Department: Computer Science & Engineering

Semester: Summer-2025

Course Title: Web Programming **Course Code:** CSE-479

Section: 02

Assignment: React.Js Assignment

Submitted To:

Md. Arman Hossain

Lecturer

Department of Computer Science and Engineering

Submitted By:

Parmita Hossain Simia
Id: 2022-3-60-2

Department of Computer Science and Engineering

Date of Report Submission: 6th August 2025

Core Concept 1: Components

Components are the building blocks of any React application. They are independent and reusable pieces of UI.

Think of them like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

Types of Components:

- **Functional Components:** Simple JavaScript functions. Preferred in modern React due to Hooks.
- **Class Components:** ES6 classes. Less common now but still seen in older codebases.

Core Concept 2: JSX (JavaScript XML)

JSX is a syntax extension for JavaScript. It looks like HTML but it's actually JavaScript.

It allows you to write HTML-like elements directly within your JavaScript code, making it easier to describe your UI.

Example:

```
const MyComponent = () => {
  return (
    <div className="text-center">
      <h1>Hello, JSX!</h1>
      <p>This is a paragraph.</p>
    </div>
  );
};
```

JSX gets compiled into regular JavaScript calls (`React.createElement()`).

Core Concept 4: State

State is data that a component can manage and change over time.

When a component's state changes, React re-renders the component to reflect the new state.

Using `useState` Hook:

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

const Counter = () => {
  const [count, setCount] = useState(0); // [currentState, updaterFunction]

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

`useState` is a Hook that lets you add React state to functional components.

Core Concept 3: Props (Properties)

Props are arguments passed into React components.

They are read-only and allow you to pass data from a parent component to a child component.

Example:

```
// Parent Component
const Greeting = () => {
  return <Welcome name="Alice" />;
};

// Child Component
const Welcome = (props) => {
  return <p>Hello, {props.name}!</p>;
};
```

Props help make components reusable and configurable.

React Hooks: useState & useEffect

Hooks are functions that let you "hook into" React features from your functional components.

`useState`:

Allows functional components to manage state. Returns a stateful value and a function to update it.

```
const [value, setValue] = useState(initialValue);
```

`useEffect`:

Performs side effects (e.g., data fetching, subscriptions) in functional components.

```
useEffect(() => {
  // Code to run after every render (or when dependencies change)
  console.log('Component mounted or updated!');

  return () => {
    // Cleanup function (optional, runs on unmount or before re-run)
    console.log('Cleanup!');
  };
}, [/* dependencies array */]);
```

Component Composition

React's power comes from composing small, independent components into larger ones.

This allows for highly modular, reusable, and maintainable codebases.

Example:

```
const Button = ({ children, onClick }) => (
  <button onClick={onClick}>{children}</button>
);

const Card = ({ title, content }) => (
  <div className="card">
    <h2>{title}</h2>
    <p>{content}</p>
    <Button onClick={() => alert('Clicked!')}>Learn More</Button>
  </div>
);

const App = () => (
  <Card title="React Basics" content="Learn about components and props." />
);
```

Components can accept other components as children (`props.children`).

Conditional Rendering

Conditional rendering in React allows you to render different elements or components based on certain conditions.

Common methods include `if/else`, ternary operators, and logical && operator.

Examples:

```
// Ternary Operator
{isLoggedIn ? <LogoutButton /> : <LoginButton />}

// Logical && Operator
{showMessage && <p>Welcome back!</p>}

// If/Else (outside JSX)
let button;
if (isLoggedIn) {
  button = <LogoutButton />;
} else {
  button = <LoginButton />;
}
```

This helps create dynamic UIs that respond to application state.

List Rendering

To render lists of items in React, you typically use the JavaScript `map()` array method.

Each item in the list should have a unique `key` prop to help React efficiently update and re-render list items.

Example:

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

const NumberList = () => {
  return (
    <ul>
      {numbers.map((number) =>
        <li key={number.toString()}>{number}</li>
      )}
    </ul>
  );
};
```

The `key` prop should be stable and unique among siblings.

Advanced Hook: useContext

The `useContext` Hook allows you to subscribe to React Context without nesting components.

Context provides a way to pass data through the component tree without having to pass props down manually at every level (prop drilling).

Usage:

```
// 1. Create Context
const ThemeContext = React.createContext('light');

// 2. Provide Context
const App = () => (
  <ThemeContext.Provider value="dark">
    <Toolbar />
  </ThemeContext.Provider>
);

// 3. Consume Context
const Toolbar = () => {
  const theme = useContext(ThemeContext);
  return <div className={`toolbar-${theme}`}>Current theme: {theme}</div>;
};
```

Useful for global data like themes, user authentication, or language preferences.

Component Composition

React's power comes from composing small, independent components into larger ones.

This allows for highly modular, reusable, and maintainable codebases.

Example:

```
const Button = ({ children, onClick }) => (
  <button onClick={onClick}>{children}</button>
);

const Card = ({ title, content }) => (
  <div className="card">
    <h2>{title}</h2>
    <p>{content}</p>
    <Button onClick={() => alert('Clicked!')}>Learn More</Button>
  </div>
);

const App = () => (
  <Card title="React Basics" content="Learn about components and props." />
);
```

Components can accept other components as children (`props.children`).

Conditional Rendering

Conditional rendering in React allows you to render different elements or components based on certain conditions.

Common methods include `if/else`, ternary operators, and logical && operator.

Examples:

```
// Ternary Operator
{isLoggedIn ? <LogoutButton /> : <LoginButton />}

// Logical && Operator
{showMessage && <p>Welcome back!</p>}

// If/Else (outside JSX)
let button;
if (isLoggedIn) {
  button = <LogoutButton />;
} else {
  button = <LoginButton />;
}
```

This helps create dynamic UIs that respond to application state.

List Rendering

To render lists of items in React, you typically use the JavaScript `map()` array method.

Each item in the list should have a unique `key` prop to help React efficiently update and re-render list items.

Example:

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

const NumberList = () => {
  return (
    <ul>
      {numbers.map((number) =>
        <li key={number.toString()}>{number}</li>
      )}
    </ul>
  );
};
```

The `key` prop should be stable and unique among siblings.

Advanced Hook: useContext

The `useContext` Hook allows you to subscribe to React Context without nesting components.

Context provides a way to pass data through the component tree without having to pass props down manually at every level (prop drilling).

Usage:

```
// 1. Create Context
const ThemeContext = React.createContext('light');

// 2. Provide Context
const App = () => (
  <ThemeContext.Provider value="dark">
    <Toolbar />
  </ThemeContext.Provider>
);

// 3. Consume Context
const Toolbar = () => {
  const theme = useContext(ThemeContext);
  return <div className={`toolbar-${theme}`}>Current theme: {theme}</div>;
};
```

Useful for global data like themes, user authentication, or language preferences.

Advanced Hook: useReducer

The `useReducer` Hook is an alternative to `useState` for more complex state logic.

It's often preferred when state logic involves multiple sub-values or when the next state depends on the previous one.

Usage:

```
const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </>
  );
};
```

It helps centralize state update logic.

Custom Hooks

Custom Hooks are JavaScript functions whose names start with "use" and that can call other Hooks.

They allow you to extract and reuse stateful logic from components, making your code cleaner and more organized.

Example: `useFetch`

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

const useFetch = (url) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchData = async () => {
      setLoading(true);
      const response = await fetch(url);
      const json = await response.json();
      setData(json);
      setLoading(false);
    };
    fetchData();
  }, [url]); // Re-run if URL changes

  return { data, loading };
};

// Usage in a component:
const MyComponent = () => {
  const { data, loading } = useFetch('https://api.example.com/data');
  if (loading) return <p>Loading...</p>;
  return <p>{data.message}</p>;
};
```

Styling in React

There are several ways to style your React applications:

- **Inline Styles:** JavaScript objects applied directly to elements.
- **CSS Stylesheets:** Traditional `.css` files imported into components.
- **CSS Modules:** Localized CSS to prevent global scope issues.
- **Styled Components:** CSS-in-JS library for writing CSS directly in JavaScript.
- **Utility-First CSS (e.g., Tailwind CSS):** Pre-defined utility classes for rapid UI development. (This presentation uses Tailwind!)

Choose the method that best fits your project's needs and team's preferences.

React Routing Basics

For multi-page applications, you need a way to navigate between different views without full page reloads.

While React itself doesn't include routing, libraries like `React Router` are widely used for client-side routing.

Core Concepts:

- **BrowserRouter:** Uses the HTML5 history API for clean URLs.
- **Routes & Route:** Define the paths and the components to render for those paths.
- **Link:** A component for navigation, preventing full page reloads.

```
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';

const App = () => (
  <BrowserRouter>
    <nav>
      <Link to="/">Home</Link> | <Link to="/about">About</Link>
    </nav>
    <Routes>
      <Route path="/" element={<HomePage />} />
      <Route path="/about" element={<AboutPage />} />
    </Routes>
  </BrowserRouter>
);
```

React Routing Basics

For multi-page applications, you need a way to navigate between different views without full page reloads.

While React itself doesn't include routing, libraries like `React Router` are widely used for client-side routing.

Core Concepts:

- **BrowserRouter:** Uses the HTML5 history API for clean URLs.
- **Routes & Route:** Define the paths and the components to render for those paths.
- **Link:** A component for navigation, preventing full page reloads.

```
import { BrowserRouter, Routes, Route, Link } from 'react-router-dom';

const App = () => (
  <BrowserRouter>
    <nav>
      <Link to="/">Home</Link> | <Link to="/about">About</Link>
    </nav>
    <Routes>
      <Route path="/" element={<HomePage />} />
      <Route path="/about" element={<AboutPage />} />
    </Routes>
  </BrowserRouter>
);
```

State Management Beyond useState

As applications grow, passing props down multiple levels (known as "prop drilling") can become cumbersome.

Solutions for managing global or complex state include:

- **Context API:** Built-in React feature for sharing state across the component tree without explicit prop passing.
- **Redux:** A predictable state container for JavaScript apps, offering a centralized store for application state.
- **Zustand:** A small, fast, and scalable bear-bones state-management solution using hooks.
- **Recoil / Jotai:** Atom-based state management libraries.

Choose a solution based on the complexity and scale of your application's state needs.

Performance Optimizations

Optimizing performance in React involves reducing unnecessary re-renders and computations.

- **`React.memo`:** A higher-order component that memoizes functional components, preventing re-renders if props haven't changed.
- **`useCallback`:** Memoizes functions, preventing them from being re-created on every render if their dependencies haven't changed. Useful for passing callbacks to memoized child components.
- **`useMemo`:** Memoizes expensive computations, re-running them only when dependencies change.
- **Lazy Loading:** Using `React.lazy` and `Suspense` to load components only when needed.

```
const MemoizedComponent = React.memo(MyComponent);
const handleClick = useCallback(() => { /* ... */ }, [deps]);
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Error Handling in React

Errors can occur anywhere in a React application. Handling them gracefully is crucial for user experience.

- **Error Boundaries:** React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the crashed component tree.
- **`try...catch`:** For imperative code like data fetching.
- **Asynchronous Error Handling:** Using `.catch()` with Promises or `try...catch` with `async/await`.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    console.error("ErrorBoundary caught an error:", error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
```

Conclusion & Next Steps

You've covered the fundamental and some advanced concepts of React.js!

To continue your React journey, consider exploring:

- **React Native:** For building mobile applications.
- **Next.js / Remix:** Frameworks for building full-stack React applications with server-side rendering.
- **Testing:** Learn about Jest, React Testing Library, and Cypress.
- **Deployment:** Deploying your React apps to platforms like Vercel, Netlify, or AWS.
- **Advanced State Management:** Dive deeper into Redux, Zustand, or Recoil.