**1. What is React?**

React is a **JavaScript library** created by Facebook to build **user interfaces** (UIs). It is especially useful for building **single-page applications (SPAs)** where the content updates dynamically without reloading the entire page.

Key Features:

* **Component-Based:** UI is divided into small, reusable pieces (components).
* **Virtual DOM:** React uses a lightweight in-memory representation of the DOM (Virtual DOM) to efficiently update only the changed parts.
* **Declarative:** Instead of describing how to do things step-by-step, React lets you focus on **what to render**.

**Example:**

function App() {

return <h1>Hello, React World!</h1>;

}

When rendered, React turns this JSX into the equivalent HTML: <h1>Hello, React World!</h1>.

**2. What are Components?**

Components are **reusable pieces of UI**. Think of them as JavaScript functions that return HTML.

Types of Components:

1. **Functional Components:** Written as plain JavaScript functions.
2. **Class Components:** Written as ES6 classes (less common today).

**Functional Component Example:**

function Greeting(props) {

return <h1>Welcome, {props.name}!</h1>;

}

**Class Component Example:**

class Greeting extends React.Component {

render() {

return <h1>Welcome, {this.props.name}!</h1>;

}

}

**3. JSX (JavaScript XML):**

JSX is **syntax sugar** for writing HTML-like code in JavaScript. It makes it easier to write UI directly in your JavaScript code.

**JSX Example:**

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

This JSX code is transformed into JavaScript like:

const element = React.createElement('h1', null, 'Hello, World!');

**4. useState for State Management**

In React, **state** represents the current data of a component. With the useState hook, functional components can now manage state.

**How useState Works:**

1. Import useState:
2. import React, { useState } from 'react';
3. Declare a state variable:
4. const [count, setCount] = useState(0);
   * count: Holds the state value.
   * setCount: Updates the state.

**Example:**

function Counter() {

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

return (

<div>

<p>Current count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

* Clicking the button increments the count value.

**5. Props (Passing Data Between Components)**

Props (short for **properties**) allow data to be passed **from a parent component to a child component**.

**Example:**

function Welcome(props) {

return <h1>Welcome, {props.name}!</h1>;

}

function App() {

return <Welcome name="Alice" />;

}

* The name prop is passed from App to Welcome.

**6. Lifting State Up**

When multiple components need to share the same data, you **"lift the state up"** to their nearest common parent.

**Example:**

function App() {

const [input, setInput] = useState("");

return (

<div>

<InputForm onChange={(value) => setInput(value)} />

<Display value={input} />

</div>

);

}

function InputForm({ onChange }) {

return <input onChange={(e) => onChange(e.target.value)} />;

}

function Display({ value }) {

return <p>{value}</p>;

}

Here:

* App manages the input state.
* InputForm updates it.
* Display shows it.

**7. Rendering Lists**

To render a list in React, use the **map() function**. Each item in the list should have a **unique key** to help React identify and update changes efficiently.

**Example:**

const numbers = [1, 2, 3];

const listItems = numbers.map((num) => <li key={num}>{num}</li>);

function App() {

return <ul>{listItems}</ul>;

}

**8. Keys in Lists**

Keys help React **track which items are changed, added, or removed**. Use a stable and unique identifier, like an id, as the key.

**Good Example:**

const todos = [{ id: 1, task: "Learn React" }, { id: 2, task: "Do Homework" }];

function TodoList() {

return (

<ul>

{todos.map((todo) => (

<li key={todo.id}>{todo.task}</li>

))}

</ul>

);

}

**9. Handling Events**

React events are similar to regular JavaScript events, but they use **camelCase** (e.g., onClick).

**Example:**

function ButtonClick() {

const handleClick = () => alert("Button was clicked!");

return <button onClick={handleClick}>Click Me</button>;

}

**10. Conditional Rendering**

Render components conditionally based on some logic.

**Example:**

function Greeting({ isLoggedIn }) {

return isLoggedIn ? <h1>Welcome Back!</h1> : <h1>Please Log In</h1>;

}

**11. Embedding map() Directly**

Instead of creating a separate variable for the mapped list, you can embed it directly into JSX.

**Example:**

function App() {

const numbers = [1, 2, 3];

return (

<ul>

{numbers.map((num) => (

<li key={num}>{num}</li>

))}

</ul>

);}

**12. Hover Effects with Tailwind CSS**

Tailwind simplifies adding hover effects with utility classes.

**Example:**

<button className="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">

Hover Me

</button>

**Key React Rules to Remember**

1. **Components should be small and focused on a single responsibility.**
2. **Use keys when rendering lists for efficient updates.**
3. **Avoid mutating state directly; always use the setter function (e.g., setState).**
4. **Keep your components pure (avoid side effects like fetching data directly).**
5. **Lift state up when multiple components need to share data.**

(**Key Notes for React Concepts**

**1. General React Concepts**

* React is a **library, not a framework**.
* Focuses on **building UIs** with components and manages **state** efficiently.
* Works with a **Virtual DOM**, ensuring fast UI updates.

**2. JSX**

* JSX is **syntax sugar** for combining HTML and JavaScript.
* Use curly braces {} to embed JavaScript expressions inside JSX.
* Always close tags, even for self-closing ones like <img />.

**3. Components**

* Components are **reusable pieces of UI**:
  + **Functional Components**: Use function to define, stateless by default but can use hooks.
  + **Class Components**: Use class syntax, have lifecycle methods (less common now).
* Component names must start with a capital letter.
* Use props to pass data into components.

**4. State Management**

* State is **mutable data** controlled within a component.
* Use useState hook in functional components:
* const [state, setState] = useState(initialValue);
* **Avoid direct state mutation** (e.g., state.value = newValue).

**5. Props**

* Props are **read-only** and passed from parent to child.
* Destructure props for cleaner code:
* function Welcome({ name }) {
* return <h1>Hello, {name}!</h1>;
* }

**6. Keys**

* Keys **must be unique among siblings** in a list.
* Use stable, unique IDs (avoid indexes if items can be reordered).
* React uses keys to identify and optimize updates for list items.

**7. Lists**

* Use .map() to render lists:
* const list = items.map((item) => <li key={item.id}>{item.name}</li>);
* Embed map() directly in JSX for shorter code.

**8. Lifting State Up**

* When data needs to be shared between components, move the state to their **closest common parent**.

**9. Events**

* Use camelCase for event handlers: onClick, onChange.
* Pass functions for event handling:
* <button onClick={() => doSomething()}>Click Me</button>

**10. Conditional Rendering**

* Use ternary operators or && for conditionally showing components:
* isLoggedIn ? <Welcome /> : <Login />;

**11. Reusability**

* Break large components into **smaller, reusable components**.
* Extract repeated logic into separate functions or hooks.

**12. Tailwind CSS Tips for React**

* Use Tailwind utility classes directly in className.
* Hover effects:
* className="bg-blue-500 hover:bg-blue-700"
* Combine conditionally applied classes using libraries like clsx or classnames.

**Best Practices**

1. Always use **unique keys** when rendering lists.
2. Keep components **pure** (don’t modify props or do side effects like fetch calls inside them).
3. Use **functional components** with hooks instead of class components.
4. Keep state localized unless needed globally.
5. **Break components logically**—one component for one task.
6. Avoid overusing inline functions for performance-sensitive areas (like lists).
7. **Comment complex logic** for clarity.

These concise key points complement your refresher notes, summarizing the most critical aspects to remember. Let me know if you’d like further clarification!

)

**React Hooks: useState and useEffect**

**1. useState Hook**

**What is useState?**

* A **React Hook** that lets you add **state** to functional components.
* It allows a component to remember values between renders (like a counter, input value, or fetched data).

**Syntax:**

const [state, setState] = useState(initialValue);

* **state:** Current value of the state.
* **setState:** Function used to update the state.
* **initialValue:** The initial value of the state (can be a string, number, boolean, object, array, etc.).

**Key Points:**

* Updating state using setState triggers a re-render of the component.
* React ensures state updates are **asynchronous**.
* Always use setState to update the state; directly modifying the state variable does **not work**.

**Examples:**

1. **Counter Example:**
2. import React, { useState } from "react";
3. function Counter() {
4. const [count, setCount] = useState(0);
5. const increment = () => {
6. setCount(count + 1); // Updates state
7. };
8. return (
9. <div>
10. <p>Count: {count}</p>
11. <button onClick={increment}>Increment</button>
12. </div>
13. );
14. }
15. **Managing Input State:**
16. const [inputValue, setInputValue] = useState("");
17. return (
18. <div>
19. <input
20. type="text"
21. value={inputValue}
22. onChange={(e) => setInputValue(e.target.value)}
23. />
24. <p>{inputValue}</p>
25. </div>
26. );

**2. useEffect Hook**

**What is useEffect?**

* A **React Hook** that lets you perform **side effects** in functional components.
* Side effects include things like fetching data, setting up subscriptions, or manually changing the DOM.

**Syntax:**

useEffect(() => {

// Effect logic here (e.g., API call, event listener setup)

return () => {

// Cleanup logic (optional, e.g., remove event listeners)

};

}, [dependencies]);

* **Effect Logic:** Code that runs when the component renders or updates.
* **Cleanup Logic:** Optional code to clean up resources (runs before the component unmounts or when dependencies change).
* **Dependencies Array:** Controls when the effect runs.

**Key Points:**

* By default, the effect runs **after every render**.
* Adding an **empty dependencies array ([])** makes the effect run **only once** (when the component mounts).
* If you specify dependencies, the effect re-runs **only when those dependencies change**.

**Examples:**

1. **Fetching Data:**
2. import React, { useState, useEffect } from "react";
3. function DataFetching() {
4. const [data, setData] = useState([]);
5. useEffect(() => {
6. const fetchData = async () => {
7. const response = await fetch("https://api.example.com/data");
8. const result = await response.json();
9. setData(result);
10. };
11. fetchData();
12. }, []); // Runs only on mount
13. return (
14. <div>
15. {data.map((item) => (
16. <p key={item.id}>{item.name}</p>
17. ))}
18. </div>
19. );
20. }
21. **Updating Title Dynamically:**
22. import React, { useState, useEffect } from "react";
23. function DynamicTitle() {
24. const [count, setCount] = useState(0);
25. useEffect(() => {
26. document.title = `Count: ${count}`; // Update the browser tab title
27. }, [count]); // Runs only when "count" changes
28. return (
29. <div>
30. <p>Count: {count}</p>
31. <button onClick={() => setCount(count + 1)}>Increment</button>
32. </div>
33. );
34. }

**Common Use Cases of useEffect:**

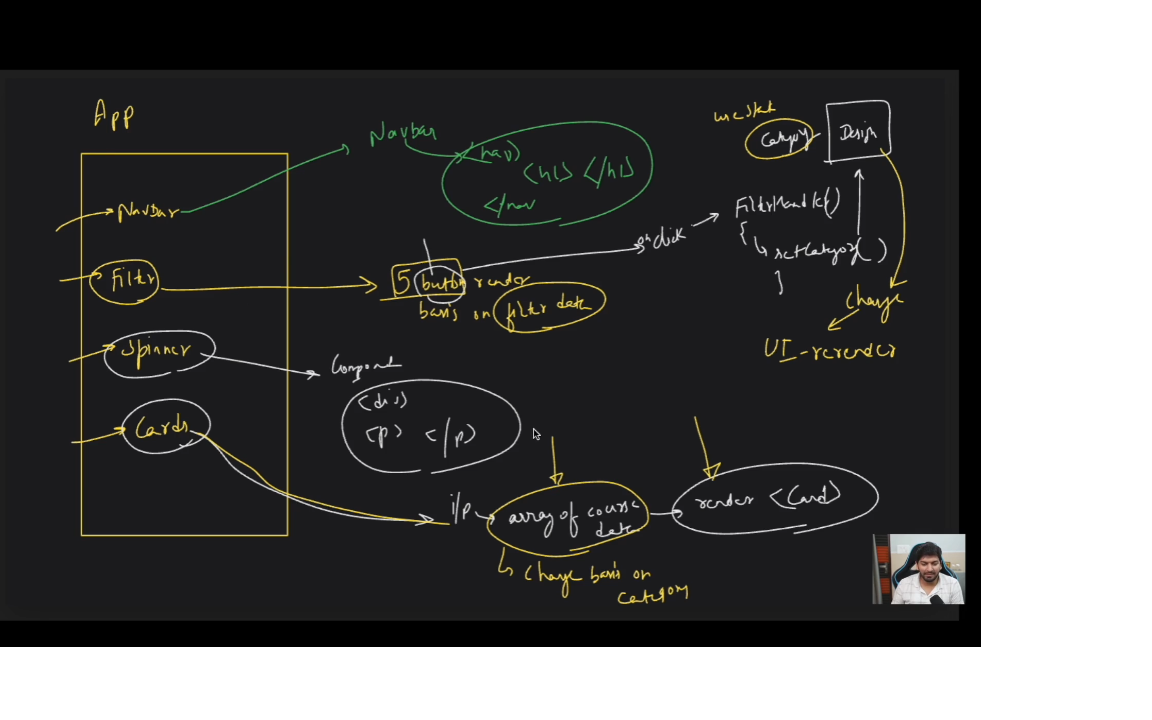
1. **Fetching API Data** (e.g., GET requests).
2. **Subscribing/Unsubscribing** (e.g., WebSocket, events).
3. **Listening to DOM Events** (e.g., resize, scroll).
4. **Updating DOM/Title** (e.g., animations, page titles).

**Comparison of useState and useEffect:**

| **Feature** | **useState** | **useEffect** |
| --- | --- | --- |
| **Purpose** | Manages component state. | Handles side effects. |
| **Trigger** | State updates trigger re-renders. | Runs based on dependencies or renders. |
| **Example Usage** | Counter, form inputs, toggles. | API calls, event listeners, timers. |

**Best Practices for useEffect:**

1. Always specify dependencies to avoid **unnecessary re-renders**.
2. Use cleanup functions for side effects like event listeners or subscriptions.
3. Avoid placing expensive computations or API calls directly inside useEffect; use **helper functions** instead.



**30th November Theory Notes**

**1. Opacity**

* Opacity refers to how transparent or see-through an element is on a web page.
* **Opacity values** range from 0 (fully transparent, invisible) to 1 (fully opaque, solid).
* It is often used to create effects like fading or making background images/text semi-transparent.

**Example**:

opacity: 0.5; /\* Makes the element 50% transparent \*/

**2. Conditional Classes in Tailwind CSS**

* Conditional classes let us change styles dynamically in React using JavaScript expressions like ${}.
* **Example Use Case**:
  + Applying a class based on whether a condition is true or false.
  + Example:
  + className={`text-lg px-2 py-1 ${
  + category === filter.title ? "border-black" : "border-transparent"
  + }`}
  + This code dynamically applies a border-black class if the category matches the filter.title. Otherwise, it uses border-transparent.

**3. React Props**

* **What are Props?**  
  Props (short for “properties”) are used in React to pass data from a parent component to a child component.
* **How do they work?**  
  Imagine a parent gives instructions to a child. Similarly, props carry data or instructions from one component to another.

**Example**:

<Filter filterData={filterData} />

Here, filterData is being sent as a prop to the Filter component.

**4. State Management with useState**

* **What is useState?**  
  useState is a React Hook used to manage values that can change in a component (called "state").
* **Why use it?**  
  When the state changes, the component automatically re-renders to show the new value.
* **Example**:
  + To manage the “Read More” functionality:
  + const [readmore, setReadmore] = useState(false);
    - Clicking a button toggles the state between true and false to show more or less text.

**5. Error Handling in React**

* **Problem**: React throws an error if you try to use .map() on undefined or null.
* **Solution**: Check if data exists before using .map().
* return courses ? courses.map(...) : null;

**6. Fetching Data with fetch**

* Fetching data in React is done using the fetch function inside the useEffect Hook.
* **Steps**:
  + Call the API to get data.
  + Convert the data into JSON format.
  + Store it in a state variable to use in your component.

**Example**:

const fetchData = async () => {

try {

const response = await fetch(apiUrl);

const data = await response.json();

setCourses(data.courses);

} catch (error) {

console.error("Error fetching data:", error);

}

};

**7. Loading Spinner**

* A spinner is a visual element that tells the user that data is loading.
* **Why use it?**  
  To improve user experience during delays (like API calls).
* **How to Create**: A spinner is usually a styled div with a circular border animation.

**Example**:

return (

<div className="spinner">

<p>Loading...</p>

</div>

);

**8. Data Filtering in React**

* Filtering allows us to display only the data that matches specific criteria (e.g., a category).
* **How it works**:
  + If the selected category is "All," display all courses.
  + Otherwise, display only the courses in the selected category.

**Example**:

const getCourses = () => {

if (category === "All") {

return Object.values(courses).flat();

}

return courses[category];

};

**9. Like Button (FcLike Icon)**

* The "like" button allows users to mark their favorite courses.
* **Key Learning**:
  + Added functionality to track liked items using useState.
  + Used the FcLike icon from react-icons for visual representation.
  + Placed the button using CSS properties like position: absolute to overlay it on images.