Assignment-2

Handling Events in React

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

In Vanilla JavaScript:

- Events are attached using methods like addEventListener.
- You work directly with the **DOM event** object.

```
document.getElementById("btn").addEventListener("click", function (event) {

  alert("Button clicked");

});
```

In React:

- Events are handled using **JSX attributes** like onClick, onChange, etc., written in **camelCase**.
- React wraps native events with its own **SyntheticEvent** system for **cross-browser compatibility** and performance.

```
function App() {

  const handleClick = (e) => {

    console.log("React click", e);

  };

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

}
```

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

1. **onClick – Triggered when an element is clicked:**

```
function ClickExample() {

const handleClick = () => alert("Button clicked");
```

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

2.onChange – Used with input elements to detect value changes:

```
function InputExample() {
  const handleChange = (e) => {
    console.log("Input changed to:", e.target.value);
  };
  return <input type="text" onChange={handleChange} />;
}
```

3.onSubmit – Handles form submissions:

```
function FormExample() {
  const handleSubmit = (e) => {
    e.preventDefault(); // Prevents page reload
    alert("Form submitted");
  };
  return (
    <form onSubmit={handleSubmit}>
      <input type="text" />
      <button type="submit">Submit</button>
    </form>
  );
}
```

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

In **class components**, event handler methods **don't automatically bind this** to the class instance.

Example problem:

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

✅ **Solutions:**

1. **Bind in the constructor:**

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

2.Use arrow functions (class field syntax):

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

Conditional Rendering

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

Conditional rendering in React means **dynamically displaying content based on a condition** —just like using if statements in JavaScript.

React renders different UI elements or components based on state, props, or logic.

🔧 **Ways to conditionally render in React:**

- if/else statements
- Ternary operator (condition ? true : false)
- Logical AND (&&)
- Early return
- Switch-case (for multiple conditions)

Example:-

```
function Greeting({ isLoggedIn }) {

  if (isLoggedIn) {

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

  } else {

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

  }

}
```

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

Using if/else (outside JSX)

Best for more complex logic. Cannot use if directly inside JSX.

```
function Message({ isMorning }) {

  let greeting;

  if (isMorning) {

    greeting = <h1>Good morning!</h1>;

  } else {

    greeting = <h1>Good evening!</h1>;

  }

return <div>{greeting}</div>;

}
```

Using ternary operator (inline in JSX)

```
function Greeting({ isLoggedIn }) {
```

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

Using logical AND (&&)

```
function Notification({ hasNewMessages }) {
  return (
    <div>
      <h1>Inbox</h1>
      {hasNewMessages && <p>You have new messages!</p>}
    </div>
  );
}
```

Lists and Keys

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

🔹 **Rendering a List in React**

In React, you typically use the .map() method to iterate over an array and return a component or element for each item.

📌 **Example:**

```
function FruitList() {
  const fruits = ["Apple", "Banana", "Cherry"];
  return (
    <ul>
      {fruits.map((fruit, index) => (
        <li key={index}>{fruit}</li>
```

```
    ))}
  </ul>
 );
}
```

🔹 **Why are keys important when rendering lists?**

Keys help React identify **which items have changed, been added, or removed**. This improves the performance and efficiency of the virtual DOM **diffing algorithm**.

Without keys or with incorrect keys:

- React may **re-render unnecessarily**.
- It may **mix up elements**, especially during updates.
- Component state may behave unpredictably.

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

🔹 **What are keys?**

- Keys are unique identifiers for elements in a list.
- They should be stable, predictable, and unique (e.g., an ID from your database).

📌 **Good Key Example:**

```
users.map(user => <li key={user.id}>{user.name}</li>)
```

🔹 **What happens if keys are missing or not unique?**

- React logs a **warning** in the console.
- UI may show **incorrect ordering** or **unexpected behavior** during re-renders.
- Performance degrades for large lists.
- Components may **lose state** or display **glitches** when reordered.

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 maintaining form data in the component's state and updating that state via event handlers like onChange.

📌 **Controlled components are the standard way to do this.**

🔹 **What are Controlled Components?**

A controlled component is an input element (e.g., <input>, <textarea>, <select>) whose value is controlled by React state.

🔧 **Example:**

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

function MyForm() {

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

  const handleChange = (e) => {

    setName(e.target.value);

  };

  const handleSubmit = (e) => {

    e.preventDefault();

    alert(`Hello, ${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?

Feature	Controlled Component	Uncontrolled Component
Value stored in	**React state**	**The DOM itself**
Updated via	**onChange + setState()**	**Using Refs (e.g., useRef)**
Form element control	**React fully controls input**	**Browser/DOM controls input**
Use case	**When you need form validation, dynamic input, etc.**	**When minimal control is needed (e.g., simple forms)**

Uncontrolled Example (using useRef):

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


function UncontrolledForm() {
  const nameRef = useRef();


  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Hello, ${nameRef.current.value}`);
  };


  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={nameRef} />
      <button type="submit">Submit</button>
```

```
    </form>
  );
}
```

Lifecycle Methods (Class Components)

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

🔹 **What Are Lifecycle Methods?**

In class components, lifecycle methods are special methods that run at specific points in a component's life (creation, updating, and removal). They allow you to run code at those points, like fetching data, setting up subscriptions, or cleaning up.

🔄 Phases of a Component's Lifecycle:

Phase	Description
Mounting	**Component is created and inserted into the DOM.**
Updating	**Component is re-rendered due to state or props changes.**
Unmounting	**Component is removed from the DOM.**
(Error Handling)	**(Optional) Lifecycle methods for catching and handling errors.**

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

componentDidMount()

- 📍 **When it's called: Right after the component is added to the DOM.**
- 📌 **Use it for:**
  - **Fetching data from an API**
  - **Setting up subscriptions or timers**
  - **DOM interactions**

```
componentDidMount() {

  console.log("Component mounted");

  // Fetch data, start interval, etc.

}
```

componentDidUpdate(prevProps, prevState)

- ❗ When it's called: After every update (re-render) due to props or state change.
- 📌 Use it for:
  - Responding to prop or state changes
  - Triggering side effects on updates (like refetching)

```
componentDidUpdate(prevProps, prevState) {

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

    // Re-fetch user data if the userId prop changes

  }

}
```

componentWillUnmount()

- ❗ When it's called: Right before the component is removed from the DOM.
- 📌 Use it for:
  - Cleaning up timers
  - Removing event listeners
  - Cancelling API calls or subscriptions

```
componentWillUnmount() {

  console.log("Component will unmount");

  clearInterval(this.timer); // Clean up

}
```

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

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

🔹 **What Are Hooks?**

Hooks are functions that let you use React features (like state and lifecycle methods) in functional components—without writing class components.

🔹 **useState() — Adds state to functional components**

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

function Counter() {

  const [count, setCount] = useState(0); // count = state, setCount = updater



  return (

    <button onClick={() => setCount(count + 1)}>

      Count: {count}

    </button>

  );

}
```

🔹 **useEffect() — Runs side effects (similar to componentDidMount, componentDidUpdate, and componentWillUnmount)**

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



function Timer() {

  const [time, setTime] = useState(0);



  useEffect(() => {

    const interval = setInterval(() => setTime((t) => t + 1), 1000);

    return () => clearInterval(interval); // Cleanup
```

```
  }, []); // Runs once (like componentDidMount)


  return <h1>Timer: {time}</h1>;

}
```

question 2: What problems did hooks solve in React development? Why are hooks important?

🚫 **Problems before hooks:**

- **You had to use class components for state and lifecycle features.**
- **Code reuse was hard—HOCs and render props were complex.**
- **Component logic became scattered across lifecycle methods.**

✅ **Hooks solved this by:**

- Enabling state and side effects in functional components
- Improving code reuse with custom hooks
- Making logic easier to organize and test

🎉 Why they matter: Hooks made React simpler, cleaner, and more functional, replacing the need for most class components.

Question 3: What is useReducer? How is it used in a React app?

🔹 **useReducer is a hook for complex state logic, similar to Redux-style reducers.**

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

const reducer = (state, action) => {

  switch (action.type) {

    case "increment":

      return { count: state.count + 1 };

    case "decrement":

      return { count: state.count - 1 };

    default:

      return state;

  }

};
```

```
function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 });
  return (
    <>
      <h2>{state.count}</h2>
      <button onClick={() => dispatch({ type: "increment" })}>+</button>
      <button onClick={() => dispatch({ type: "decrement" })}>-</button>
    </>
  );
}
```

Question 4: What is the purpose of useCallback() and useMemo() hooks?

Hook	Purpose
useCallback	**Memoizes a function**, prevents unnecessary re-creations
useMemo	**Memoizes a value/result**, avoids re-computation

Question 5: What's the Difference between useCallback() and useMemo()?

Feature	useCallback	useMemo
What it returns	**A memoized function**	**A memoized value**
Use case	**Prevent re-creating functions on re-render**	**Avoid expensive recalculations**
Syntax	**useCallback(fn, deps)**	**useMemo(() => result, deps)**

📌 Example:

```
const memoizedFn = useCallback(() => doSomething(a, b), [a, b])
```

```
const memoizedValue = useMemo(() => computeExpensiveValue(x), [x])
```

Question 6: What is useRef()? How does it work in React apps?

🔹 Purpose of useRef():

- Access **DOM elements** directly (like document.querySelector)
- Store **mutable values** that **persist across renders** without causing re-renders

📌 **Example 1: Accessing DOM**

```
import React, { useRef, useEffect } from "react";


function FocusInput() {
  const inputRef = useRef();


  useEffect(() => {
    inputRef.current.focus(); // Access DOM node
  }, []);


  return <input ref={inputRef} type="text" />;
}
```

Example 2: Storing a value without re-rendering

```
const count = useRef(0)
```

```
count.current += 1
```