changes in props or state.
    - Perform side effects like fetching new data based on updated props.
    - Update the DOM or trigger other updates safely.
- You can compare previous and current props/state to decide whether to act.

3. componentWillUnmount()

- Called just before the component is removed from the DOM.
- Purpose:
    - Clean up resources to avoid memory leaks.
    - Remove timers, cancel network requests, and unsubscribe from event listeners or subscriptions.

Summary Table

Method	When Called	Purpose
componentDidMount()	After first render (mounting)	Initialize data, subscriptions, DOM-related setup
componentDidUpdate()	After every update (except first)	Respond to prop/state changes, side effects
componentWillUnmount()	Before removal (unmounting)	Cleanup (timers, listeners, subscriptions)

10. Hooks (useState, useEffect, useReducer, useMemo, useRef, useCallback)

Question 1: What are React hooks? How do useState() and useEffect() hooks work in functional components?

- React hooks are special functions that let you use state and lifecycle features in functional components.
- They enable functional components to manage state, side effects, context, refs, and more — things previously only possible in class components.
- Hooks make components simpler and more reusable.

useState() Hook

- Purpose: To add state to a functional component.
- Returns a state variable and a function to update it.
- Syntax:

```
const [state, setState] = useState(initialValue);
```

useEffect() Hook

- Purpose: To perform side effects in functional components (similar to lifecycle methods in classes).
- Runs after render, and can be configured to run:
    - After every render (default)
    - Only once (on mount)
    - When specific values change
- Syntax:

```
useEffect(() => {
  return () => {  };
}, [dependencies]);
```

Hook	Purpose	Key Points
useState	Add state to functional components	Returns [state, setState]; triggers re-render on update
useEffect	Perform side effects (data fetch, timers)	Runs after render; cleanup function optional; controlled by dependencies

Question 2: What problems did hooks solve in React development? Why are hooks considered an important addition to React?

1. Complexity and Boilerplate in Class Components
    - Managing state and lifecycle in class components requires a lot of boilerplate (e.g., constructors, binding this, multiple lifecycle methods).
    - Hooks let you use state and side effects without classes, making code simpler and cleaner.
2. Difficult to Reuse Logic Across Components
    - With classes, sharing stateful logic (like data fetching or subscriptions) required patterns like higher-order components or render props, which can be hard to read and maintain.
    - Hooks enable custom hooks, allowing easy reuse and composition of stateful logic.
3. Confusing Lifecycle Methods
    - Splitting related logic across multiple lifecycle methods (componentDidMount, componentDidUpdate, componentWillUnmount) leads to fragmented and hard-to-follow code.
    - Hooks like useEffect consolidate these lifecycle events in one place with clear dependency control.
4. this Keyword Confusion
    - Class components require binding this or using arrow functions, which can cause bugs and confusion for beginners.
    - Hooks avoid this entirely, since they are just functions.

Why Hooks Are an Important Addition

- Simplify Functional Components
    - They allow functional components to have state and side effects, enabling simpler and more intuitive code.
- Better Code Reusability
    - Custom hooks let developers share reusable logic cleanly across different components.
- Improved Readability and Maintenance
    - Hooks group related logic together instead of scattering it across lifecycle methods.

- No More Classes Required
    - Hooks enable building complete React apps with just functions, which are easier to understand and test.
- Encourage Best Practices
    - React's rules of hooks and patterns guide developers towards more predictable and bug-free code.

Summary:

Problem Solved	Benefit of Hooks
Boilerplate & complexity of classes	Cleaner, simpler functional components
Hard to reuse logic across components	Easy to create reusable custom hooks
Fragmented lifecycle methods	Unified side effects with useEffect
this binding confusion	No this keyword, simpler syntax

Question 3: What is useReducer ? How we use in react app?

- useReducer is a React hook used for managing complex state logic in functional components.
- It is an alternative to useState but better suited when the state depends on multiple sub-values or when the next state depends on the previous state.
- It's inspired by the Redux reducer pattern, where you update state based on an action and a reducer function.

How useReducer Works

- You define a reducer function that takes the current state and an action, then returns the new state.
- You call useReducer with the reducer and an initial state.
- It returns the current state and a dispatch function to send actions.

Syntax

const [state, dispatch] = useReducer(reducer, initialState);

reducer: (state, action) => newState function

initialState: starting state value

state: current state

dispatch: function to send actions to the reducer

Question 4: What is the purpose of useCallback & useMemo Hooks?

Purpose of useCallback

- useCallback memoizes a callback function to prevent unnecessary re-creations on every render.
- It returns a memoized version of the function that only changes if the specified dependencies change.
- Useful to optimize performance, especially when passing callbacks to child components that rely on reference equality (like React.memo).
- Helps avoid unnecessary re-renders of child components caused by changing function references.

Example of useCallback

```
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

Purpose of useMemo

- useMemo memoizes the result of a calculation to avoid expensive recalculations on every render.
- It returns a memoized value computed by a function, recalculating only when dependencies change.
- Useful for optimizing expensive computations or complex calculations.
- Helps keep rendering fast by caching computed values.

Example of useMemo

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Hook	Purpose	Use Case
useCallback	Memoize functions	Avoid recreating functions causing child re-renders
useMemo	Memoize computed values	Avoid expensive recalculations on every render

Question 5: What's the Difference between the useCallback & useMemo Hooks?

Aspect	useCallback	useMemo
Purpose	Memoizes a function so it doesn't get recreated unnecessarily.	Memoizes the result of a function (a computed value) to avoid recalculating it.
Returns	Returns a memoized callback function.	Returns a memoized value (the result of the function).
Use case	When you want to pass a stable function reference to child components or hooks.	When you want to optimize expensive calculations by caching their results.
Example usage	const memoizedFn = useCallback(() => doSomething(), [deps]);	const memoizedValue = useMemo(() => computeValue(), [deps]);
What it memoizes	The function itself.	The output of a function.

- useCallback = memoizing the function itself (the "recipe").
- useMemo = memoizing the result of calling the function (the "cake").

Question 6 : What is useRef ? How to work in react app?

- useRef is a React hook that provides a mutable ref object which persists for the full lifetime of the component.
- It is mainly used to access and interact with DOM elements directly or to store mutable values that don't cause re-renders when updated.
- The ref object has a .current property where the value is stored.

How useRef Works

- When you call useRef(initialValue), it returns an object: { current: initialValue }.
- This object stays the same between renders.
- Changing .current does not trigger a component re-render.
- Useful for:
    - Accessing DOM nodes (e.g., focus an input).
    - Keeping any mutable value around without causing re-renders (like timers, previous values, etc.).

Feature	Description
Purpose	Access DOM nodes or store mutable values without re-rendering
Returned object	{ current: initialValue }
Re-render on change?	No
Common use cases	DOM refs, timers, previous state, external mutable data