Mini-Project(SCORE- KEEPER)

Events in JSX

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

For example, the HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

is slightly different in React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Another difference is that you cannot return false to prevent default behavior in React. You must call `preventDefault` explicitly. For example, with plain HTML, to prevent the default form behavior of submitting, you can write:

```
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

In React, this could instead be:

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
```

```
  }
  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

Here, e is a synthetic event. React defines these synthetic events, so you don't need to worry about cross-browser compatibility. React events do not work the same as native events. See the SyntheticEvent reference guide to learn more.

When using React, you generally don't need to call addEventListener to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered.

Virtual DOM under the hood

The virtual DOM (VDOM) is a programming concept where an ideal, or "virtual", representation of a UI is kept in memory and synced with the "real" DOM by a library such as ReactDOM. This process is called reconciliation.
In reality, the virtual DOM is just an organized collection of React elements — plain objects, and it mimics the browser DOM in a way that is easier to maintain and update.
Let's take a step back and check out an example of a React element.

```
const title = <h1>Hello, world!</h1>
```

JSX does the heavy lifting to convert the familiar HTML syntax into a React element. Without JSX, this is just:

```
const title = React.createElement('h1', null, 'Hello, world!')
```

And below is the created React element under the hood.

```
{
type: "h1",
props: {
  children: "Hello, world!"
  }
}
```

More on Event-Handling

When you define a component using an ES6, a common pattern is for an event handler like ‘handleClick’ to be a method on the components. For example, the Toggle component returns a button that lets the user toggle between “ON” and “OFF” states:

```
var state = {isToggleOn: true};
handleClick() {
    state.isToggleOn = !isToggleOn;
     // rerender the App component
}
function Toggle{
      return (
      <button onClick={handleClick}>
        {state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
}
```

The alternate way to pass arguments to event handlers is using inline functions-

```
      <button onClick={() => state.isToggleOn = !isToggleOn; }>
         {state.isToggleOn ? 'ON' : 'OFF'}
      </button>
```

You have to be careful about the meaning of this in JSX callbacks. In JavaScript, class methods are not bound by default. If you forget to bind `this.handleClick` and pass it to `onClick`, this will be `undefined` when the function is called.
But this is not the case in functional components and that's why we are using public fields syntax to correctly bind callbacks.

Forms in JSX

HTML form elements work a bit differently from other DOM elements in React because form elements naturally keep some internal state. For example, this form in plain HTML accepts a single name:

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

This form has the default HTML form behavior of browsing to a new page when the user submits the form. If you want this behavior in React, it just works. But in most cases, it's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered. The standard way to achieve this is with a technique called **"controlled components"**.

```
//variable to store form's values.
var state = {value: ''};

//methods used on handling Events in JSX
handleChange(event) {
    state = {value: event.target.value};
  }
handleSubmit(event) {
    alert('A name was submitted: ' + state.value);
```

```
    event.preventDefault();
}

// created form in JSX
const Form = () =>{
    return (
      <form onSubmit={handleSubmit}>
        <label>
          Name:
          <input type="text" value={state.value}
            onChange={handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
}
```

In a controlled component, form data is handled by a React component. The alternative is uncontrolled components, where form data is handled by the DOM itself. To write an uncontrolled component, instead of writing an event handler for every state update, you can **use a ref** to get form values from the DOM.

Iterate over Arrays in JSX

Given the code below, we use the map() function to take an array of numbers and double their values. We assign the new array returned by map() to the variable doubled and log it: For example:

```
const numbers = [1, 2, 3, 4, 5];

const doubled = numbers.map((number) => number * 2);

console.log(doubled);
```

This code logs **[2, 4, 6, 8, 10]** to the console.

You can build collections of elements and **include them in JSX** using curly braces **{}**.

Note- The map() method is used to transform the elements of an array, whereas the forEach() method is used to loop through the elements of an array. The map() method can be used with other array methods, such as the filter() method, whereas the forEach() method cannot be used with other array methods.

Few Important Concepts

Creating Refs

Refs are created using `React.createRef()` and attached to React elements via the `ref` attribute. Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

Accessing Refs

When a ref is passed to an element in render, a reference to the inputRef becomes accessible at the current attribute of the ref.

```
let inputRef = React.createRef();
```

The value of the **ref** differs depending on the type of the node:

- When the `ref` attribute is used on an HTML element, the ref is created in the constructor with `React.createRef()` receives the underlying DOM element as its current property.
- When the `ref` attribute is used on a custom class component, the `ref` object receives the mounted instance of the component as its **current**.

Adding a Ref to a DOM Element

This code uses a ref to store a reference to a DOM node:

```
const Form = () =>{
        <form onSubmit={handleSubmit}>
          <input ref = {inputRef} placeholder="Name"/>
            <button> Submit </button>
  </form>
 }
```

- Pass it as `<input ref={inputRef}>.` This tells React to put this <input>'s DOM node into `inputRef.current`.
- In the handleClick function, read the input DOM node from inputRef.current and call focus() on it with `inputRef.current.focus().`
- Pass the handleClick event handler to <button> with onClick.

While DOM manipulation is the most common use case for refs, the `createRef` can be used for storing other things outside React. Similarly to the state, refs remain between renders. Refs are like state variables that don't trigger re-renders when you set them.

SyntheticEvent

Your event handlers will be passed instances of `SyntheticEvent`, a cross-browser wrapper around the browser's native event. It has the same interface as the browser's native event, including `stopPropagation()` and `preventDefault()`, except the events work identically across all browsers.
React normalizes events so that they have consistent properties across different browsers.
To explore more, you can click here - Click

Summarising it

Let’s summarise what we have learned in this Lecture:

- Learned about the Events in JSX.
- Learned about Form in JSX.
- Learned about Virtual DOM under the hood.
- Learned about how to store elements in an array and populate it.

Some References:

- More information JSX Events: Link
- More information on Ref and the DOM: Link
- To read more about Form: Link