PROGRAM-7

ReactJS – Render HTML, JSX, Components – function & Class

What is React?

- React is a JavaScript library for building user interfaces.
- React is used to build single-page applications.
- React is a tool for building UI components.
- React allows us to create reusable UI components.
- React is a front-end JavaScript library.
- React was developed by the Facebook Software Engineer Jordan Walke.
- React is also known as React.js or ReactJS.

How does React Work?

- React creates a VIRTUAL DOM in memory.
- Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.
- React only changes what needs to be changed!
- React finds out what changes have been made, and changes **only** what needs to be changed.

Advantages of React

- ✓ Faster UI updates with Virtual DOM
- ✓ Reusable and maintainable code through components.
- ✓ Large community and ecosystem (React Router, Redux, Next.js, etc.).
- ✓ Easy to learn if you know JavaScript.
- ✓ Works for both **web** (React) and **mobile apps** (React Native).

React vs Framework

- React is a library (not a full framework).
- It only handles the view layer (UI).
- For routing, state management, and APIs, we use additional tools.

Real-World Usage

- Used by big companies → Facebook, Instagram, Netflix, Uber, Airbnb, etc.
- Suitable for:
  - Dashboards
  - Social media apps
  - E-commerce websites
  - Real-time applications (chat, video apps)

a) AIM: Write a program to render HTML to a web page.

React Render HTML

- React's goal is in many ways to render HTML in a web page.
- React renders HTML to the web page via a container, and a function called **createRoot().**

The createRoot Function

- The createRoot function is located in the main.jsx file in the src folder, and is a built-in function that is used to create a root node for a React application
- The **createRoot()** function takes one argument, an HTML element.
- The purpose of the function is to define the HTML element where a React component should be displayed.

The Container

- React uses a container to render HTML in a web page.
- Typically, this container is a <div id="root"></div> element in the index.html file.

The render Method

- The render method defines what to render in the HTML container.
- The result is displayed in the <div id="root"> element.

Setting up a React Environment

- First, make sure you have Node.js installed. You can check by running this in your terminal:
  ```
  node -v
  ```
- If Node.js is installed, you will get a result with the version number:
  ```
  v22.15.0
  ```
- If not, you will need to install Node.js.

Build Tool (Vite)

Vite is a modern frontend build tool that helps developers set up and run web projects very quickly.

The word **"Vite"** comes from **French**, meaning **"fast"**.
Its main focus is **speed**: faster startup and faster builds compared to older tools.

Vite = A **fast tool** to build and run modern web projects.

- **Dev server**: Instant updates when coding.
- **Build tool**: Optimizes code for production.
- **Supports** React, Vue, Svelte, etc.
- Name "Vite" = French for **fast**.

Install a Build Tool (Vite)

- When you have Node.js installed, you can start creating a React application by choosing a build tool.
- Run this command to install Vite:

  ```
  npm install -g create-vite
  ```

- If the installation was a success, you will get a result like this:

  ```
  added 1 package in 649ms
  ```

Create a React Application

Run this command to create a React application named my-react-app:

```
npm create vite@latest my-react-app -- --template react
```

If you get this message, just press y and press Enter to continue:

```
Need to install the following packages

create-vite@6.5.0
Ok to proceed? (y)
```

If the creation was a success, you will get a result like this:

```
 > npx

> create-vite my-react-app --template react

o  Scaffolding project in C:\Users\stale\my-react-app...

—  Done.
```

Install Dependencies

- As the result above suggests, navigate to your new react application directory:

```
cd my-react-app
```

- And run this command to install dependencies:

```
npm install
```

Which will result in this:

```
added 154 packages, and audited 155 packages in 8s

33 packages are looking for funding

run `npm fund` for details

found 0 vulnerabilities
```

Run the React Application

- Now you are ready to run your first *real* React application!
- Run this command to run the React application my-react-app:

```
npm run dev
```

Which will result in this:

```
VITE  v6.3.5  ready in 217  ms


➜ Local: http://localhost:5173/
➜ Network:  use     --host  to      expose
➜ press h + enter to show help
```

Rendering:

In **React**, **rendering** means **taking React elements (JSX or components) and displaying them inside the browser's DOM**.

Step-by-Step Explanation of the Example

1. HTML Setup

```
<div id="root"></div>
```

- This empty <div> is where React will show all the content.
- Think of it as the **container** for React.

2. Import React and ReactDOM

- React → helps us create components using JSX.
- ReactDOM → is responsible for actually rendering (showing) those components inside the browser.

3. Create a Component

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

- This is a **function component**.
- It returns **JSX** (<h1>...</h1>), which looks like HTML but is actually JavaScript.

4. Render to DOM

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

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

- **document.getElementById("root")** → finds the <div id="root">.
- **ReactDOM.createRoot(...)** → tells React where to put the UI.
- **.render(<Hello />)** → renders the Hello component inside that div.

✅ So what happens?

- React takes <Hello /> → replaces it with <h1>Hello, React Rendering!</h1>
- Then inserts it inside <div id="root"></div>
- Finally, the user sees **Hello, React Rendering!** on the webpage.

👉 In short:

1. **Component** creates UI (JSX).
2. **ReactDOM** puts that UI inside the HTML page.

Look in the my-react-app directory, and you will find a src folder. Inside the src folder there is a file called **main.js**, open it and it will look like this:

Example:

The default content of the src/main.jsx file is replaced with the following content

main.jsx

```
import { createRoot } from 'react-dom/client'

createRoot(document.getElementById('root')).render(
  <h1>Hello React!</h1>
)
```

Output: **Hello React!**

Example

Display a paragraph inside the "root" element:

main.jsx

```
import { createRoot } from 'react-dom/client'

createRoot(document.getElementById('root')).render(
  <p>Welcome!</p>
)
```

Output:
Welcome

b. AIM: Write a program for writing markup with JSX

What is 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.

Coding JSX

- JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement() and/or appendChild() methods.
- JSX converts HTML tags into react elements.
- You are not required to use JSX, but JSX makes it easier to write React applications.

main.jsx

```
const myElement = <h1>I Love JSX!</h1>;

createRoot(document.getElementById('root')).render(
  myElement
);
```

JSX is an extension of the JavaScript language based on ES6, and is translated into regular JavaScript at runtime.

Writing markup with JSX

JSX is a syntax extension for JavaScript, used with React to describe what the UI should look like. This guide will explain why React mixes markup with rendering logic, how JSX differs from HTML, and how to display information using JSX.

How JSX is Different from HTML?

JSX may look similar to HTML, but there are key differences:

- **JavaScript Expressions:** JSX allows you to embed JavaScript expressions inside curly braces {}. This lets you dynamically render content based on your application state or props.
- **Attributes**: JSX attributes follow camelCase naming conventions instead of the kebab-case used in HTML. For example, class in HTML becomes className in JSX.
- **Self-Closing Tags**:Like XML, JSX requires that tags be properly closed. If an element doesn't have children, you must use a self-closing tag (e.g., <input /> instead of <input>).
- **Fragment Syntax**: JSX supports fragment syntax, allowing you to group multiple elements without adding extra nodes to the DOM. This can be done using <React.Fragment> or shorthand syntax <> </>.

Rules of JSX

1. Return a single root element

To return multiple elements from a component, enclose them within a single parent element.

```
<div>
  <h1>Geeks for Geeks</h1>
  <ul>
    ......
  </ul>
</div>
```

2. Close all the Tags

JSX mandates explicit closure of tags: self-closing tags such as `<img>` must be written as `<img />`, and wrapping tags like `<li>oranges` should be corrected to `<li>oranges</li>`.

```
<>
<img   src= "https://google.com/photos.jpg"
         alt =" google "
         class = "photo"
/>
   <ul>
         <li> Blue </li>
         <li> Green </li>
         <li>  Red </li>
  </ul>
</>
```

3. camelCase most of the things

JSX converts into JavaScript, where JSX attributes become keys in JavaScript objects. To handle this conversion seamlessly, React adopts camelCase for attribute names. For instance, attributes like stroke-width in JSX are transformed into strokeWidth in JavaScript. Similarly, to avoid conflicts with reserved words like class, React uses className instead, aligning with corresponding DOM properties for clarity and compatibility.

```
<img
       src= "https://imgur.com/"
       alt= "gfg"
       className = "photo"
/>
```

Step -by- step procedure:

Look in the my-react-app directory, and you will find a src folder. Inside the src folder there is a file called **main.js**, open it and replace with your required content.

❖ **Example 1**

Main.JSX:

```
const myElement = <h1>I Love JSX!</h1>;

createRoot(document.getElementById('root')).render(
  myElement
);
```

Output: I Love JSX!

Example 2: Expressions in JSX

With JSX you can write expressions inside curly braces { }.

The expression can be a React variable, or property, or any other valid JavaScript expression. JSX will execute the expression and return the result:

Execute the expression 5 + 5:

main.jsx

```
const myElement = <h1>React is {5 + 5} times better with JSX</h1>;
```

Output: React is 10 times better with JSX

Example 3: Inserting a Large Block of HTML

To write HTML on multiple lines, put the HTML inside parentheses:

Create a list with three list items:

main.jsx

```
const myElement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
  </ul>
);
```

Output:

- Apples
- Bananas
- Cherries

Example 3: One Top Level Element

The HTML code must be wrapped in *ONE* top level element.

So if you like to write *two* paragraphs, you must put them inside a parent element, like a div element.

Example

Wrap two paragraphs inside one DIV element

main.jsx

```
const myElement = (
  <div>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </div>
);
```

Output:

```
I am a paragraph.
I am a paragraph too.
```

- JSX will throw an error if the HTML is not correct, or if the HTML misses a parent element.
- Alternatively, you can use a "fragment" to wrap multiple lines. This will prevent unnecessarily adding extra nodes to the DOM.
- A fragment looks like an empty HTML tag: <></>.

Example
Wrap two paragraphs inside a fragment:

main.jsx

```
const myElement = (
  <>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </>
);
```

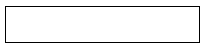
Output:

```
I am a paragraph.
I am a paragraph too.
```

Elements Must be Closed

JSX follows XML rules, and therefore HTML elements must be properly closed.
Example
Close empty elements with />
main.jsx
```
const myElement = <input type="text" />;
```

Output:



Note: JSX will throw an error if the HTML is not properly closed.

Attribute class = className

The class attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the class keyword is a reserved word in JavaScript, you are not allowed to use it in JSX.

Use attribute **className** instead.

JSX solved this by using className instead. When JSX is rendered, it translates className attributes into class attributes.

Example

Use attribute className instead of class in JSX:

main.jsx

```
const myElement = <h1 className="myclass">Hello World</h1>;
```

Comments in JSX

Comments in JSX are written with {/* */}

Example

main.jsx

```
const myElement = <h1>Hello {/* Wonderful */} World </h1>;
```

Output:

```
Hello World
```

c. AIM: Write a program for creating and nesting components (function and class).

React Components

What is a Component?

- A **component** is a reusable, independent piece of UI in React.
- Each component can represent part of a webpage (like a button, header, or form).
- Components make code **modular, reusable, and easier to maintain**.

Types of Components

React provides two main ways to create components:

a) Function Components

- A function that returns JSX.
- Simple and lightweight.
- Can accept **props** (input data) and display them.
- In modern React, hooks (useState, useEffect, etc.) make function components powerful.

Example:

```
function Greeting(props) {
  return <h2>Hello, {props.name}!</h2>;
}
```

b) Class Components

- Written as JavaScript **classes**.
- Must extend React.Component.
- Have a **render()** method that returns JSX.
- Can manage their own **state** and lifecycle methods.

Example:

```
class Welcome extends React.Component {
  render() {
    return <h1>Welcome, {this.props.name}!</h1>;
  }
}
```

3. Nesting Components

- Nesting means **using one component inside another**.
- This helps build complex UIs by combining small, reusable components.

Example:

```
function App() {
  return (
    <div>
      <Greeting name="Alice" />
      <Welcome name="Bob" />
    </div>
  );
}
```

Here:

- App is the parent component.
- Greeting and Welcome are **nested (child) components** inside App.

4. Props and State in Nesting

- **Props** (properties) → Passed from parent to child. They are read-only.
- **State** → Local to a component; used to manage changing data.
- Nested components usually **receive data via props** from parent components.

5. Why Use Nesting?

✅ Reusability → One component can be used in many places.
✅ Separation of Concerns → Each component handles its own logic/UI.
✅ Easy Maintenance → Small, independent units are easier to debug.
✅ Clear Hierarchy → Parent → Child → Sub-child structure makes UI structured.

📌 **Summary:**

- Function components → simple, use hooks.
- Class components → use render() and lifecycle methods.
- Nesting → placing one component inside another to build a hierarchy.

📌 **Example: Creating & Nesting Function and Class Components**

```
// Import React and ReactDOM
import React from "react";
import ReactDOM from "react-dom/client";
```

```
// ✅ Function Component

function Greeting(props) {
  return <h2>Hello, {props.name}!</h2>;
}

// ✅ Class Component

class Welcome extends React.Component {
  render() {
    return (
      <div>
        <h1>Welcome Component</h1>
        {/* Nesting function component inside class */}
        <Greeting name="Alice" />
        <Greeting name="Bob" />
      </div>
    );
  }
}

// ✅ Another Function Component nesting both
function App() {
  return (
    <div>
      <h1>App Component</h1>
      {/* Nesting class component */}
      <Welcome />
    </div>
  );
}

// Render into the root element
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<App />);
```

Output:

```
App Component
Welcome Component
Hello, Alice!
Hello, Bob!
```

🔍 **Explanation:**

1. **Greeting** → A simple **function component** with props.
2. **Welcome** → A **class component** that **nests Greeting** multiple times.
3. **App** → The main function component, which **nests Welcome**.
4. **root.render(<App />)** → Renders everything into the HTML page.