1. Explain React events

React events are synthetic representations of native browser events. When you handle events in React, you're not directly interacting with the browser's native DOM events. Instead, React wraps these native events with an abstraction layer called the **Synthetic Event System**.

- **Cross-browser Consistency:** React's Synthetic Event system normalizes events across different browsers. This means you don't have to worry about inconsistencies or quirks between how different browsers implement their native events (e.g., `event.preventDefault()` works the same everywhere).
- **Performance Optimization:** React uses event delegation. Instead of attaching event listeners to every individual DOM element, React attaches a single event listener to the root of your application (e.g., the `document` or the element where your React app is mounted). When an event occurs, it bubbles up to this root listener, and React then dispatches the event to the appropriate component. This significantly reduces memory overhead and improves performance.
- **Mimics Native Browser Events:** Synthetic events have the same interface as native browser events, including methods like `preventDefault()` and `stopPropagation()`. This makes them feel familiar to developers accustomed to native JavaScript event handling.
- **Event Pooling (Historical Note - Less relevant for modern React):** In older React versions, synthetic events were pooled for performance. This meant event objects were reused, and their properties were nulled out after the event callback was invoked. Developers had to call `event.persist()` if they needed to access event properties asynchronously. In modern React (v17+), Synthetic Events are no longer pooled by default, so you can access event properties even after the event handler has finished executing without calling `event.persist()`.

2. Explain about event handlers

An **event handler** in React is a function that gets executed when a specific event occurs on a React element. You assign these functions as values to event attributes in your JSX.

- **CamelCase Naming:** React event attributes are named using camelCase (e.g., `onClick`, `onChange`, `onSubmit`, `onKeyPress`), unlike standard HTML attributes which are often lowercase (e.g., `onclick`, `onchange`).
- **Pass a Function, Not a String:** In HTML, you might pass a string of JavaScript code to an event attribute (`<button onclick="alert('Hello')">`). In React, you pass a **function reference** (or an arrow function that calls a function) directly within curly braces `{}`.

Ways to Define Event Handlers:

a. **Inline Functions (Arrow Functions for simple cases or passing arguments):** This is common for simple actions or when you need to pass arguments to the handler.

b. **Method on a Class Component:** In class components, event handlers are typically defined as methods of the class. You often need to `bind this` to ensure it refers to the component instance.

c. **Functional Components (using `useState` and plain functions):** In functional components (with React Hooks), event handlers are just regular JavaScript functions. `this` binding is generally not a concern because functional components don't have their own `this` instance in the same way class components do.

3. Define Synthetic event

A **Synthetic Event** in React is an object that acts as a cross-browser wrapper around the browser's native event. It's an instance of `SyntheticEvent`, which is part of React's Synthetic Event System.

- **Purpose:** Its primary purpose is to provide a consistent and performant way to handle events across different browsers and ensure that events behave identically, regardless of the underlying browser implementation.
- **Properties and Methods:** Synthetic events expose the same interface as native browser events. This means you can access properties like `event.target`, `event.currentTarget`, `event.preventDefault()`, `event.stopPropagation()`, etc., just as you would with native DOM events.
- **Normalization:** For example, `event.which` (which returns the button or key pressed) can behave differently across browsers for native events, but React's synthetic event ensures its value is consistent.
- **Underlying Native Event:** You can access the original native browser event using `event.nativeEvent`. However, this is rarely necessary and generally discouraged as it reintroduces browser inconsistencies.

Example:

```
function MyComponent() {
  const handleChange = (event) => {
    // 'event' here is a SyntheticEvent object
    console.log("Input value changed to:", event.target.value);
    // event.preventDefault() can be used on form submissions, etc.
  };

  return (
    <input type="text" onChange={handleChange} />
  );
}
```

4. Identify React event naming convention

React follows a specific naming convention for its event attributes in JSX, which differs slightly from standard HTML.

- **CamelCase:** All React event attributes are named using **camelCase**.
  - **HTML:** `onclick`, `onchange`, `onsubmit`
  - **React (JSX):** `onClick`, `onChange`, `onSubmit`
- **Common Event Types:**
  - **Mouse Events:** `onClick`, `onDoubleClick`, `onMouseDown`, `onMouseUp`, `onMouseEnter`, `onMouseLeave`, `onMouseMove`, `onMouseOver`, `onMouseOut`
  - **Keyboard Events:** `onKeyDown`, `onKeyPress`, `onKeyUp`
  - **Form Events:** `onChange` (for input, select, textarea value changes), `onSubmit`, `onInput`, `onFocus`, `onBlur`
  - **Touch Events:** `onTouchStart`, `onTouchEnd`, `onTouchMove`, `onTouchCancel`
  - **Clipboard Events:** `onCopy`, `onCut`, `onPaste`
  - **Focus Events:** `onFocus`, `onBlur`
  - **Image Events:** `onLoad`, `onError` (for `<img>` tags)
  - **Media Events:** `onPlay`, `onPause`, `onEnded` (for audio/video)

Example:

```
<input type="text" onChange={handleInputChange} />

<button onClick={handleButtonClick}>Submit</button>

<form onSubmit={handleFormSubmit}>

  <textarea onFocus={handleFocus} onBlur={handleBlur}></textarea>

  <input type="checkbox" onChange={handleCheckboxChange} />

</form>
```