# LESSON 04
# JSX, Props, and State Concepts

## WEEK 01

# Objectives

- Understanding JSX in React
- Working with Props
- Managing State in React

# Understanding JSX

# Understanding JSX

## ❖ What is JSX?

- JSX is a syntax extension for JavaScript that allows you to write HTML-like code inside JavaScript, which React can render to the DOM. It simplifies UI development by combining the structure and behavior in one place.

  **const element = <h1>Hello, world!</h1>;**

## ❖ Why JSX?

- JSX makes the code more readable and concise, allowing developers to write components in a declarative way.
- React components use JSX to define the structure of the UI.

## ❖ JSX Expression Syntax

- JSX can be used inside JavaScript functions and expressions. It can include variables, functions, and even conditions.

  **const name = "John";**

  **const element = <h1>Hello, {name}!</h1>;**

# JSX Syntax Rules

- **Self-Closing Tags**
  DESC: All JSX tags must be self-closed, even for elements that don't contain children, such as <img /> or <input />.
- **Wrapping Tags**
  JSX elements must have one root element. Multiple elements should be wrapped in a single parent element, like a  or React Fragment (<> and </>).
- **Using Expressions Inside JSX**
  You can include any JavaScript expression inside curly braces {}. For example: {5 + 2}, {variableName}, or {myFunction()}.
- **Attributes in JSX**
  Attributes in JSX follow the camelCase convention. For instance, use className instead of class and htmlFor instead of for.
- **Comments in JSX**
  Comments in JSX are enclosed in curly braces with /* */. Example: {/* This is a comment */}.

# JSX in React Components

- **Using JSX in Components**
  JSX is commonly used to define the structure of React components, which are the building blocks of React applications.
- **Rendering JSX**
  The JSX code inside components is rendered by React using ReactDOM.render(), which displays the components in the browser.
- **Props in JSX**
  Props are passed to components as attributes in JSX. You can access them inside the component using props.
  Example: <MyComponent name="Alice" />
- **State and JSX**
  React components can use state to dynamically render JSX content based on the current state of the component.
- **Conditional Rendering in JSX**
  You can use JavaScript conditional expressions like ternary operators inside JSX for dynamic content rendering.

# JSX with Expressions and Loops

## ❖ JavaScript Expressions Inside JSX

You can use JavaScript expressions inside JSX to dynamically render data.

Example: {user.name}.

## ❖ Conditionals in JSX

Use JavaScript operators like if or ternary operators within JSX to conditionally render components or elements.

Example: {isLoggedIn ? <LogoutButton /> : <LoginButton />}

## ❖ Looping in JSX

Loop over arrays to render lists in JSX using JavaScript's map() method.

Example: ['Apple', 'Banana', 'Orange']. map(item => <li key={item}>{item}</li>);

# Working with Props

# Introduction to Props in React

- **Definition of Props:** Props (short for properties) are inputs to React components, which help configure and customize components.
- **Basic Usage:** Props allow data to be passed from a parent component to a child component.
- **Why use Props with TypeScript:** TypeScript adds static typing to props, helping to catch errors at compile time.

# Defining Simple Types:

- **Props can be simple data types such as:**
  - strings
  - numbers
  - boolean
  - object
  - array

# Default Props

## ❖ Using Default Props

Default values for props can be specified to ensure components function even if a prop is missing.

```
1  type Props = { message?: string };
2
3  export default function DefaultProps({ message = '' }: Props) {
4    return <div>{message}</div>;
5  }
6
```

# Function as Props

❖ **Passing Functions as Props:**

Props can also be functions passed to child components for handling events or logic.

```tsx
type Props = {
  onClick: () => void;
};

export default function PropsAsFunction({ onClick }: Props) {
  return (
    <div>
      <button onClick={onClick}>Click Me</button>
    </div>
  );
}
```

# Destructuring Props

## ❖ Destructuring in Function Parameters

Extract props directly in the function signature for cleaner code.

```tsx
type Props = {
  name: string;
  age: number;
};

export default function DestructuringProps({ name, age }: Props) {
  return (
    <div>
      <p>Name: {name}</p>
      <p>Age: {age}</p>
    </div>
  );
}
```

# Props with Children

## ❖ Children Prop

React automatically includes a children prop, which is useful for nested components.

```tsx
import React from 'react';

type Props = {
  children: React.ReactNode;
};

export default function PropsWithChildren({ children }: Props) {
  return (
    <div>
      <h2>PropsWithChildren</h2>
      <div>{children}</div>
    </div>
  );
}

```

# PropTypes with Arrays and Objects

## ❖ Arrays and Objects as Props

Props can also be arrays or objects, allowing more complex data structures to be passed.

```
type Props = {
  items: string[];
  user: {
    id: number;
    name: string;
  };
};
```

# Component tutorials

1. Create SuperButton component: *Include props: Children, Icon, ButtonType, Style, TextStyle, Disabled, Loading.*
2. Importing and Exporting Components.
3. Writing Markup with JSX.
4. Passing Props to a Component.
5. Conditional Rendering.

# Managing State in React

# Introduction to State in React

- **What is State?**
  State is a built-in object used to hold data that may change over time.
- **Why is State Important?**
  It helps components respond to user input and behave dynamically.
- **Where to Use State?**
  Inside function components using the useState hook.
- **State vs Props**
  Props are passed in; state is managed internally.
- **Triggering Rerenders**
  Updating state causes React to re-render the component.
- **State Lifecycle**
  From initialization to updates and unmounting.

# Managing State in React

## ❖ What is State?

- ➢ State is a built-in object that stores dynamic data for a component, triggering re-renders when updated.

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

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

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

# Why State Matters

## ❖ Importance of State

- ➢ Controls re-rendering.
- ➢ Enables interactivity.
- ➢ Drives conditional rendering.
- ➢ Influences component lifecycle.
- ➢ Shared state can connect components.

# useState Hook Basics

## ❖ useState Hook

- ➢ Syntax: const [value, setValue] = useState(initialValue)
- ➢ Used for primitive/local state.
- ➢ Triggers re-render on update.

```
1  function Toggle() {
2    const [isOn, setIsOn] = useState(false);
3    return <button onClick={() ⇒ setIsOn(!isOn)}>{isOn ? 'On' : 'Off'}</button>;
4  }
```

# Updating State Correctly

- **Avoid direct mutation:**
  - Always use setState functions.
  - State updates are asynchronous.
  - Use previous state:

    setCount(prev => prev + 1)

# Initializing State

## ❖ Lazy Initialization

➢ Use a function for expensive initial state to compute it only once.

```
1  function App() {
2    const [data, setData] = useState(() ⇒ expensiveComputation());
3    return <div>{data}</div>;
4  }
5
6  function expensiveComputation() {
7    return Array(1000).fill('data');
8  }
```

# Updating Objects in State

## ❖ Copy and Update

➢ Spread objects to create a new copy when updating state properties.

```
1  function Form() {
2    const [form, setForm] = useState({ name: '', age: 0 });
3    const updateName = (e) ⇒ {
4      setForm({ ...form, name: e.target.value });
5    };
6    return <input value={form.name} onChange={updateName} />;
7  }
```

# Updating Arrays in State

## ❖ Non-Mutative Updates

- ➢ Use array methods like map, filter, or spread to update arrays.

```
1  function Todos() {
2    const [todos, setTodos] = useState(['Task 1']);
3    const removeTodo = (index) ⇒ {
4      setTodos(todos.filter((_, i) ⇒ i ≠≠ index));
5    };
6
7    return <button onClick={() ⇒ removeTodo(0)}>Remove</button>;
8  }
```

# Batching State Updates

## ❖ Automatic Batching

➢ React groups multiple state updates in one render for performance.

```
1  function Counter() {
2    const [count, setCount] = useState(0);
3    const increment = () ⇒ {
4      setCount(count + 1);
5      setCount(count + 1); // Only increments once
6    };
7    return <button onClick={increment}>{count}</button>;
8  }
```

# Updater Functions

## ❖ Functional Updates

➢ Pass a function to the setter to update state based on its previous value.

```
1  import { useState } from 'react';
2
3  export default function UpdaterFunctions() {
4    const [count, setCount] = useState(0);
5    const increment = () ⇒ setCount((prev) ⇒ prev + 1);
6    return <button onClick={increment}>{count}</button>;
7  }
```

# useEffect and State

## ❖ Side Effects with State

useEffect runs after state updates, useful for syncing state with external systems.

```
1  const [count, setCount] = useState(0);
2   useEffect(() => {
3     document.title = `Count: ${count}`;
4   }, [count]);
5
6   return <button onClick={() => setCount(count + 1)}>{count}</button>;
```

# Lifting State Up

## ❖ Share State Between Components

- ➢ Move state to a common parent to share it with child components via props

```
1  export default function LiftingStateUp() {
2    const [count, setCount] = useState(0);
3
4    const handleIncrement = () ⇒ setCount(count + 1);
5    const handleDecrement = () ⇒ setCount(count - 1);
6
7    return (
8      <div>
9        <h1>Counter App</h1>
10       <Label text={count} />
11       <Counter count={count} onIncrement={handleIncrement} onDecrement={handleDecrement} />
12     </div>
13   );
14 }
```

# Prop Drilling

## ❖ Passing State Down

- ➢ State is passed through multiple components, leading to complex prop chains.

```
1  export default function PropDrilling() {
2    const [theme, setTheme] = useState('light');
3    return <Middle theme={theme} />;
4  }
5
6  function Middle({ theme }) {
7    return <Child theme={theme} />;
8  }
9
10 function Child({ theme }) {
11   return <div>{theme}</div>;
12 }
```

# Sharing State with Context

❖ **Global State**

**createContext** and **useContext** provide state to components without prop drilling

```jsx
import React, { createContext, useContext, useState } from 'react';
const ThemeContext = createContext();

export default function ContextAPI() {
  const [theme, setTheme] = useState('light');
  return (
    <div>
      <h5>Context API Example</h5>
      <ThemeContext.Provider value={theme}>
        <button onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}>Toggle Theme</button>
        <Child />
      </ThemeContext.Provider>
    </div>
  );
}

function Child() {
  const theme = useContext(ThemeContext);
  return <div>{theme}</div>;
}
```

# Complex State Logic

## ❖ useReducer Hook

**useReducer** manages state with a reducer function, ideal for complex updates.

```jsx
import { useReducer } from 'react';

export default function UseReducerHook() {
  const [state, dispatch] = useReducer(
    (state, action) => {
      switch (action.type) {
        case 'increment':
          return { count: state.count + 1 };
        case 'decrement':
          return { count: state.count - 1 };
        case 'reset':
          return { count: 0 };
        default:
          return state;
      }
    },
    { count: 0 },
  );
  return (
    <div>
      <p>{state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>
      <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
    </div>
  );
}

```

# Combining useState and useReducer

## ❖ Flexible State Management

Use useState for simple state, useReducer for complex logic in the same component.

```jsx
export default function CombineStateAndReducer() {
  const [name, setName] = useState('');

  const [todos, dispatch] = useReducer((state, action) => {
    if (action.type === 'add') {
      return [...state, action.payload];
    }
    if (action.type === 'remove') {
      return state.filter((todo) => todo !== action.payload);
    }
  }, []);

  return (
    <>
      <div>
        <input value={name} onChange={(e) => setName(e.target.value)} />
        <button onClick={() => dispatch({ type: 'add', payload: name })}>Add</button>
      </div>
      <div>
        {todos.map((todo, index) => (
          <div key={index}>
            {todo}
            <button onClick={() => dispatch({ type: 'remove', payload: todo })}>Remove</button>
          </div>
        ))}
      </div>
    </>
  );
}
```

# State in Class Components

## ❖ Legacy State Management

Class components use **this.state** and **this.setState** for state, less common today.

```
1  import React from 'react';
2
3  export default class Counter extends React.Component {
4    state = { count: 0 };
5
6    increment = () ⇒ this.setState({ count: this.state.count + 1 });
7
8    render() {
9      return <button onClick={this.increment}>{this.state.count}</button>;
10   }
11 }
```

# Component tutorials

1. Create CheckBox component
2. Create LikeButton component
3. Create Picture Viewer App
4. Create ToDo App