Q1. How are events handled in React compared to vanilla JavaScript? Explain the concept of synthetic events.

Ans: Event handling in React is similar to handling events in vanilla JavaScript, but there are some important differences in syntax, behavior, and implementation.

- Event Handling in Vanilla JavaScript

In vanilla JavaScript, events are attached directly to DOM elements using methods like addEventListener().

Example (Vanilla JavaScript):

```
button.addEventListener("click", function () {
  alert("Button clicked");
});
```

- Events are handled directly on real DOM elements
- Different browsers may handle events slightly differently
- Developers must manually manage event binding and cleanup

- Event Handling in React

In React, events are handled using JSX attributes and are written in camelCase. Instead of passing a string, a function is passed as the event handler.

Example (React):

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

function handleClick() {
  alert("Button clicked");
}
```

- Events are attached using JSX
- Event handlers are written as functions
- React manages event binding automatically

- Key Differences Between React and Vanilla JavaScript Event Handling

Feature	Vanilla JavaScript	React
Syntax	addEventListener()	JSX event props
Naming	lowercase (onclick)	camelCase (onClick)
Event Object	Native browser event	Synthetic Event
Cross-browser support	Manual handling	Built-in
Cleanup	Manual	Automatic

- What are Synthetic Events in React?

A Synthetic Event is a wrapper around the native browser event provided by React. React creates a single, consistent event system that works the same across all browsers.

Instead of using the browser's native event directly, React uses SyntheticEvent, which normalizes event behavior.

Example:

```
function handleClick(e) {
  console.log(e); // SyntheticEvent
}
```

- Why Synthetic Events are Important

1. Cross-Browser Compatibility
   Synthetic events behave the same way in all browsers.

2. Better Performance
   React uses event delegation, attaching a single event listener at the root level instead of multiple listeners on individual elements.
3. Consistent API
   Developers get the same event properties regardless of browser differences.
4. Automatic Memory Management
   React automatically manages event lifecycle, reducing memory leaks.

- Example Comparing Both

Vanilla JavaScript:

```
function handleClick(event) {
  console.log(event.target);
}
```

React (Synthetic Event):

```
function handleClick(e) {
  console.log(e.target);
}
```

Usage looks similar, but React's event is a Synthetic Event.

Q2. What are some common event handlers in React.js? Provide examples of onClick, onChange, and onSubmit.

Ans: In React.js, event handlers are used to respond to user actions such as clicking a button, typing in an input field, or submitting a form. React uses camelCase naming for events and passes a function instead of a string.

Some of the most commonly used event handlers in React are onClick, onChange, and onSubmit.

1. onClick

The onClick event is triggered when a user clicks on an element such as a button.

Example:

```
function ClickExample() {
  const handleClick = () => {
```

```
    alert("Button clicked!");
  };

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

2. onChange

The onChange event is triggered when the value of an input field changes. It is commonly used to handle user input in forms.

Example:

```
function ChangeExample() {
  const handleChange = (e) => {
    console.log(e.target.value);
  };

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

3. onSubmit

The onSubmit event is used with forms and is triggered when a form is submitted. It usually includes preventDefault() to stop the page from reloading.

Example:

```
function SubmitExample() {
  const handleSubmit = (e) => {
    e.preventDefault();
    alert("Form submitted!");
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" />
```

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

Q3. Why do you need to bind event handlers in class components?

Ans: In React class components, event handlers often need to be bound to the component instance so that the keyword this correctly refers to the current component. Without binding, this inside the event handler would be undefined or not refer to the component, causing errors when accessing this.state or this.props.

Why binding is required

In JavaScript, the value of this depends on how a function is called, not where it is defined. When an event handler is passed as a callback, it loses its context, so this no longer points to the class component.

Example without binding (Error case):

```
class Counter extends React.Component {
  state = { count: 0 };
  handleClick() {
    console.log(this.state.count); //  this is undefined
  }
  render() {
    return <button onClick={this.handleClick}>Click</button>;
  }
}
```

How to bind event handlers

1. Binding in the constructor (Traditional way)

```
class Counter extends React.Component {
  constructor() {
    super();
```

```
    this.state = { count: 0 };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    console.log(this.state.count); //  works
  }

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

2. Using arrow functions (Recommended)

Arrow functions automatically bind this to the class instance.

```
class Counter extends React.Component {
  state = { count: 0 };

  handleClick = () => {
    console.log(this.state.count);
  };

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