Lecture 8:

React Guide

1. React JSX
2. React Components
3. React Class Components
4. React Function Components
5. Styling React Using CSS
6. React Hooks
7. React useState Hook
8. Counter App using React

React JSX

JSX stands for JavaScript XML.JSX allows us to write HTML in React.JSX makes it easier to write and add HTML in React. JSX (JavaScript XML) is a syntax extension for JavaScript that allows you to write HTML-like code directly in your JavaScript files. It is commonly used with **React** to define the structure of UI components in a more readable and expressive way.

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement() and/or appendChild() methods.

Key Features of JSX:

1. **HTML-like Syntax** – Write markup inside JavaScript.

```
const element = <h1>Hello, World!</h1>;
```

2. **Embed JavaScript Expressions** – Use `{}` to include dynamic values.

```
const name = "Alice";
const greeting = <p>Hello, {name}!</p>;
```

3. **Self-Closing Tags** – Similar to HTML, but mandatory for elements without children.

```
const img = <img src="logo.png" alt="Logo" />;
```

4. **Components as Tags** – Use custom React components like HTML tags.

```
function Welcome() {
  return <h1>Welcome!</h1>;
}
const app = <Welcome />;
```

Why Use JSX?

- **Readability** – Easier to visualize UI structure.
- **Performance** – React optimizes JSX into efficient JavaScript.
- **Expressiveness** – Combines logic and markup naturally.

React Components: The Building Blocks of React Apps

React components are **reusable, self-contained pieces of UI** that can be composed together to build complex interfaces. They can be thought of as **JavaScript functions that return JSX** (or HTML-like markup)

```
import "./App.css";

function Greeting() {

  return <h1>Hello</h1>;

}

function App() {

  return (

    <div>

      <Greeting />

      <h1>hiii</h1>

    </div>

  );

}

export default App;
```

Types of React Components

1. Functional Components (Recommended)
2. Class Components(see w3school react tutorial)

Functional Components

A functional component is just a function that returns JSX:

```
function Welcome() {
  return <h1>Hello, React!</h1>;
}
```

- **Uppercase naming** (`Welcome`, not `welcome`) is a convention to distinguish components from regular functions.
- They can be **exported** for use in other files:

```
export default function Welcome() {
  return <h1>Hello, React!</h1>;
}
```

Using Props (Passing Data)

Props (short for "properties") allow data to be passed into components.

```
Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}

// Usage:
<Greeting name="Alice" />  // Output: "Hello, Alice!"
```

For example

```
Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
App() {
  return (
    <div>
   <Greeting name="Alice" />
    </div>  );}export default App;
```

Destructuring props for cleaner code:

```
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}
```

```
import logo from "./logo.svg";
import "./App.css";


function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}function App() {
  return (
    <div>
   <Greeting name="Alice" />
    </div>
  );}export default App;
```

Styling React Using CSS

There are many ways to style React with CSS, this tutorial will take a closer look at three common ways:

- Inline styling
- CSS stylesheets
- CSS Modules

Inline Styling

To style an element with the inline style attribute, the value must be a JavaScript object:

```
Function mypage {
  return (
    <>
```

```
    <h1 style={{color: "red"}}>Hello Style!</h1>

      <h1 style={{backgroundColor: "lightblue"}}>Hello Style!</h1>


    <p>Add a little style!</p>
  </>
 );
}
```

JavaScript Object

You can also create an object with styling information, and refer to it in the style attribute:

```
function Header () {
 const myStyle = {
  color: "white",
  backgroundColor: "DodgerBlue",
  padding: "10px",
  fontFamily: "Sans-Serif"
 };
 return (
  <>
   <h1 style={myStyle}>Hello Style!</h1>
   <p>Add a little style!</p>
  </>
 );
}
```

CSS Stylesheet

You can write your CSS styling in a separate file, just save the file with the .css file extension, and import it in your application.

```
import './App.css';

const Header = () => {

 return (

  <>

   <h1>Hello Style!</h1>

   <p>Add a little style!.</p>
```

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

CSS Modules

Another way of adding styles to your application is to use CSS Modules.

CSS Modules are convenient for components that are placed in separate files.

Create the CSS module with the .module.css extension, example: my-style.module.css.

```
.bigblue {
  color: DodgerBlue;
  padding: 40px;
  font-family: Sans-Serif;
  text-align: center;
}
```

Import the stylesheet in your component:

```
import styles from './my-style.module.css';
function Car {
  return <h1 className={styles.bigblue}>Hello Car!</h1>;
}export default Car;
```

Import the component in your application:

```
import ReactDOM from 'react-dom/client';
import Car from './Car.js';
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Car />);
```

And run example

React Hooks: A Complete Guide

Hooks are functions that let you "hook into" React state and lifecycle features from **functional components**. Introduced in React 16.8, they allow you to use state and other React features without writing classes.

Why Use Hooks?

✅ **Simplify component logic** (no more `this` binding issues)
✅ **Reuse stateful logic** between components (custom hooks)
✅ **Organize related code** together (instead of splitting across lifecycle methods)
✅ **Work with functional components** (modern React best practice)

React **useState** Hook

The `useState` hook is the most fundamental React Hook for adding **state management** to functional components. It lets you store and update data that persists between re-renders.

Import `useState`

To use the `useState` Hook, we first need to `import` it into our component.

At the top of your component,

`import` the `useState` Hook.

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

Basic Syntax

```
import { useState } from 'react';

const [state, setState] = useState(initialValue);
```

- `state` → Current state value
- `setState` → Function to update state
- `initialValue` → Starting value (any type: number, string, array, object, etc.)

Ecample

```
function FavoriteColor() {
  const [color, setColor] = useState("red");

  return <h1>My favorite color is {color}!</h1>
}
```

Update State

To update our state, we use our state updater function.

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return (
    <>
      <h1>My favorite color is {color}!</h1>
```

```
      <button
        type="button"
        onClick={() => setColor("blue")}
      >Blue</button>
    </>
  )
}
```

Another example

```
function Car() {
  const [brand, setBrand] = useState("Ford");
  const [model, setModel] = useState("Mustang");
  const [year, setYear] = useState("1964");
  const [color, setColor] = useState("red");

  return (
    <>
      <h1>My {brand}</h1>
      <p>
        It is a {color} {model} from {year}.
      </p>
    </>
  )
}
```

UseState hook contains an object

```
function Car() {
  const [car, setCar] = useState({
    brand: "Ford",
    model: "Mustang",
    year: "1964",
    color: "red"
  });

  return (
    <>
      <h1>My {car.brand}</h1>
      <p>
        It is a {car.color} {car.model} from {car.year}.
      </p>
    </>
  )
}
```

Updating Objects and Arrays in State

When state is updated, the entire state gets overwritten.

What if we only want to update the color of our car?

If we only called `setCar({color: "blue"})`, this would remove the brand, model, and year from our state.

We can use the JavaScript spread operator to help us.

Example:

Use the JavaScript spread operator to update:

```
function UserProfile() {

  const [user, setUser] = useState({

    name: 'Alice',

    age: 25,

  });



 Function  updateName (){

    setUser({ ...user, name: 'Bob' });

  };



  return (

    <div>

      <p>Name: {user.name}</p>

      <p>Age: {user.age}</p>

      <button onClick={updateName}>Change Name</button>

    </div>

  );

}
```

Examples

Counter App

```
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </div>
  );
}
```