Module 7:
Advanced ReactJS

Dr. L.M. Jenila Livingston

Professor

VIT Chennai

Overview

- Dynamic Data
- Lists in JSX
- React Dataflow:
  - React Props
    - Props with a single value
    - Props with multiple values
    - Props restructuring
    - React Props Validation
  - React State
- React Hooks
- Passing Form Values – useRef(), useState()
- Styling React
- Routing
- Deploying React
- Case studies for building dynamic Applications

Display dynamic data

```
function RecipeTitle() {
    const title = 'Hello World';
    return (
        <h2>{ title }</h2>
    )
};
export default RecipeTitle;
```

- The constant title is assigned the string 'Hello World'.
- {title} is used to embed JavaScript expressions within HTML-like code
- {title} dynamically inserts the value of title into the rendered output

Lists in JSX – map(), key

```
function CarList() {
  const cars = [
    { id: 1, name: "Toyota" },
    { id: 2, name: "Honda" },
    { id: 3, name: "Ford" },
    { id: 4, name: "BMW" },
    { id: 5, name: "Tesla" }
  ];

  return (
    <div>
      <h2>Car Brands</h2>
      <ul>
        {cars.map((car) => (
          <li key={car.id}>{car.name}</li>
        ))}
      </ul>
    </div>
  );
}

export default CarList;
```

- The **key prop** is **required** when rendering a list in React.
- The **map()** function iterates over cars array.
- Each item in the array is displayed as an <li>.
- The key attribute is set using {car.id}
- The value (car brand name) attribute is set using {car.name}

```
Car Brands
- Toyota
- Honda
- Ford
- BMW
- Tesla
```

Lists in JSX – if no unique key

Use **.map()** function to render lists dynamically

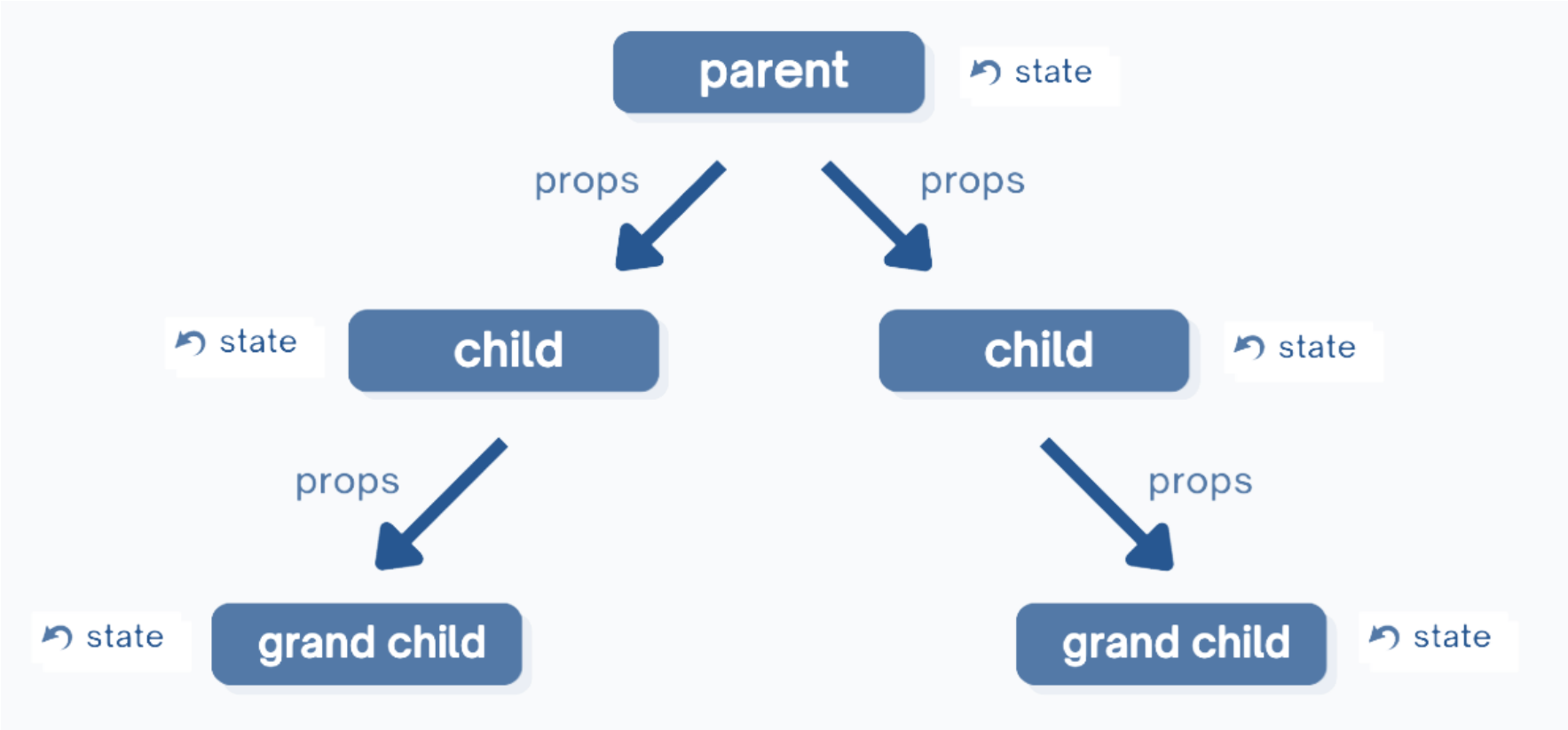
```
function App() {
  const fruits = ["Apple", "Banana", "Orange", "Mango"];
  return (
    <ul>
      {fruits.map((fruit, index) => (
        <li key={index}> {fruit} </li>
      ))}
    </ul>
  );
}

export default App;
```

- Use .map() to loop through the fruits array
- **index** → The position of the item in the array
- **fruit** → The current item

React Dataflow

- React follows a **one-way data flow** (from parent to child components), which makes the application predictable and easier to debug.
  - React **Props**
  - React **State** (Hook)

Props Vs State



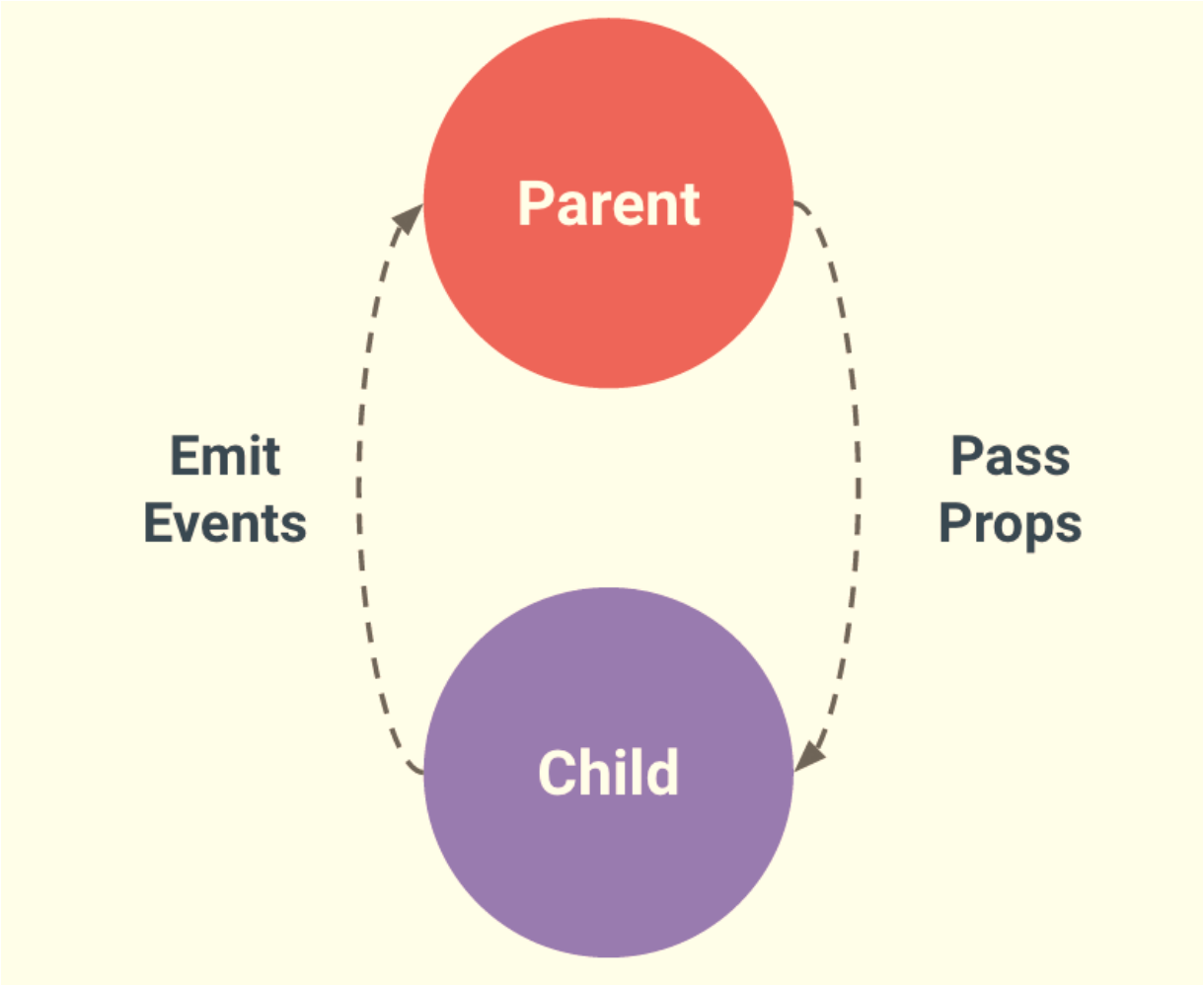
React Props

- Props (short for **properties**) allow us to **pass data between components**.
- They make components reusable and dynamic.
- Props are read-only.
- Used to display dynamic data contained inside JavaScript, use the syntax { }, sometimes called ***handlebars***

React Props

- Props with a single value
- Props with multiple values
- Props restructuring
  - single and multiple values
  - default props
- React Props Validation

React Props



Props with a single value

- Data from a parent component to child component

Child Component

```
// RecipeTitle.js
function RecipeTitle(props) {
    return <h2>{props.title}</h2>;  // Accessing props.title
}

export default RecipeTitle;
```

Parent Component

```
// App.js
import RecipeTitle from './RecipeTitle';

function App() {
    return <RecipeTitle title="Delicious Pancakes" />;
}

export default App;
```

Props with multiple values

Child Component

```
// RecipeTitle.js
function RecipeTitle(props) {
    return (
        <div>
            <h2>{props.title}</h2>
            <p>Author: {props.author}</p>
            <p>Servings: {props.servings}</p>
        </div>
    );
}

export default RecipeTitle;
```

Parent Component

```
// App.js
import RecipeTitle from './RecipeTitle';

function App() {
    return (
        <RecipeTitle
            title="Delicious Pancakes"
            author="John Doe"
            servings={4}
        />
    );
}

export default App;
```

Props - Restructuring

Child Component

```
function RecipeTitle({ title }) {
    return <h2>{title}</h2>;
};

export default RecipeTitle;
```

Parent Component

```
import RecipeTitle from './RecipeTitle';

function App() {
    return <RecipeTitle title="Delicious Pancakes" />;
}

export default App;
```

Props – Restructuring: Default prop

defaultProps is a feature in React that allows you to **set default values** for props in case they are not passed from the parent component.

Method1:
App.js (child component)

```
function App({ name = "Guest" }) {
  return (
    <>
      <h1>Hello, {name}!</h1>
      <div>Current Time: {new Date().toLocaleString()}</div>
    </>
  );
}

export default App;
```

- If no name is passed, it will display: **Hello, Guest!**
- new Date().toLocaleString() converts the timestamp into a human-readable format based on the user's locale.

```
Hello, Jenila!
Current Time: 3/1/2025, 6:45:23 PM
```

Props – Restructuring: Default prop

Method2:

App.js (child component)

```
import React from "react";

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

// Set a default prop value
Greeting.defaultProps = {
  name: "Guest",
};

export default Greeting;
```

Passing User name with Props (index.js to App.js)

Index.js: (parent component)

```
import React from "react";
import ReactDOM from 'react-dom/client';
import App from "./App";

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

root.render(
  <React.StrictMode>
    <App name="Jenila" />
  </React.StrictMode>
);
```

Props Restructuring - with multiple Values

Child Component

```
// RecipeTitle.js
function RecipeTitle({ title, author, servings })
    return (
        <div>
            <h2>{title}</h2>
            <p>Author: {author}</p>
            <p>Servings: {servings}</p>
        </div>
    );
}

export default RecipeTitle;
```

Parent Component

```
// App.js
import RecipeTitle from './RecipeTitle';

function App() {
    return (
        <RecipeTitle
            title="Delicious Pancakes"
            author="John Doe"
            servings={4}
        />
    );
}

export default App;
```

React Props Validation

- **PropTypes** ensure proper data type usage.
- Helps catch errors during development.
- ***npm install prop-types*** (if not already installed)
- **Import PropTypes from 'prop-types';**

Syntax:

```
ComponentName.propTypes = {
  propName: PropTypes.type.isRequired, // Marks as required
  propName: PropTypes.type,            // Optional prop
};
```

Example:

```
UserProfile.propTypes = {
  name: PropTypes.string.isRequired, // Required string prop
  age: PropTypes.number,             // Optional number prop
};
```

If name is missing, React will show a warning message

Validation Types

Prop Type	Description	Example
`PropTypes.string`	Must be a string	`name: PropTypes.string`
`PropTypes.number`	Must be a number	`age: PropTypes.number`
`PropTypes.bool`	Must be a boolean	`isActive: PropTypes.bool`
`PropTypes.array`	Must be an array	`items: PropTypes.array`
`PropTypes.object`	Must be an object	`data: PropTypes.object`
`PropTypes.func`	Must be a function	`onClick: PropTypes.func`

Feature	Syntax	Purpose
Required Prop	`.isRequired`	Makes a prop mandatory
Restrict Values	`oneOf([...])`	Limits prop to specific values
Multiple Types	`oneOfType([...])`	Allows multiple data types
Typed Array	`arrayOf(type)`	Ensures array contains specific types

React Props Validation

```
import PropTypes from 'prop-types';

function User({ age }) {
  return <p>Age: {age}</p>;
}

User.propTypes = { age: PropTypes.number.isRequired };
```

React Props Validation

```
import PropTypes from 'prop-types';

function Child({ message }) {
  return <h2>{message}</h2>;
}

//PropTypes Validation to ensure message is a requires string
Child.propTypes = { message: PropTypes.string.isRequired };

export default Child;
```

```
import React from 'react';
import Child from './Child';

function Parent() {
    return <Child message="Hello, React!" />;
}

export default Parent;
```

React Props Validation

```
import React from "react";
import PropTypes from "prop-types";

const UserProfile = ({ name, age }) => (
  <div>
    <h2>Name: {name}</h2>
    <p>Age: {age}</p>
  </div>
);

UserProfile.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number,
};

export default UserProfile;
```

```
import React from "react";
import UserProfile from "./UserProfile";

const ParentComponent = () => <UserProfile name="John Doe" age={25} />;

export default ParentComponent;
```

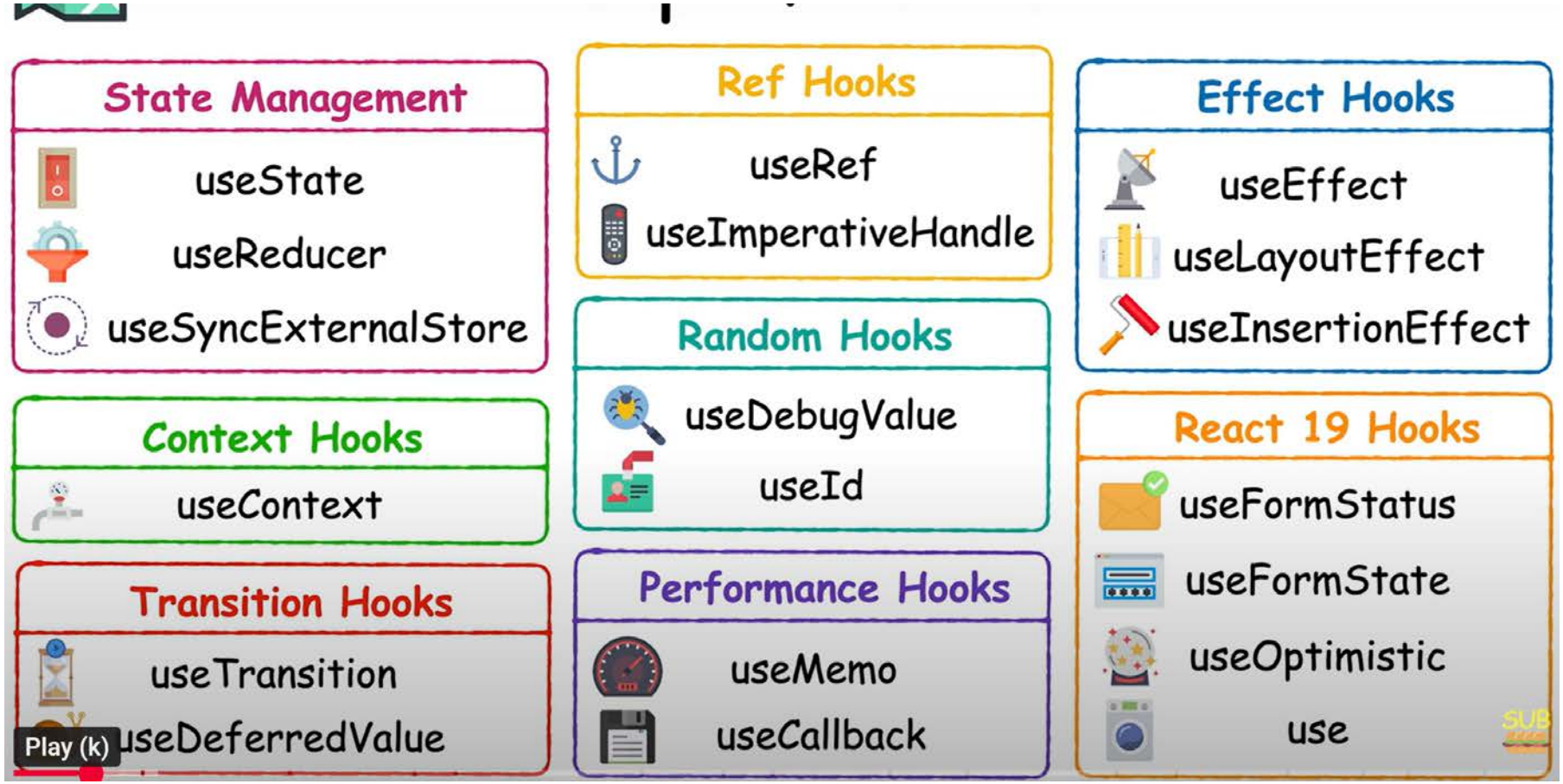
React Hooks

- React Hooks allow **functional components** to manage state and other react features without needing class components.
- Hooks allow us to "hook" into React features such as state and lifecycle methods.
- **Note:** Hooks will not work in React **class** components.

Hook Rules

- You must import Hooks from react
- Hooks can only be called inside React function components.
- Hooks can only be called at the top level of a component.
- Hooks cannot be conditional

Map of Hooks


State Management
useState
useReducer
useSyncExternalStore
Ref Hooks
useRef
useImperativeHandle
Effect Hooks
useEffect
useLayoutEffect
useInsertionEffect
Random Hooks
useDebugValue
useId
Context Hooks
useContext
React 19 Hooks
useFormStatus
useFormState
useOptimistic
use
Transition Hooks
useTransition
useDeferredValue
Performance Hooks
useMemo
useCallback
Play (k)

Hooks

1. State hooks:
    - **useState():** Simple state logic
    - **useReducer()**: Complex state logic
2. Effect hooks
    - **useEffect()**
3. Ref hooks
    - **useRef()**
4. Context hooks
    - **useContext()**

3 Major Hooks

1. State Hooks

Use to manage component state

- **useState()**: Manages Local State
- **useReducer()**: Manages Complex State Logic

React State

- State is used to store component-specific data.
- It enables dynamic updates without page reloads.
- State should not be directly modified; always use setState() in class components or **useState()** in functional components.

```
const [val, setVal] = useState(100);
```

The current value (State **variable**)

A setter function to change the value

The initial value to use

Functional Steps: useState()

- **Step 1:** Import useState hook from react.

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

- **Step 2:** Define a **functional component** Counter that maintains a **state variable** age using **useState**(42).

```
function Counter() {
    const [age, setAge] = useState(42);
```

- **Step 3: Display** the state variable - current age value.

```
<h2>Age: {age}</h2>
```

- **Step 4:** Add a button that, when clicked, triggers **setAge**(age + 1), updating the state.

```
<button onClick={() => setAge(age + 1)}>Increment</button>
```

useState()

```
const [age, setAge] = useState(42)
```

{/*Function is defined onClick*/}

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

function Counter() {
    const [age, setAge] = useState(42);

    return (
        <div>
            <h2>Age: {age}</h2>
            <button onClick={() => setAge(age + 1)}>Increment</button>
        </div>
    );
}

export default Counter;
```

useState()

{/*Function is called onClick*/}

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

function Counter() {
  const [age, setAge] = useState(42);

  // Function to handle age increment
  function incrementAge() {
    setAge(age + 1);
  }

  return (
    <div>
      <h2>Age: {age}</h2>
      <button onClick={incrementAge}>Increment</button>
    </div>
  );
}

export default Counter;
```

Use {} for calling a function
{incrementAge}

useState() Counters

- Increment/Decrement

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

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

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);

  return (
    <div>
      <button onClick={decrement}>-</button>
      {count}
      <button onClick={increment}>+</button>
    </div>
  );
}

export default Counter;
```

useState()

```
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>
    </>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FavoriteColor />);
```

useReducer()

- The useReducer Hook accepts two arguments.

Const count, dispatch = **useReducer(reducer, 0)**

- The useReducer Hook returns the current state (value) and a **dispatch** method to update the state
- The **reducer** function contains your custom state logic
- 0 -> initial value
- Define a reducer function **(parameters: state, action)** that takes state and action and updates the state accordingly.
- triggers **dispatch,** which calls the reducer function and updates the state.

Functional Steps: useReducer()

- **Step 1:** Import useReducer hook to manage the component's state.

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

- **Step 2:** Define a **reducer function** **(parameters: state, action)** that takes state and action and updates the state accordingly.

```
const reducer = (state, action) => {
```

- **Step 3:** Use **useReducer(reducer, 0)**, where 0 is the initial state.
  - It returns **count** (current state) and **dispatch** (to update state).

```
const [count, dispatch] = useReducer(reducer, 0);
```

- **Step 4:** Display the current **count** value.

```
<h2>Count: {count}</h2>
```

- **Step 5:** The button triggers **dispatch**("increment"), which calls the reducer function and increases the count.

```
<button onClick={() => dispatch("increment")}>Increment</button>
```

useReducer()

```
import React, { useReducer } from "react";

// Reducer function
const reducer = (state, action) => {
    switch (action) {
        case "increment":
            return state + 1;
        default:
            return state;
    }
};

function Counter() {
    // useReducer takes the reducer function and initial state
    const [count, dispatch] = useReducer(reducer, 0);

    return (
        <div>
            <h2>Count: {count}</h2>
            {/* Calls reducer with an action */}
            <button onClick={() => dispatch("increment")}>Increment</button>
        </div>
    );
}

export default Counter;
```

useState Vs useReducer

Feature	`useState`	`useReducer`
Usage	Simple state management	Complex state logic
Syntax	`const [state, setState] = useState(initialState);`	`const [state, dispatch] = useReducer(reducer, initialState);`
State Type	Typically used for primitive or small state values	Used for objects or multiple state updates
State Updates	Directly updates using `setState(newValue)`	Uses `dispatch(action)` to update state through a reducer function

2. useEffect

- Perform **side effects** in your components
- Some examples of side effects are: fetching data, directly updating the DOM, and timers.
- Runs code after render (API calls, subscriptions, etc.)
- Has two arguments (second argument is optional): useEffect(<function>, <dependency array>)

```
useEffect(() => {
  incrementCount();
}, [count]);
```

Empty dependency array `[]` → Runs only on mount.

Functional Steps: useEffect()

- **Step 1:** Import useState and useEffect from React.

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

- **Step 2:** Define a **state variable** (count) using useState(0)

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

- **Step 3:** Use useEffect() to execute code to change the count.

```
useEffect(() => {
  setTimeout(() => {
    setCount((count) => count + 1);
  }, 1000);
});
```

- **Step 4:** Display the current count vaue.

```
<h1>I have rendered {count} times!</h1>;
```

useEffect()

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

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

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  });

  return <h1>I have rendered {count} times!</h1>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Timer />);
```

useEffect()

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

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

  // Separated function to handle count increment
  const incrementCount = () => {
    setTimeout(() => {
      setCount((prevCount) => prevCount + 1);
    }, 1000);
  };

  useEffect(() => {
    incrementCount();
  }, [count]); // Dependency on count ensures it updates every second

  return <h1>I have rendered {count} times!</h1>;
}

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<Timer />);
```

3. useRef()

- **Access DOM Elements / Persist Values** (e.g., focusing an input field)
- **Storing Mutable Values** (without re-rendering)
- **Keeping Previous State Values**

useRef()

- Remember!
  - const **inputRef** = **useRef**(0);
  - **current** keyword: You can access the current value of that ref through the `inputRef.current` property. This value is intentionally mutable.

  - **ref attribute** in the input field.

```
<input ref={inputRef} type="text" placeholder="Enter text" />
```

useRef()

- **Syntax**

```
const reference = useRef(initialValue);
```

Eg: const reference = useRef(0); ***//***creates an object: { current: 0 }

```
// Accessing the reference value
const referenceValue = reference.current;
}

// Updating reference value
reference.current = reference.current + 1;
```

Note: The **.current** keyword in **useRef hook** is used to **access and modify** the value stored inside a ref object.

Functional Steps: useRef()

- **Step 1:** Import useRef from React.

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

- **Step 2:** Create a reference using useRef().

```
const inputRef = useRef(null);
```

- **Step 3:** Attach useRef to an input element

```
<input ref={inputRef} type="text" placeholder="Enter text" />
```

- **Step 4:** Use the reference to manipulate the DOM element

```
const focusInput = () => {
  inputRef.current.focus();
};
```

- **Step 5:** Add a button to trigger focusInput()

```
<button onClick={focusInput}>Focus Input</button>
```

useRef()
DOM: focusing Input field

```react/usereffexample.js
import React, { useRef } from "react";

function UseRefExample() {
  // Step 2: Create a reference
  const inputRef = useRef(null);

  // Step 4: Function to focus the input field
  const focusInput = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      {/* Step 3: Attach useRef to input */}
      <input ref={inputRef} type="text" placeholder="Type something..." />

      {/* Step 5: Button to trigger focus */}
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
}

export default UseRefExample;
```

- The useRef() hook creates a reference (inputRef).
- This reference is attached to the <input> element using **Ref attribute**.
- Clicking the button calls focusInput(), which focuses the input field. **(inutRef.current.focus())**

Type something... Focus Input

useRef()
Store mutable values

```
import React, { useRef } from "react";

function Counter() {
  const countRef = useRef(0);

  function increment() {
    countRef.current += 1;
    alert("Current Count: " + countRef.current);
  }

  return (
    <div>
      <h2>Click the button to see the count updates</h2>
      <button onClick={increment}>Increment</button>
    </div>
  );
}

export default Counter;
```

```
import React, { useState, useEffect, useRef } from "react";

function PreviousValue() {
  const [count, setCount] = useState(0);
  const prevCount = useRef(0);

  useEffect(() => {
    prevCount.current = count;
  }, [count]);

  return (
    <div>
      <h2>Current Count: {count}</h2>
      <h2>Previous Count: {prevCount.current}</h2>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default PreviousValue;
```

useRef()
Keep previous state values

Current Count: 2
Previous Count: 1
Increment

4. Using createContext()

- Used for **global state management** (like a lightweight Redux alternative)
- Share state between Components

```
const ThemeContext = createContext();
const theme = useContext(ThemeContext);
```

1 Create your context:

```
const AppContext = createContext()
```

3 Put data on value prop:

```
<AppContext.Provider value="👋">
```

2 Wrap Provider component:

```
<AppContext.Provider>
  <App />
</AppContext.Provider>
```

4 Get data with useContext

```
function Title() {
  const text = useContext(AppContext)
  return <h1>{text}</h1>
}
```

Fuctional Steps: createContext()

- **Step 1:**Import createContext and useContext from react.

```
import React, { createContext, useContext } from "react";
```

- **Step 2:** Create a new context using createContext().

```
const AppContext = createContext();
```

- **Step 3:** Wrap the App component inside <AppContext.Provider>.
- **Step 4:** Pass the value "👋 Hello from Context!" to the provider.

```
<AppContext.Provider value="👋 Hello from Context!">
    <Title />
</AppContext.Provider>
```

- **Step 5:** The Title component retrieves the value using useContext(AppContext) and displays it.

```
const text = useContext(AppContext);
return <h1>{text}</h1>;
```

createContext()

```
import React, { createContext, useContext } from "react";

// 1. Create your context
const AppContext = createContext();

function App() {
    return (
        // 2. Wrap Provider component & 3. Put data on value prop
        <AppContext.Provider value="👋 Hello from Context!">
            <Title />
        </AppContext.Provider>
    );
}

// 4. Get data with useContext
function Title() {
    const text = useContext(AppContext);
    return <h1>{text}</h1>;
}

export default App;
```

👋 Hello from Context!

Passing Form Values using hooks

- Get input values using **useRef()**
- Get input using **useState()**

Passing **Form Values** – **useRef()** - Functional Steps

1. Importing React and useRef Hook

```
import React, { useRef } from "react";
```

2. Creating Refs for Input Fields

```
const nameRef = useRef(null);
const ageRef = useRef(null);
```

3. Handling Form Submission (**handleSubmit** function)
   - Prevent default form submission behavior.
   - Access and display the input values using **Ref.current.value**

```
event.preventDefault();
alert(`Name: ${nameRef.current.value}, Age: ${ageRef.current.value}`);
```

4. **Form UI:**
   - **Call handleSubmit** when the form is submitted.
   - Attach **ref attributes** to input fields.

```
<input type="text" ref={nameRef} />
<input type="number" ref={ageRef} />
```

Passing form values – useRef()

```
import React, { useRef } from "react";

function FormWithRef() {
  const nameRef = useRef();
  const ageRef = useRef();

  function handleSubmit(event) {
    event.preventDefault();
    alert(`Name: ${nameRef.current.value}, Age: ${ageRef.current.value}`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" placeholder="Name" ref={nameRef} />
      <input type="number" placeholder="Age" ref={ageRef} />
      <button type="submit">Submit</button>
    </form>
  );
}

export default FormWithRef;
```

- To stop the default behavior of reload, use **event.preventDefault()** to ensure that the page **does not refresh**, allowing React to handle the form data dynamically.

Passing form values – useState()-Example 1

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

function FormComponent() {
  const [name, setName] = useState("");
  const [age, setAge] = useState("");

  function handleSubmit(event) {
    event.preventDefault();
    alert(`Name: ${name}, Age: ${age}`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" placeholder="Name" value={name}
        onChange={(event) => setName(event.target.value)} />
      <input type="number" placeholder="Age" value={age}
        onChange={(event) => setAge(event.target.value)} />
      <button type="submit">Submit</button>
    </form>
  );
}

export default FormComponent;
```

Passing **Form Values** – useState()

Avoid coding repetition: event.target.value

Use single variable: formData

Use single function: setFormData

1. Importing React and useState Hook
2. Defining the State with useState

```
const [formData, setFormData] = useState({ name: "", age: "" });
```

3. Handling Input Changes (**handleChange** function)
4. Handling Form Submission (handleSubmit function)
   - Prevent default form submission behavior.
   - display the input values
5. Form UI:
   - **Call** handleSubmit when the form is submitted
   - Attach **value** attributes to input fields. **value=**{**formData**.name}
   - Handling Input Changes (**onChange** event) – call **handleChange** function

Passing form values – useState() - Example 2

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

function FormComponent() {
  const [formData, setFormData] = useState({ name: "", age: "" });

  function handleChange(event) {
    setFormData({
      ...formData,
      [event.target.name]: event.target.value, // Dynamically updating state
    });
  }

  function handleSubmit(event) {
    event.preventDefault(); // Prevents page reload
    alert("Name: " + formData.name + ", Age: " + formData.age);
  }
```

```
  return (
    <form onSubmit={handleSubmit}>
      <label>
        Name: <input type="text" name="name" value={formData.name} onChange={handleChange} />
      </label>
      <br />
      <label>
        Age: <input type="number" name="age" value={formData.age} onChange={handleChange} />
      </label>
      <br />
      <button type="submit">Submit</button>
    </form>
  );
}

export default FormComponent;
```

Passing form values
useState Vs useRef

Comparison of Approaches

Method	Best For	Pros	Cons
useState	Small forms	Easy to implement	Re-renders on each update
useRef	Simple form submission	No re-renders	Cannot trigger UI updates

Styling React

Inline CSS
Internal CSS
External CSS

Inline CSS

```
import React from "react";

function StyledComponent() {
  const headingStyle = {
    color: "blue",
    fontSize: "24px",
    textAlign: "center",
  };

  return (
    <div>
      <h1 style={headingStyle}>Hello, React with Inline CSS!</h1>
      <p style={{ color: "green", fontWeight: "bold" }}>This is a styled paragraph.</p>
    </div>
  );
}

export default StyledComponent;
```

headingStyle is an object that stores CSS properties:
Better readability and reusability

In JSX, the **double curly braces ({{ }})** are used because:

- The **first pair** { ... } is for embedding JavaScript expressions inside JSX.
- The **second pair** { key: value } represents the JavaScript object containing the CSS inline styles.

Internal CSS

```
function StyledComponent() {
  return (
    <>
      <style>
        {`
          h1 {
            color: blue;
            font-size: 24px;
            text-align: center;
          }
          p {
            color: green;
            font-weight: bold;
          }
        `}
      </style>
      <h1>Hello, React with Internal CSS!</h1>
      <p>This is a styled paragraph.</p>
    </>
  );
}
```

External CSS

- CSS File (`styles.css`)

```
CSS

.container {
  text-align: center;
  margin-top: 20px;
}

.heading {
  color: blue;
  font-size: 24px;
}

.paragraph {
  color: green;
  font-weight: bold;
}
```

- Add **CSS styles** in App.css.
  - **import "./App.css";**

JSX Component (StyledComponent.js)

```
import React from "react";
import "./styles.css"; // Import external CSS

function StyledComponent() {
  return (
    <div className="container">
      <h1 className="heading">Hello, React with CSS Classes!</h1>
      <p className="paragraph">This is a styled paragraph.</p>
    </div>
  );
}

export default StyledComponent;
```

HTML	React JSX
`class="container"`	`className="container"`

External CSS – global style

- CSS File (`styles.css`)

```css
.container {
  text-align: center;
  margin-top: 20px;
}

.heading {
  color: blue;
  font-size: 24px;
}

.paragraph {
  color: green;
  font-weight: bold;
}
```

import in index.js

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";
import "./styles.css"; // Apply global styles

ReactDOM.render(<App />, document.getElementById("root"));
```

Routing

- React Router is a library used for navigation in React applications, allowing dynamic routing.
- You can set up routing in React **without modifying index.js** because React Router works inside your main App.js. You don't need to touch index.js unless you're configuring a higher-level setup
- Install router
  - npm install react-router-dom

- Example:

```
import { BrowserRouter as Router, Route } from 'react-router-dom';
<Router> <Routes>
  <Route path='/home' element={<Home />} />
</Routes></Router>
```

Routing

App.js (Main Component)

```
import React from "react";
import { BrowserRouter as Router, Route, Routes, Link } from "react-router-dom";
import Home from "./Home";
import About from "./About";

function App() {
  return (
    <Router>
      {/* Navigation Menu */}
      <nav>
        <Link to="/">Home</Link> | <Link to="/about">About</Link>
      </nav>

      {/* Route Configuration */}
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Router>
  );
}

export default App;
```

- **BrowserRouter (Router)**: Wraps the app to enable routing.
- **Link**: Navigates between pages without reloading.
- **Routes**: Contains multiple Route components.
- **Route**: Defines the path and which component to render.

Routing

- **Home.js**

```
import React from "react";
function Home() {
  return <h1>Welcome to the Home Page</h1>;
}
export default Home;
```

- **About.js**

```
import React from "react";
function About() {
  return <h1>About Us Page</h1>;
}
export default About;
```

Deploying React

- React apps can be deployed using:
  - GitHub Pages
  - Netlify
  - Vercel
  - Firebase Hosting
- Use `**npm run build**` to create a production build.

- Example:

Steps to deploy on GitHub Pages:

1. Install gh-pages: `**npm install gh-pages --save-dev**`
2. Add homepage in **package.json**
3. Run `**npm run deploy**`

Case Studies: Building Dynamic Applications

1. **E-commerce App:** Uses state for cart management, props for product details.
2. **Social Media App:** Uses hooks for fetching data, styled-components for theming.
3. **Weather App:** Uses API calls and React Router for navigation.

- Example:

Example - Fetching Weather Data:

```
fetch('https://api.weatherapi.com/weather')
.then(res => res.json())
.then(data => setWeather(data));
```

Thank You