Module-18) React - Components, State, Props

Introduction to React.JS

Question 1: What is React.js? How is it different from other JavaScript frameworks and libraries?

Answer: React.js is an open-source JavaScript library developed by **Facebook** (now Meta) for building **user interfaces**, particularly **single-page applications** (SPAs) where data dynamically changes over time. It focuses on the **view layer** (V in MVC) and enables the development of reusable UI components.

React leverages a **declarative, component-based architecture**, which simplifies the process of developing complex UIs by breaking them down into encapsulated, maintainable pieces. One of its key innovations is the **Virtual DOM**—a lightweight in-memory representation of the actual DOM that allows React to perform efficient UI updates via **reconciliation** and **diffing algorithms**.

Core Concepts of React.js

1. **Component-Based Architecture**: Everything in React is a component—a self-contained, reusable piece of UI logic, defined either as a function or class.
2. **JSX (JavaScript XML)**: React uses JSX, a syntactic extension that allows mixing HTML-like tags with JavaScript logic, enhancing readability and expressiveness.
3. **Virtual DOM**: React maintains a virtual representation of the UI in memory and uses it to compute the most efficient way to update the browser's DOM.
4. **One-Way Data Binding**: Data flows unidirectionally from parent to child components, enhancing predictability and ease of debugging.
5. **Hooks API**: Introduced in React 16.8, hooks like useState, useEffect, and useContext allow functional components to manage state and side effects, eliminating the need for class components in most use cases.

How React Differs from Other JavaScript Frameworks and Libraries

Feature / Aspect	React.js	Angular (by Google)	Vue.js	jQuery

Type	Library (View Layer)	Full-fledged Framework	Progressive Framework	Library
Architecture	Component-based	Component-based + MVC-like	Component-based	Imperative DOM manipulation
DOM Handling	Virtual DOM	Real DOM with change detection	Virtual DOM	Real DOM
Data Binding	One-way	Two-way	Two-way (but can support one-way)	One-way (imperative)
Learning Curve	Moderate	Steep (complex API)	Moderate to Easy	Easy
Templating Syntax	JSX (HTML + JS)	HTML Templates + Directives	HTML Templates with Directives	HTML + jQuery Syntax
State Management	Internal state, Context API, Redux	Services, RxJS	Vuex	External plugins
Community and Ecosystem	Very large, backed by Meta	Large, backed by Google	Growing, community-driven	Legacy (declining in modern SPAs)

Key Differentiators of React

1. **Library vs. Framework**: Unlike Angular, which is a monolithic framework, React is focused strictly on the UI layer. This makes it more flexible and allows developers to integrate other libraries (like Redux, Zustand, or React Router) based on project needs.
2. **JSX vs. Templates**: React's JSX promotes JavaScript logic in templates, enabling greater flexibility and reducing context-switching compared to Angular or Vue's templating systems.
3. **Functional Paradigm with Hooks**: With the advent of hooks, React embraces functional programming paradigms more fully than other frameworks, reducing boilerplate and increasing reusability.
4. **Ecosystem and Customizability**: React's minimalistic core encourages the use of external libraries for routing, state management, and side effects, allowing for a highly customizable architecture. Angular, by contrast, offers everything out of the box, which may be overkill for smaller applications.

Question 2: Explain the core principles of React such as the virtual DOM and component-based architecture.

Answer: React is a popular JavaScript library used for building user interfaces, especially single-page applications. Its design is based on several powerful principles that make development efficient and the user interface highly responsive.

---

1. Component-Based Architecture

React follows a component-based approach, where the entire UI is divided into small, independent, and reusable pieces called *components*. Each component manages its own logic and rendering and can be reused throughout the application.

- This improves code modularity and maintainability.
- Components can be nested, managed, and tested independently.

Example: A Header, Footer, or UserCard can be separate components, reused across multiple pages.

---

2. Virtual DOM (Document Object Model)

The Virtual DOM is a lightweight copy of the actual DOM. When changes occur in a React application:

1. A virtual DOM tree is created.
2. React compares it with the previous version (using a "diffing" algorithm).
3. It calculates the minimal number of changes needed.
4. Only those specific changes are updated in the real DOM.

This process makes React faster and more efficient, as direct manipulation of the real DOM is costly in terms of performance.

---

3. Declarative UI

React uses a declarative approach for building user interfaces. Instead of telling the system how to do things step by step (imperative), you describe what you want the UI to look like, and React takes care of the rendering.

This makes the code easier to read and debug, especially in complex interfaces.

---

4. Unidirectional Data Flow

In React, data flows in a single direction — from parent components to child components via **props**. This one-way data flow makes the application more predictable and easier to debug.

If a child component needs to update data, it communicates with the parent via callbacks or state management libraries.

---

5. JSX (JavaScript XML)

React uses JSX, a syntax extension that allows writing HTML-like code within JavaScript. JSX makes the structure of UI components more readable and easier to write.

Example:

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

JSX is not mandatory, but it is widely used because it improves code clarity and developer experience.

Question 3: What are the advantages of using React.js in web development?

Answer: React.js is a powerful JavaScript library developed by Facebook, used primarily for building user interfaces. It has become one of the most popular tools for modern web development due to the following advantages:

---

1. Component-Based Architecture

- React promotes a modular structure where the UI is built using independent and reusable components.
- This makes code more **organized**, **maintainable**, and **scalable**.

---

2. Virtual DOM for Better Performance

- React uses a **Virtual DOM** to minimize direct updates to the real DOM.
- It compares changes using a “diffing” algorithm and only updates the necessary parts.
- This results in **faster rendering** and **better performance**.

---

3. Declarative Syntax

- Developers can describe **what** the UI should look like, and React handles **how** to update it.
- This leads to **cleaner code** and **easier debugging**.

---

4. Reusability and Maintainability

- Components can be reused in different parts of the application or across projects.
- This reduces duplication and improves **development speed**.

---

5. Strong Community and Ecosystem

- React has a large community, rich documentation, and thousands of third-party libraries.
- This makes it easier to find solutions, tools, and support.

---

6. JSX (JavaScript XML)

- React uses JSX, which allows HTML to be written directly within JavaScript.

- This makes UI code more **readable** and **intuitive** for developers.

---

7. Easy to Learn (for JavaScript Developers)

- Developers with basic JavaScript knowledge can quickly learn React.
- Its simplicity and clear structure make onboarding easier.

---

8. SEO-Friendly

- React supports server-side rendering (with tools like Next.js), which improves **SEO performance** for web applications.

---

9. Unidirectional Data Flow

- Data flows in a single direction, from parent to child.
- This structure helps in managing and debugging complex applications more easily.

---

10. Compatible with Other Technologies

- React can be integrated with other libraries or frameworks (like Redux, Express, or even legacy code).
- This flexibility makes it suitable for a wide range of projects.

JSX (JavaScript XML)

Question 1: What is JSX in React.js? Why is it used?

Answer: JSX stands for **JavaScript XML**. It is a **syntax extension** for JavaScript that is used in React to describe what the UI should look like. JSX allows developers to write HTML-like code directly inside JavaScript.

Example of JSX:

const element = <h1>Hello, React!</h1>;

This code looks like HTML, but it's actually JavaScript and gets compiled into React.createElement() calls behind the scenes.

Why JSX is Used in React.js:

1. Improves Readability

- JSX looks very similar to HTML, so it's easy for developers to understand and visualize the UI structure.

2. Combines Markup and Logic

- JSX allows developers to write both UI structure and JavaScript logic in the same place, which helps in building components faster.

3. Helps Catch Errors Early

- JSX code is checked at compile time, so errors can be identified early during development.

4. Supports JavaScript Expressions

- You can use JavaScript inside JSX using curly braces {}.

```
const name = "Jay";
const greeting = <h2>Hello, {name}</h2>;
```

5. Makes Code Declarative

- JSX supports React's declarative approach by clearly describing what the UI should look like.

6. Tooling and Ecosystem Support

- JSX works seamlessly with modern build tools like Babel and Webpack, which convert JSX into browser-compatible JavaScript.

Question 2: How is JSX different from regular JavaScript? Can you write JavaScript inside JSX?

Answer:

1. **What is JSX?**
   - JSX stands for **JavaScript XML**.
   - It allows writing **HTML-like code inside JavaScript**.
   - JSX is **syntactic sugar** for React.createElement() calls.
   - Helps visually structure UI within the JavaScript code.
2. **What is Regular JavaScript?**
   - Uses **standard syntax** for logic, functions, and data manipulation.
   - Does **not support HTML-like tags** like <div> or <h1> unless used as strings or in DOM manipulation.
3. **Main Differences:**
   - JSX mixes **UI layout** and **logic** in a more readable way.
   - Regular JavaScript focuses only on **logic** and **functionality**.
   - JSX must be **compiled (e.g., by Babel)** before running in the browser.
   - Regular JavaScript runs **directly in the browser**.
4. **Using JavaScript Inside JSX:**
   - You can use **JavaScript expressions** (but not full statements).
   - Wrap expressions in **curly braces {}** inside JSX.
5. **Examples:**

   -
     ```
     const name = "Jay";
     const element = <h1>Hello, {name}!</h1>;
     ```

   -
     ```
     const price = 50;
     const discount = 10;
     const total = <p>Total: ₹{price - discount}</p>;
     ```
6. **Limitations:**
   - Cannot use **statements** like if, for, or while **directly in JSX**.
   - Use such logic **outside the JSX** or with **ternary operators** inside JSX.

Question 3: Discuss the importance of using curly braces {} in JSX expressions.

Answer: In JSX, **curly braces {} are used to embed JavaScript expressions** inside the HTML-like syntax. This is one of the most powerful features of JSX and is essential for creating dynamic and interactive user interfaces in React.

Why are curly braces important in JSX?

1. To Insert JavaScript Expressions

Curly braces allow you to insert variables, perform calculations, or call functions directly within JSX.

Example:

```
const name = "Uttam";
const element = <h1>Hello, {name}!</h1>;
```

In this example, {name} inserts the value of the JavaScript variable into the rendered output.

---

2. To Make the UI Dynamic

Using {} enables you to display dynamic content based on data or logic. Without curly braces, JSX would treat the content as plain text.

Example:

```
const items = 3;
<p>You have {items * 2} points</p>  // Displays: You have 6 points
```

---

3. To Call Functions

You can use curly braces to call JavaScript functions and display their return values inside the UI.

Example:

```
function greet() {
  return "Welcome back!";
}
<h2>{greet()}</h2>
```

---

4. To Use Conditional Expressions

Although you can't use full if statements directly inside JSX, you can use **ternary operators** or **logical operators** within {} to conditionally render content.

Example:

```
const isLoggedIn = true;

<h3>{isLoggedIn ? "Logout" : "Login"}</h3>
```

Important Notes:

- **Only expressions** can go inside {}, not full statements like if, for, or while.
- You cannot use {} outside JSX. They are special only within the return part of a React component.

Components
(Functional & Class Components)

Question 1: What are components in React? Explain the difference between functional components and class components.

Answer: In React, **components are the building blocks** of the user interface. Each component is a **self-contained piece of UI** that can have its own structure, styling, and logic. Components allow developers to **split the UI into reusable, independent parts**, making the code more modular and easier to manage.

There are **two main types** of components in React:

1. Functional Components

Functional components are **simple JavaScript functions** that return JSX. They are also called **stateless components** (although with React Hooks, they can now manage state too).

Example:

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

Key Features:

- Defined as functions.
- Use props as an argument to access data.
- Can use **Hooks** (like useState, useEffect) to manage state and side effects.
- Easier to write and understand.
- Preferred in modern React development.

---

2. Class Components

Class components are defined using **ES6 classes**. They were used for components that needed to manage state or lifecycle methods before Hooks were introduced.

Example:

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

Key Features:

- Must extend React.Component.
- Use this.props and this.state to manage data.

- Require a render() method to return JSX.
- Can use lifecycle methods like componentDidMount(), componentDidUpdate(), etc.
- More verbose and complex than functional components.

Main Differences Between Functional and Class Components:

Feature	Functional Component	Class Component
Syntax Style	JavaScript function	ES6 class extending React.Component
this keyword	Not used	Required to access props and state
State Management	Uses Hooks like useState()	Uses this.state
Lifecycle Methods	Uses Hooks like useEffect()	Uses built-in lifecycle methods
Code Simplicity	More concise and readable	More boilerplate and complex
Modern Usage	Recommended in most new React projects	Mostly used in older codebases

Question 2: How do you pass data to a component using props?

Answer: In React, **props (short for "properties")** are used to pass data from one component to another, usually from a **parent component to a child component**. This allows components to be **dynamic and reusable**.

How Props Work:

1. **Pass data as attributes** when you use the component.
2. **Access the data** inside the child component using the props object.

Example:

Parent Component:

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

Here, the App component is passing a prop called name with the value "Alice" to the Greeting component.

Child Component:

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

Inside the Greeting component, the value of the name prop is accessed using props.name.

Using Destructuring (Optional but common):

Instead of using props.name, you can use **object destructuring** to make the code cleaner:

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

Why Use Props?

- To **customize** the content of a component.
- To **reuse** the same component with different data.
- To **separate concerns** between parent and child components.

Question 3: What is the role of render() in class components?

Answer: In React, the render() method is a **required function** in all **class components**. Its main role is to **describe what the UI should look like** for that component.

- The render() method returns **JSX**, which tells React what to display on the screen.
- It is called **automatically by React** when the component is first loaded and **whenever the component's state or props change**.
- It must return **a single parent element** (like a <div> or a fragment <>...</>).

Example:

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

In this example:

- The render() method returns an <h1> element that displays a greeting.
- this.props.name is used to insert dynamic data.

Why is render() Important?

- It defines the **visual output** of the component.
- React uses its return value to **update the DOM** efficiently using the Virtual DOM.
- Without render(), a class component **won’t show anything** on the screen.

Props and State

Question 1: What are props in React.js? How are props different from state?

Answer: In React.js, **props** (short for *properties*) are a way to **pass data from one component to another**, usually from a **parent component to a child component**. They are used to customize or configure a component's behavior or appearance.

What are Props?

- Props are **read-only**.
- Passed as **attributes** to components.
- Help in making components **reusable and dynamic**.

Example:

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

Here, name="Alice" is a prop passed to the Greeting component.

How are Props Different from State?

1. Purpose

- Props are used to pass data from a parent component to a child component.
- State is used to manage dynamic data within the component itself.

---

2. Mutability

- Props are immutable. Once passed, they cannot be changed by the receiving component.
- State is mutable. A component can change its own state using the setState function (in class components) or useState hook (in functional components).

---

3. Control

- Props are controlled by the parent component. The child has no control over the values it receives.
- State is controlled by the component where it is defined. It determines how the component behaves or renders.

---

4. Usage

- Props are used when you want to configure a component or reuse it with different data.
- State is used when you want the component to track information and react to user input, API calls, etc.

---

5. Lifecycle

- Props are set once when the component is rendered and may update only if the parent re-renders.
- State can change over time, often in response to events or user interactions.

---

6. Access

- Props are accessed via props.propertyName or by destructuring in function/class components.
- State is accessed via this.state and modified using this.setState() (class components) or useState() in functional components.

Question 2: Explain the concept of state in React and how it is used to manage component data.

Answer: In React, **state** refers to a **built-in object** that allows a component to **store and manage dynamic data** that can change over time. It helps React components keep track of information like user input, API responses, toggles, counters, or any data that affects what is displayed on the screen.

- State is **local** to a component and not accessible to other components (unless passed down as props).
- When state changes, React **automatically re-renders** the component to reflect the new data.
- State is mainly used when a component needs to **remember something between renders**.

Using State in Functional Components (with Hooks):

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

function Counter() {
  const [count, setCount] = useState(0);
// count is the state variable, setCount is the function to update
it

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click Me
      </button>
    </div>
  );
}
```

In this example:

- count is the state variable.
- setCount is the function used to update the state.
- When the button is clicked, count is updated and the component re-renders.

Using State in Class Components:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={this.increment}>Click Me</button>
      </div>
    );
  }
}
```

Why State Is Important:

- It makes components **interactive and dynamic**.
- Without state, components would be static and only display fixed content.
- It plays a crucial role in creating responsive and user-driven applications.

Question 3: Why is this.setState() used in class components, and how does it work?

Answer: this.setState() is used in class components**:**

1. **To Update Component State:**
   - this.setState() is used to **change the state** of a class component.

  - Directly modifying state (e.g., this.state.count = 1) does **not trigger re-rendering** and is considered bad practice.

2. **To Trigger a Re-render:**
   - When you call this.setState(), React automatically **re-renders** the component with the updated state.
   - This ensures the UI stays in sync with the underlying data.

3. **To Merge State Updates:**
   - Unlike replacing the whole state object, this.setState() performs a **shallow merge** with the existing state.
   - Only the specified keys are updated; other keys remain unchanged.

How does this.setState() work?

- **Syntax Example:**

  ```
  this.setState({ count: this.state.count + 1 });
  ```

- **Process:**
  1. You call this.setState() and pass it an object or function to update the state.
  2. React **merges the new state** with the current one.
  3. React then **re-renders the component** to reflect the new state in the UI.

Function Form of setState:

If the new state depends on the previous state, you should use a function:

```
this.setState((prevState) => ({
  count: prevState.count + 1
}));
```

This avoids bugs when multiple setState() calls happen close together.