React Events :

1. **React Synthetic Event Object**: In React, when an event occurs (e.g., a button click, keypress, etc.), the browser generates a native event object. React intercepts these native events and creates a synthetic event object using an event wrapper. This synthetic event object provides a consistent interface across different browsers and is used for event handling in React.

2. **Event Pooling**: React implements event pooling to optimize performance. This means that instead of creating a new synthetic event object for every event, React reuses the same synthetic event object from a pool for events of the same type.

3. **Resetting of Event Object**: After the event handler has completed executing and any synchronous code related to the event is done, React asynchronously resets the synthetic event object. This reset process clears the event object's properties to their initial values, effectively making it ready for reuse.

4. **Single Synthetic Event Object for Events of the Same Type**: When you have multiple events of the same type (e.g., multiple `onClick` events), React uses the same synthetic event object from the event pool for all these events. This means that multiple event handlers for the same event type will receive the same synthetic event object. The event object will be reused and reset after each handler has finished executing.

5. **Accessing Event Properties Asynchronously**: If you need to access event properties asynchronously (e.g., inside a `setTimeout` callback), you should use `event.persist()`. This prevents React from resetting the event object, ensuring that the event properties remain accessible even after the event handler has completed.

6. **Efficient Event Handling**: Event pooling helps reduce memory overhead and improve performance by reusing synthetic event objects and avoiding unnecessary object creation and garbage collection.

In conclusion, React creates a single synthetic event object from the event pool for events of the same type, and this object is reused for all event handlers of that type. The event object is reset after each handler has finished executing. To access event properties asynchronously, use `event.persist()` to prevent the event object from being reset and to ensure the properties remain accessible beyond the synchronous event handling phase. React's event pooling and synthetic event system help provide a consistent and efficient event handling experience across different browsers and events in React applications.

Handling events with React is fairly similar to handling events on DOM elements.

However, rather than calling `addEventListener`, as you would in Vanilla JavaScript, you provide an event listener when the element is initially rendered.

And, there are a few syntactic differences:

- React events are named in `camelCase` (like `onClick`), rather than all lowercase.
- Using JSX, you pass a function as the event handler, rather than a string.

React makes use of its own event system to provide cross-browser compatibility.

To do this, React wraps native browser events in its own structure called a `SyntheticEvent` and passes them to React event handlers. React has defined these synthetic events according to the W3C spec, which affords them cross-browser compatibility.

Because React's `SyntheticEvent` has the same interface as a native browser event, you can make use of methods like `preventDefault()` and `stopPropagation()`.

Every `SyntheticEvent` object has the following attributes:

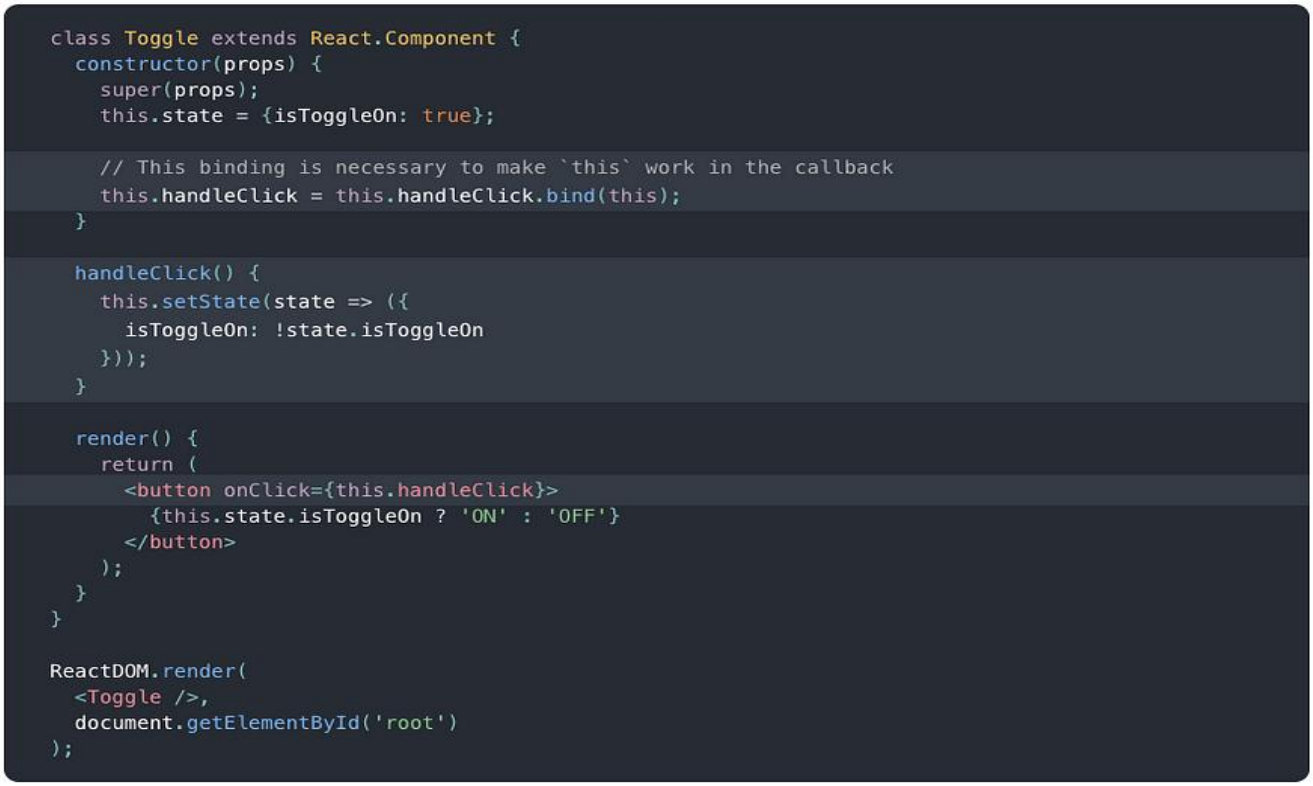
```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
DOMEventTarget target
number timeStamp
string type
```

https://reactjs.org/docs/events.html

When you define a class-based component, a common pattern is for an event handler to be a method on the class.

In the below example, the `Toggle` component renders a button that allows the user to toggle between “ON” and “OFF” states.

The `handleClick()` method changes the state whenever a click event occurs, and the `render()` method displays the current state on the DOM:



```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

ReactDOM.render(
  <Toggle />,
  document.getElementById('root')
);
```

https://reactjs.org/docs/handling-events.html

For performance purposes, React has a built-in feature that “pools” its `SyntheticEvents`. This means that after an event handler is invoked, all of the properties of the `event.target` are reset to `null`.

By nullifying the properties after the execution of the callback, the event object can be reused by React elsewhere in our application.

This is a great example of React’s mission to be DRY, modular, and reusable.

By virtue of event pooling, the browser doesn't need to allocate memory for a new object for each instance of an event. Instead, it will use the same event object in-memory for all event instances.

After the event handler has finished executing, the keys of the event object remain in place, but the values of each key are reset to null. This frees up memory space, but it also means that the values of the executed event are lost.

Because of this nullification, the properties cannot be accessed asynchronously.

By the time an asynchronous function like `setState()` is executed, it will try to access the event's properties, like `event.currentTarget.value` and, as we've seen, all of the event object's properties have been reset to `null` so the return value will be... `null`. Oh no!

You'll likely encounter a familiar error like `Uncaught TypeError: Cannot read property 'data' of null`.

To prevent an event object from being pooled, and therefore nullified, we call `event.persist()` at the beginning of the function.

Calling this method will remove the `SyntheticEvent` from the pool and the event object will not be reused by later DOM events; this allows references to the event to be retained asynchronously.

As an alternative to `event.persist()`, you can store the data you need in a variable (ex. `const pug = event.target`).

In this example, `pug` would house the reference to the DOM object, which is independent of a reference to the event object. By saving the `event.target` to a variable, we can retain access to that data without needing to access the target through the actual event object.