📓 React Interview Series

🎯 What We Will Learn

- What to expect from this React Interview Questions series
- Depth of concepts covered
- Preview of the first lecture

🤔 Introduction to the Series

Welcome to the **React Interview Questions** series — a comprehensive set of tutorials designed to help you **crack React interviews**, from **junior** to **senior developer** level.

☑️ Goals of the Series:

- Provide an **in-depth understanding** of core React concepts
- Help you handle **interview questions** confidently and clearly
- Cover **real-world use cases** and **source-level details**

📍 What to Expect in the First Lecture

In the very first lecture, the instructor lays out what will be covered:

🔍 Core Topics:

- **How React works under the hood**
- Setting up a **React app manually** — *without* using any bundlers like:
  - `create-react-app (CRA)`
  - `Vite`
- Understanding the **Virtual DOM** in depth
- Exploring **React's internal algorithms**
- And **a lot more advanced topics**

💬 Tone and Delivery

- The series takes a **step-by-step, deep-dive approach**
- No assumptions made — ideal for **serious learners**
- Designed to prepare you for both:
  - **Conceptual questions**
  - **Practical coding rounds**

Here are the **full, detailed, no-shortcut notes** for the React tutorial you shared. These notes explain everything from first principles — setting up React from scratch using CDNs, JSX, Babel, and rendering a simple counter app — **just like you were watching the tutorial live**. The structure follows a logical path and includes detailed code and explanations for each step.

🤔 React From Scratch — Full Walkthrough (No Create React App or Vite)

📌 What We Will Learn

- Why React exists
- Difference between React and Vanilla JS
- Introduction to **React** and **ReactDOM**

- What is JSX?
- What is Babel and why is it needed?
- Creating a React component manually
- Using `React.createElement()` to create DOM trees
- Introduction to state and `useState`
- Rendering the app using CDN setup
- Why CRA or Vite makes things easier
- Writing JSX with Babel in plain HTML setup

🚀 Motivation: Why Do We Need React?

As apps and websites became more interactive and complex, managing everything manually with plain JavaScript became messy, repetitive, and hard to maintain.

React (created by Meta) was introduced to solve this.

It allows us to:

- Break UIs into reusable building blocks called **components**
- Efficiently update only the parts of the DOM that change (using **Virtual DOM** via ReactDOM)
- Create fast, dynamic, and maintainable UIs

React is **not just for web** — tools like **React Native** allow us to build **mobile apps** too!

🛠️ Setting Up React Without CRA or Vite

We're doing everything from **scratch**, manually — using **CDNs** and pure HTML + JS.

🔧 Step 1: HTML Boilerplate

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>React Under the Hood</title>
</head>

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

    <!-- React and ReactDOM CDN -->
    <script crossorigin src="https://unpkg.com/react@18/umd/react.development.js">
</script>
    <script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>

    <!-- Babel CDN for JSX support -->
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>

    <!-- Your JS code -->
    <script type="text/babel" src="script.js"></script>
</body>

</html>
```

🔥 What This Does:

- React & ReactDOM are loaded via CDN
- Babel is used to **transpile JSX** (React's HTML-like syntax) to JS that browsers understand
- `#root` is where your entire React app will render

🧱 React Without JSX – Manual Component with `createElement`

Let's write a basic **Counter** component **without JSX** to understand what happens under the hood.

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

    const increment = () => {
        setCount(count + 1);
    };

    return React.createElement("div", null,
        React.createElement("p", null, `Count: ${count}`),
        React.createElement("button", {
            onClick: increment
        }, "Increment")
    );
}
```

🧠 Explanation:

- `React.createElement(tag, props, children...)` builds a virtual DOM tree
- React compares this with the real DOM and makes minimal changes
- This is what JSX gets converted into under the hood

🧩 Mounting React Component

Now, mount this component into the DOM using **ReactDOM.createRoot**:

```
const rootElement = document.getElementById("root");
const root = ReactDOM.createRoot(rootElement);
root.render(React.createElement(Counter));
```

☑️ Output:

> *A paragraph and button rendered inside `#root`. Clicking the button increments the count.*

💡 What Is JSX?

JSX stands for **JavaScript XML**. It's a syntax extension that lets you write **HTML-like code** inside JavaScript:

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

  const increment = () => {
    setCount(count + 1);
  };

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

⚠ Problem:

Browsers don’t understand JSX. That's where **Babel** comes in.

🔁 Babel – JSX to JS Converter

- Babel is a **transpiler** that converts JSX into regular JS (`React.createElement(...)`)
- Without Babel, JSX code throws errors like “Unexpected token `<`”

🧠 JSX = Syntactic Sugar

JSX:

```
<h1>Hello</h1>
```

Babel transpiles to:

```
React.createElement("h1", null, "Hello");
```

🧪 JSX with Babel — Example Setup

Ensure:

- Babel CDN is included
- Your script tag has `type="text/babel"`

```
<script type="text/babel" src="script.js"></script>
```

Now you can write JSX directly inside `script.js` .

💉 What Is `useState` ?

`useState` is a React Hook used to create **reactive variables**.

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

- `count` is the current value
- `setCount` is a function to update it
- Updating state triggers **re-render**

Why not just use `let count = 0` ?

- Because React won't re-render if the variable changes
- React needs to be notified when to update the UI — that's what `useState` does

💥 Final JSX-Based Counter App Code

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

    const increment = () => {
        setCount(count + 1);
    };

    return ( <
        div >
        <
        p > Count: {
            count
        } < /p> <
        button onClick = {
            increment
        } > Increment < /button> < /
        div >
    );
}

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

> ☑ *This works because Babel is included and compiles JSX to regular JS behind the scenes.*

🛠 Why Not Always Set Up Manually?

Setting React manually is good for **learning**. But in real projects we use tools like:

- **Create React App (CRA)** — full-featured but heavy
- **Vite** — lightweight, super fast dev server

They handle:

- Babel + JSX + transpiling
- Hot module replacement

- File-based code splitting
- Environment setup

💬 Expert-Level Interview Questions

1. **What is the role of `ReactDOM.createRoot`?**
   > *It creates a root React node that controls rendering to a real DOM node, introduced in React 18 for concurrent features.*
2. **What happens when you click a button in a React component?**
   > *The event triggers a state update via `setState`, React re-renders the virtual DOM, compares with old, and updates the real DOM efficiently.*
3. **Why is JSX not valid JavaScript?**
   > *Because it includes XML-like syntax which browsers don't understand, so it needs to be transpiled using Babel.*
4. **Can we use `let count = 0` instead of `useState`?**
   > *No, because changes to regular variables won't trigger re-render in React. `useState` hooks into React's rendering lifecycle.*
5. **What does `React.createElement` return?**
   > *A plain JS object representing a Virtual DOM node.*
6. **How does Babel help React apps run in the browser?**
   > *Converts JSX and ES6+ syntax into compatible JavaScript using presets/plugins (like `@babel/preset-react`).*

Here are **fully detailed, structured notes** based on the video lesson you provided, written in a clear and complete format to help you deeply understand **React's rendering process** and the role of **Virtual DOM, state updates, diffing, and reconciliation**.

🧠 React Rendering Process

📌 What We Will Learn

- How React handles rendering internally
- The two phases of rendering: **Render Phase** and **Commit Phase**
- How state updates cause re-renders
- Why normal variables don't trigger re-render
- Parent-child component re-renders
- How multiple `useState` updates behave
- What is Virtual DOM and how React uses it
- Understanding the **Diffing Algorithm** and reconciliation

🚀 Motivation

React is **declarative**, not imperative. This means you don't directly interact with the DOM like in vanilla JavaScript. Instead, React uses an intermediate layer called the **Virtual DOM**, which allows React to:

- Batch DOM updates efficiently
- Minimize expensive DOM manipulation
- Make rendering fast and predictable

Let's understand how rendering works under the hood.

🧱 React Rendering Process

React's rendering is divided into **two main phases**:

1️⃣ Render Phase (a.k.a. Reconciliation Phase)

- React **calculates** what changes are required.
- It builds a **Virtual DOM Tree** (a JS object representation of real DOM).
- It **compares** the new Virtual DOM to the previous Virtual DOM using the **Diffing Algorithm**.

> *This phase is pure and does not affect the real DOM.*

2️⃣ Commit Phase

- React applies the calculated differences to the **real DOM**.
- This includes:
  - DOM mutations
  - Ref callbacks
  - Layout effects

> *This phase does affect the actual DOM.*

🔄 Two Types of Render

1. **Initial Render**
   - Occurs when the component first mounts.
   - JSX is transpiled by Babel into `React.createElement()` calls.
   - Virtual DOM is created and committed to the actual DOM.
2. **Re-render**
   - Triggered when state or props change.
   - A new Virtual DOM is created and compared to the previous one.
   - Only the **differences** are applied to the real DOM.

🔬 Example: Console Log on Render

```
console.log("Component rendered");
```

- On page load → Message appears once.
- On state update via a button (e.g., `setCount`) → Message appears again each time the component re-renders.

🧱 Why Normal Variables Don't Trigger Render

```
let count1 = 0;
count1 = count1 + 1;
```

- These are **normal JavaScript variables**, not part of React's **reactive system**.
- UI will **not update**, and the component **won't re-render**.
- Only `useState`, `useReducer`, or `useContext` changes trigger re-renders.

🐒 Parent → Child Render Chain

```
function CounterParent() {
  const [showMessage, setShowMessage] = useState(false);
  return (
    <>
      <button onClick={() => setShowMessage(!showMessage)}>Toggle</button>
```

```
      {showMessage && <p>Now you see me</p>}
      <Counter />
    </>
  );
}
```

Observations:

- On toggling state in **parent**, both parent and child are **re-rendered**.
- Components are just **JavaScript functions**, so they **execute again** when their parent renders.

🔁 Multiple State Updates at Once

```
const toggleMessages = () => {
    setShowMessage1(!showMessage1);
    setShowMessage2(!showMessage2);
};
```

- React **batches** state updates by default.
- Even if multiple states are updated, React only **re-renders once**.
- This batching behavior enhances performance.

❗ setState Pitfall: Repeated Calls Don’t Stack

```
setCount(count + 1);
setCount(count + 1);
setCount(count + 1);
```

Result:

- Only increases by **1**, not 3.
- Why?
  - React **schedules** state updates.
  - All calls above see the same `count` value and update it from the same snapshot.

☑️ Solution: Functional Updates

```
setCount(prev => prev + 1);
setCount(prev => prev + 1);
setCount(prev => prev + 1);
```

Result:

- Correctly increases count by **3**.
- Because each function receives the **latest updated value**.

🌿 Virtual DOM – What Is It?

- The **Virtual DOM** is a **JavaScript object** that represents the actual DOM structure.

- It contains all the elements (`type` , `props` , `children` , etc.).
- Created by **JSX → Babel → React.createElement** process.

```
const element = ( <
    div >
    <
    p > Count: 0 < /p> <
    button > Increment < /button> < /
    div >
);

// Internally becomes an object:
{
    type: 'div',
    props: {
        children: [{
                type: 'p',
                props: {
                    children: 'Count: 0'
                }
            },
            {
                type: 'button',
                props: {
                    children: 'Increment'
                }
            }
        ]
    }
}
```

🤯 Why Virtual DOM?

- Real DOM manipulations are **slow**.
- React uses Virtual DOM to:
  - **Detect changes**
  - **Batch and optimize** updates
  - **Minimize unnecessary operations**

⚛️ State Update Rendering Flow

Step-by-Step:

1. Component is initially rendered → Virtual DOM created.
2. User interacts (e.g., clicks a button).
3. `useState` triggers a **state change**.
4. React generates a **new Virtual DOM**.
5. React compares it with the **previous Virtual DOM** using:
   - The **Diffing Algorithm**
6. React finds **minimal changes** needed.
7. React performs those changes in the **Commit Phase** (real DOM).

🧾 Diffing Algorithm

- A **tree comparison algorithm**.
- Compares:
  - **Element types** (e.g., `div` vs `span`)
  - **Props**
  - **Keys** (for lists)
  - **Children**

Example:

Before:

```
<div>
  <p>Count: 0</p>
</div>
```

After clicking:

```
<div>
  <p>Count: 1</p>
</div>
```

- React sees only the **text inside** `<p>` has changed.
- Only that part is updated in the DOM.
- Super efficient 🚀

🧾 Summary

Concept	Description
Declarative	You describe "what UI should look like" and React figures out "how to do it."
Virtual DOM	Lightweight JS representation of actual DOM
Render Phase	Calculates changes → builds new Virtual DOM
Commit Phase	Applies the actual changes to the real DOM
Diffing	React compares old & new Virtual DOMs to find minimal changes
Reconciliation	The process of syncing changes from Virtual DOM to Real DOM
setState batching	Multiple state updates are combined to optimize performance
Functional Updates	Use `prev => prev + 1` to avoid stale state problems

🧠 Expert Interview Questions

1. **Why doesn't React re-render when a normal variable changes?**
   - React doesn't track changes to regular variables—only changes in **state** or **props** cause re-render.
2. **What is the difference between Virtual DOM and Real DOM?**
   - Virtual DOM is a JS representation of UI, used by React to efficiently update the Real DOM.

3. **What happens during the Render and Commit phases in React?**
   - **Render Phase** builds the new Virtual DOM and computes diffs.
   - **Commit Phase** applies changes to the actual DOM.
4. **How does React prevent unnecessary re-renders?**
   - Through **diffing** and **reconciliation**, React updates only the parts of the DOM that changed.
5. **Why does calling `setCount(count + 1)` multiple times only increment once?**
   - Because state updates are asynchronous and based on a stale snapshot. Solution: use functional updates.

15 of the most asked react JS interview questions

1. What is React and Why is It Used?

- **React** is a popular JavaScript library primarily used for building user interfaces, especially for single-page applications where you want a dynamic, interactive user experience.
- **Why use React?**
  - **Efficiency**: React uses a virtual DOM (Document Object Model) that enhances performance by reducing the number of direct interactions with the actual DOM.
  - **Component-Based Architecture**: React allows you to break down the UI into small, reusable pieces called components. This makes the development process more modular and maintainable.
  - **State Management**: React components can manage their own state, making it easy to update and render changes in the UI dynamically.
  - **Declarative Syntax**: React uses a declarative approach to building UIs, meaning you define what the UI should look like for any given state, and React takes care of the rendering for you.

2. What is JSX and Why is it Used?

- **JSX** stands for **JavaScript XML**. It allows you to write HTML-like syntax within JavaScript code. It's **not HTML**, but it looks similar, making it easier for developers to define the structure of the UI in a readable way.
- **Why use JSX?**
  - It allows HTML to be written directly inside JavaScript, making it easier to visualize the component structure.
  - React components are typically written in JSX to define what the UI should look like.
  - JSX is compiled to regular JavaScript by React tools (such as Babel), which is why it works seamlessly with JavaScript.

3. What is a React Component?

A **React component** is a **reusable building block** of the user interface (UI). It defines how a part of the UI should appear and behave.

- **Types of Components**:
  - **Functional components**: Simple JavaScript functions that accept props and return JSX.
  - **Class components**: More feature-rich components that extend from `React.Component` and allow for the use of features like `state` and lifecycle methods.

Here's a basic example of a **functional component**:

```
const MyComponent = () => {
  return <h1>Hello, from a functional component!</h1>;
};
```

4. What is the Difference Between State and Props?

- **State**:
  - Used to manage dynamic data within a component. It represents the internal state of a component and can be modified using functions like `setState` (in class components) or the `useState` hook (in functional components).
  - Example: A counter in a component is typically managed by state.
- **Props**:
  - Props (short for "properties") are read-only data passed **from parent components to child components**. They cannot be changed by the child component, only by the parent.
  - Example: Passing a `name` prop to a child component that renders it.

Key Differences:

- **Mutability**: State is mutable, props are immutable.
- **Ownership**: State is owned and controlled by the component itself, while props are owned and controlled by the parent component.
- **Usage**: State is used for data that can change over time, while props are used to pass data from one component to another.

5. What is Prop Drilling?

- **Prop drilling** occurs when you need to pass data from a parent component to a deeply nested child component. This is done through intermediate components, resulting in the data being "drilled" through all levels of components.
- **Example**: If you have a `Grandparent` component, `Parent` component, `Child` component, and `Grandchild` component, and the `Grandparent` needs to send data to the `Grandchild`, it will pass the data through `Parent` and `Child` components via props. This can lead to cumbersome code when many nested components are involved.
- **Solution to Prop Drilling**:
  - You can use **React Context API** or **Redux** to avoid prop drilling by providing data directly to components without the need to pass it through every level.

6. What is React Fragment?

- A **React Fragment** allows you to group multiple elements without introducing an extra node to the DOM.
  - **Why use it?**: Sometimes, you want to return multiple elements from a component, but wrapping them in a `div` or other elements can cause unnecessary nesting in the DOM, which might affect styling or layout.
  - A fragment does not create a new DOM node but allows you to return multiple JSX elements from a component.

Example:

```
return (
  <React.Fragment>
    <h1>Hello</h1>
    <p>This is a fragment example</p>
  </React.Fragment>
);
```

You can also use the shorthand syntax:

```
return (
  <>
    <h1>Hello</h1>
    <p>This is a fragment example</p>
  </>
);
```

7. How Do You Define and Use State in React Functional Components?

In React **functional components**, you use the `useState` hook to manage state.

- **Example**:

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

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

  const increment = () => {
    setCount(count + 1); // Updates state
  };

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

- **How is State Different from Normal Variables?**
  - State causes the component to re-render when it changes. Normal variables do not trigger a re-render when they are updated.
  - `useState` gives us a piece of state and a function to update that state. The UI updates automatically when state changes.

8. How Do You Define and Use State in React Class Components?

In **class components**, you define state inside the `constructor` and use `this.setState()` to update the state.

- **Example**:

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

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

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

  render() {
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

9. What is the Virtual DOM in React?

- The **Virtual DOM (VDOM)** is a lightweight representation of the real DOM in memory.
- **Why is it needed?**
  - Direct manipulation of the real DOM is slow and can cause performance issues, especially in large applications with frequent updates.
  - React uses the virtual DOM to optimize performance by only updating the real DOM when necessary.
- **How does it work?**
  - React creates a virtual DOM whenever the state or props of a component change.
  - The virtual DOM is compared with the actual DOM using an efficient algorithm called **Reconciliation**.
  - React determines the minimal set of changes needed and then applies them to the real DOM.
- **Benefits**:
  - Faster rendering by reducing direct DOM manipulation.
  - React improves performance by batch-updating the DOM and reducing the number of operations.

10. What is the Diffing Algorithm in React?

- The **Diffing Algorithm** is the process used by React to efficiently update the DOM. It compares the previous virtual DOM with the new virtual DOM and calculates the minimum number of changes needed to update the actual DOM.
- **How it works**:
  - When a component's state or props change, React generates a new virtual DOM.
  - React compares the old virtual DOM and the new virtual DOM to identify the changes that need to be applied to the real DOM.
  - The algorithm prioritizes updates based on the differences (diffs), optimizing re-renders.
- **Why is it important?**
  - It reduces the cost of updating the DOM by minimizing unnecessary updates, resulting in improved app performance.

11. What are React Lifecycle Methods?

- React **lifecycle methods** are special methods that get called at different stages of a component's life (from creation to destruction).
- **Common Lifecycle Methods**:
  - `componentDidMount`: Called after a component has been rendered to the screen. Ideal for making network requests.
  - `componentDidUpdate`: Called after a component's state or props have changed. Used for performing operations after the update.

  - `componentWillUnmount`: Called right before a component is removed from the DOM. Used for cleanup (e.g., cancelling API requests).
- **In Functional Components**:
  - You use **React hooks** like `useEffect` to perform similar tasks that lifecycle methods do in class components.

12. What is `useEffect` Hook?

- The `useEffect` hook lets you perform side effects in your function components, such as fetching data, setting up subscriptions, or manually changing the DOM.
- **Syntax**:

```
useEffect(() => {
  // Your side effect code here (e.g., API call)
}, [dependencies]);
```

- **How it works**:
  - The effect runs after the render, making it perfect for performing operations like fetching data.
  - You can specify dependencies in the second argument (array) to control when the effect should re-run. If no dependencies are provided, it runs on every render.
- **Example**:

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

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

  useEffect(() => {
    // Effect: Log count changes to the console
    console.log("Count changed to:", count);
  }, [count]); // Runs only when count changes

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

13. What is `useContext` Hook?

- The `useContext` hook allows you to access the value of a context directly in your functional components without the need to use `Context.Consumer`.
- **Why use it?**
  - To share data (like theme, language, authentication) across components without passing props down manually through every layer of components (i.e., without prop drilling).
- **Example**:

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

// Create a Context
const MyContext = React.createContext();

const MyComponent = () => {
  const contextValue = useContext(MyContext); // Access context value

  return <p>Context Value: {contextValue}</p>;
};

// Provider
const App = () => {
  return (
    <MyContext.Provider value="Hello from Context">
      <MyComponent />
    </MyContext.Provider>
  );
};
```

14. What is the `useReducer` Hook?

- The `useReducer` hook is an alternative to `useState` and is often preferred when the state logic is complex or involves multiple sub-values.
- **How it works**:
  - It takes a **reducer function** (similar to Redux reducers) and returns the current state and a dispatch function.
  - It's useful for managing complex state transitions.
- **Example**:

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

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 (
    <div>
      <p>Count: {state.count}</p>
```

```
      <button onClick={() => dispatch({ type: "increment" })}>Increment</button>
      <button onClick={() => dispatch({ type: "decrement" })}>Decrement</button>
    </div>
  );
};
```

15. What are Controlled and Uncontrolled Components?

- **Controlled Components**:
  - In a controlled component, form data (like input values) is controlled by the state of the React component.
  - You use state and `onChange` events to update the value.

  Example:

```
const ControlledInput = () => {
  const [value, setValue] = useState("");

  const handleChange = (e) => setValue(e.target.value);

  return <input type="text" value={value} onChange={handleChange} />;
};
```

- **Uncontrolled Components**:
  - In an uncontrolled component, form data is handled by the DOM itself, not by React state. You can use a `ref` to access the input value.

  Example:

```
const UncontrolledInput = () => {
  const inputRef = useRef(null);

  const handleSubmit = () => {
    alert(`Entered value: ${inputRef.current.value}`);
  };

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={handleSubmit}>Submit</button>
    </div>
  );
};
```

Map, Filter, and Reduce in React

1. Introduction to Map, Filter, and Reduce in React

- We have already covered the `map`, `filter`, and `reduce` array methods in JavaScript.
- In React, these methods are especially useful for rendering lists and processing data efficiently.

2. Map Function in React

What is the Map Function?

- The `map` function is commonly used to iterate through an array and render components dynamically in React.
- It allows the creation of a new array of React elements based on an original array.

Example: Rendering List Using Map

- Given a list of products, we want to render them dynamically in a React component.

```
const products = [
  { id: 1, name: "Product 1", price: 20, category: "electronics" },
  { id: 2, name: "Product 2", price: 30, category: "clothing" },
  // more products...
];

function ProductList() {
  return (
    <ul>
      {products.map((product) => (
        <li key={product.id}>
          {product.name} - ${product.price} - {product.category}
        </li>
      ))}
    </ul>
  );
}
```

Key Points:

- The `map` function iterates over `products`, and for each `product`, a new `<li>` element is returned.
- **Important**: Always provide a `key` when rendering lists to avoid React warnings and improve performance.
- **Return Syntax**:
  - If you use curly braces `{}`, you need to explicitly return the value.
  - If you use parentheses `()`, the return is implicit.

Common Mistake:

- If you don't return the value within curly braces `{}` in `map`, it won't render anything.

3. Filter Function in React

What is the Filter Function?

- The `filter` function creates a new array with elements that pass a test provided by a function.

Example: Filter Products by Category

- Let's say we want to filter products based on the category, e.g., "electronics".

```
function FilteredProducts() {
  const filteredProducts = products.filter(
    (product) => product.category === "electronics"
```

```
  );

  return (
    <ul>
      {filteredProducts.map((product) => (
        <li key={product.id}>
          {product.name} - ${product.price} - {product.category}
        </li>
      ))}
    </ul>
  );
}
```

Key Points:

- `filter` is used to create a new array with products where the `category` is "electronics".
- The filtered result is then mapped over to display the list.

4. Reduce Function in React

What is the Reduce Function?

- The `reduce` function executes a reducer function (that you provide) on each element of the array, resulting in a single output.

Example: Summing Total Prices

- To get the total price of all products, we can use `reduce`:

```
function TotalPrice() {
  const totalPrice = products.reduce(
    (accumulator, product) => accumulator + product.price,
    0
  );

  return <p>Total Price: ${totalPrice}</p>;
}
```

Key Points:

- `reduce` takes a callback function with two arguments:
  - `accumulator`: accumulates the result
  - `currentValue`: current item being processed
- The second argument (`0`) is the initial value of the accumulator.

5. Chaining Methods: Map, Filter, and Reduce Together

Example: Filtering and Rendering with Discount

- We can chain `filter` and `map` together to first filter products and then apply some transformation.

```
function DiscountedProducts() {
  const discountedProducts = products
    .filter((product) => product.price > 20)
    .map((product) => ({
      ...product,
      discountedPrice: product.price - product.price * 0.1,
    }));

  return (
    <ul>
      {discountedProducts.map((product) => (
        <li key={product.id}>
          {product.name} - ${product.discountedPrice} (Discounted from
${product.price})
        </li>
      ))}
    </ul>
  );
}
```

Key Points:

- **Chaining**: You can chain methods like `filter`, `map`, and `reduce` for more complex transformations.
- **Adding Keys**: Always ensure that each element has a `key` when rendering lists.

6. Handling Discount Calculations

Example: Applying Discount to Products with Price > 20

- You can use `map` to modify an array of objects, adding a new `discountedPrice` key.

```
function DiscountedPriceList() {
  const discounted = products
    .filter((product) => product.price > 20)
    .map((product) => ({
      ...product,
      discountedPrice: product.price - product.price * 0.1, // 10% discount
    }));

  return (
    <ul>
      {discounted.map((product) => (
        <li key={product.id}>
          {product.name}: ${product.discountedPrice} (Discounted from
${product.price})
        </li>
      ))}
    </ul>
  );
}
```

Key Points:

- Here, `filter` ensures we only process products with a price greater than $20.
- `map` is used to modify the price by applying a 10% discount.

7. Filter Unique Items in an Array

What is the Goal?

- Filter out duplicate items in an array, leaving only unique values.

Example: Filtering Unique Names

```
function UniqueNames() {
  const names = ["John", "Jane", "John", "Alex", "Alex"];

  const uniqueNames = names.filter((name, index) => names.indexOf(name) ===
index);

  return (
    <ul>
      {uniqueNames.map((name, index) => (
        <li key={index}>{name}</li>
      ))}
    </ul>
  );
}
```

Key Points:

- `filter` combined with `indexOf` ensures we only return the first occurrence of each element.
- The result is a list of unique names.

8. Conclusion

- In React, `map`, `filter`, and `reduce` are powerful tools for dynamically rendering lists and manipulating data.
- Understanding how to use these methods in combination will help you handle various data processing scenarios in React applications.
- Always ensure to provide a `key` when rendering lists to improve performance and avoid warnings.

Certainly! Here's a complete set of notes from the lesson based on the video transcription:

Conditional Operators in JavaScript (React)

Introduction

In this lesson, we will explore some of the **conditional operators** commonly used in JavaScript and React development, especially those that are frequently asked in interviews. We'll go through logical operators like **AND** (`&&`) and **OR** (`||`), as well as operators such as **Optional Chaining** (`?.`) and **Nullish Coalescing** (`??`).

1. AND (`&&`) vs. OR (`||`) Operator

Logical AND (`&&`) Operator:

- **Definition:** The logical AND operator (`&&`) returns `true` **only** if **both** operands are `true` . If either operand is `false` , it returns `false` immediately.
  - **React Usage:** The `&&` operator is commonly used to **conditionally render elements**. If the condition before `&&` is true, the expression after `&&` is rendered; otherwise, nothing is rendered.

Example:

```
function MyComponent() {
  const x = 5;
  const y = 10;

  return (
    <>
      {x > 0 && y > 0 ? <span>Both are greater than zero</span> : <></>}
    </>
  );
}
```

- **Explanation:**
  - In the example above, both `x` and `y` are greater than zero. Since the condition is true, the component renders `<span>Both are greater than zero</span>` .
  - If `x` were 0 or `y` were 0, nothing would be rendered.

Logical OR (`||`) Operator:

- **Definition:** The logical OR operator (`||`) returns `true` if **any one** of the operands is `true` . If both operands are `false` , it returns `false` .
  - **React Usage:** The `||` operator is used to **render an element** if **any condition** is true. If the first condition is `true` , the rest of the conditions are not checked.

Example:

```
function WeatherComponent() {
  const isRaining = false;
  const isSunny = true;

  return (
    <>
      {(isRaining || isSunny) && <span>It's either raining or sunny or both!
</span>}
    </>
  );
}
```

- **Explanation:**
  - Here, `isRaining` is `false` and `isSunny` is `true` . The OR operator checks the first condition (`isRaining`), and since it's `false` , it moves on to check the second condition (`isSunny`), which is `true` .
  - Since one of the conditions is `true` , the message "It's either raining or sunny or both!" is rendered.

Key Difference between AND (`&&`) and OR (`||`):

- The **AND** (`&&`) operator requires **both** conditions to be true to render the element.
- The **OR** (`||`) operator will render the element if **any one** of the conditions is true.

2. Optional Chaining (`?.`) vs. Nullish Coalescing (`??`)

Optional Chaining (`?.`):

- **Definition:** Optional chaining allows you to **safely access deeply nested properties** in objects, even if some of the properties along the chain are `null` or `undefined`. If any reference in the chain is `null` or `undefined`, it returns `undefined` instead of throwing an error.

Example:

```
function UserComponent() {
  const user = { address: { city: 'New York' } };

  return (
    <>
      <p>{user?.address?.city}</p>  {/* Safely accesses city */}
    </>
  );
}
```

- **Explanation:**
  - In this example, the component tries to access `user?.address?.city`. If `user` or `address` is `null` or `undefined`, the operation will **not throw an error**. Instead, it will return `undefined` gracefully.
  - Without optional chaining, accessing `user.address.city` when `address` is `undefined` would throw a runtime error.

Key Benefits of Optional Chaining:

- It makes the code **cleaner** and **less error-prone** when dealing with deeply nested data.
- It's especially useful when working with data that may or may not exist, such as from API responses.

Nullish Coalescing (`??`):

- **Definition:** The nullish coalescing operator (`??`) returns the **right-hand operand** if the left-hand operand is `null` or `undefined`. It **does not** return the right operand if the left operand is a **falsy value** such as `false`, `0`, `""` (empty string), or `NaN`.

Example:

```
function GreetingComponent() {
  const userInput = null;
  const defaultValue = "Hello, Guest!";

  return (
    <>
      <p>{userInput ?? defaultValue}</p>  {/* Displays "Hello, Guest!" */}
    </>
```

```
    );
}
```

- **Explanation:**
  - In this case, `userInput` is `null`, so the **nullish coalescing operator** returns the `defaultValue` ("Hello, Guest!").
  - If `userInput` had been an empty string (`""`), `0`, `false`, or `NaN`, `??` would still have returned `userInput` because **only** `null` or `undefined` trigger the default value.

Key Benefits of Nullish Coalescing:

- It allows you to **handle missing data** and fall back to default values while distinguishing between `null` / `undefined` and other falsy values (e.g., `0`, `false`).

3. Summary of Key Operators

AND (`&&`) Operator:

- **Used for:** Rendering an element only if **all conditions** are true.
- **Example:** `condition1 && condition2 && <Component />`

OR (`||`) Operator:

- **Used for:** Rendering an element if **any condition** is true.
- **Example:** `condition1 || condition2 || <Component />`

Optional Chaining (`?.`) Operator:

- **Used for:** Safely accessing nested properties in an object without throwing an error when encountering `null` or `undefined`.
- **Example:** `user?.address?.city`

Nullish Coalescing (`??`) Operator:

- **Used for:** Providing a **default value** when an operand is `null` or `undefined` (but **not** for other falsy values like `0` or `false`).
- **Example:** `userInput ?? "default value"`

React Components: Class-Based vs. Function-Based

In this section, we're diving into React components and learning how to create them using both class-based and function-based approaches.

Class-Based Components in React

Class-based components are the traditional way of defining components in React, and they are written using ES6 class syntax. Although they have been largely replaced by function-based components in modern React development, class-based components still offer a lot of value. They can hold state and access lifecycle methods, making them essential for understanding React's evolution.

How to Create a Class-Based Component

1. **Syntax:** A class-based component extends `React.Component`:

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

    render() {
        return ( <
            div >
            <
            p > Count: {
                this.state.count
            } < /p> <
            button onClick = {
                this.incrementCount
            } > Increment < /button> < /
            div >
        );
    }

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

2. **Key Concepts in the Example:**
   - **Constructor & `super(props)` **: The constructor is used for initializing the state and binding methods to the component instance. The `super(props)` is necessary to access `this.props` in the class.
   - **State**: The state is defined as an object in the constructor. For instance, `this.state = { count: 0 }` .
   - **Render Method**: The `render()` method returns the JSX that will be rendered to the DOM.
   - **Event Handling**: The `incrementCount` method updates the state when the button is clicked.

Understanding the `constructor` and `super` Keywords

Constructor:

- **Purpose:** Used for initializing state and binding event handler methods.
- **Binding Methods:** In class components, methods need to be explicitly bound to the component instance, as shown with `this.incrementCount = this.incrementCount.bind(this)` or by using arrow functions.

Super Keyword:

- **Purpose:** The `super` keyword calls the constructor of the parent class (`React.Component`) and allows the component to access props and methods from the parent class.
- **Usage in the constructor:** `super(props)` ensures that the `props` are correctly passed down to the base class (i.e., `React.Component`).

Component Lifecycle Methods

In class-based components, React provides lifecycle methods that are called at different stages of a component's life, such as when it is mounted, updated, or unmounted.

1. **Mounting Phase**: Occurs when the component is created.
   - `constructor`: Initializes the component state and binds methods.
   - `render`: Renders the JSX.
   - `componentDidMount`: Runs after the component is mounted. Often used for API calls or setting up subscriptions.

   Example:

```
componentDidMount() {
    console.log("Component Mounted");
}
```

2. **Updating Phase**: When the component is updated due to changes in state or props.
   - `componentDidUpdate(prevProps, prevState)`: Called after the component updates. Useful for comparing previous and current state or props.

   Example:

```
componentDidUpdate(prevProps, prevState) {
    if (prevState.count !== this.state.count) {
        console.log("Count updated");
    }
}
```

3. **Unmounting Phase**: When the component is removed from the DOM.
   - `componentWillUnmount`: Used to clean up resources like subscriptions or timers.

   Example:

```
componentWillUnmount() {
    console.log("Component Unmounted");
}
```

Function-Based Components and Hooks

What are Hooks?

Hooks allow functional components to manage state and lifecycle methods. Prior to hooks, functional components were limited in terms of state and lifecycle capabilities, but hooks changed that.

Two commonly used hooks are:

1. `useState`: Allows you to add state to functional components.

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

2. `useEffect`: Allows you to handle side effects (e.g., fetching data, subscribing to events) in functional components. It can replace lifecycle methods such as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.
   Example:

```
useEffect(() => {
    console.log("Component Mounted");
}, []); // Empty array means it runs only once when the component mounts.
```

- **Dependencies:** If you add dependencies to the array, the effect runs again when those values change.

Example with state change tracking:

```
useEffect(() => {
    console.log("Count updated:", count);
}, [count]);
```

Converting Class-Based Components to Functional Components

In many job interviews, you might be asked to convert class-based components into function-based ones. This often happens when a company wants to modernize its codebase written in class components. Let's see how you can do this with the `DataList` component.

1. **Class-Based Version**:

```
class DataList extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            data: [],
            isLoading: true,
            error: null
        };
    }

    componentDidMount() {
        fetch('https://api.example.com/data')
            .then(response => response.json())
            .then(data => this.setState({
                data,
                isLoading: false
            }))
            .catch(error => this.setState({
                error,
                isLoading: false
            }));
    }

    render() {
        if (this.state.isLoading) {
            return <div > Loading... < /div>;
```

```
        }

        if (this.state.error) {
            return <div > Error: {
                this.state.error.message
            } < /div>;
        }

        return ( <
            div > {
                this.state.data.map(item => ( <
                    div key = {
                        item.id
                    } > {
                        item.name
                    } < /div>
                ))
            } <
            /div>
        );
    }
}
```

2. **Function-Based Version**:

```
const DataList = () => {
    const [data, setData] = useState([]);
    const [isLoading, setIsLoading] = useState(true);
    const [error, setError] = useState(null);

    useEffect(() => {
        fetch('https://api.example.com/data')
            .then(response => response.json())
            .then(data => {
                setData(data);
                setIsLoading(false);
            })
            .catch(error => {
                setError(error);
                setIsLoading(false);
            });
    }, []); // Empty dependency array to mimic componentDidMount

    if (isLoading) {
        return <div > Loading... < /div>;
    }

    if (error) {
        return <div > Error: {
            error.message
        } < /div>;
    }
```

```
        return ( <
            div > {
                data.map(item => ( <
                    div key = {
                        item.id
                    } > {
                        item.name
                    } < /div>
                ))
            } <
            /div>
        );
    };
```

Key Differences:

- **State Management**: In class components, state is managed using `this.state` and `this.setState()` . In functional components, you use the `useState` hook to manage state.
- **Lifecycle Methods**: Class components use lifecycle methods like `componentDidMount` , `componentDidUpdate` , and `componentWillUnmount` . In functional components, you use the `useEffect` hook to handle side effects and mimic these lifecycle methods.
- **Rendering**: Class components use the `render()` method to return JSX, while functional components return JSX directly from the function body.
- **Binding**: In class components, you need to bind methods to the component instance. In functional components, arrow functions automatically bind `this` to the component context.
- **Syntax**: Class components are defined using ES6 class syntax, while functional components are defined as regular JavaScript functions or arrow functions.
- **Hooks**: Functional components can use hooks like `useState` , `useEffect` , and custom hooks, which provide additional functionality and allow for cleaner code organization.
- **Performance**: Functional components with hooks can be more performant and easier to read, especially for simple components. They also allow for better code reuse through custom hooks.
- **Testing**: Functional components are often easier to test and reason about due to their simpler structure and lack of lifecycle methods.
- **Error Boundaries**: Class components can be used as error boundaries to catch JavaScript errors in their child components. Functional components cannot be used as error boundaries directly, but you can create higher-order components (HOCs) or use libraries like `react-error-boundary` to achieve similar functionality.
- **Ref Handling**: Class components can use `React.createRef()` to create refs, while functional components use the `useRef` hook to create refs. The syntax and usage differ slightly, but both achieve similar results in accessing DOM elements or storing mutable values.

Props, State, and Children in React

Props

- **Definition**: Props (short for "properties") are **read-only** data passed from a parent component to a child component. They are **immutable** and used for communication between components.
- **Class Components**: In class components, props are accessed using `this.props` . Here's an example:

```
class ChildComponent extends React.Component {
    render() {
        return <div > Name: {
            this.props.name
        }, Age: {
```

```
            this.props.age
        } < /div>;
    }
}
```

The parent component passes props like this:

```
class ParentComponent extends React.Component {
    render() {
        return <ChildComponent name = "John"
        age = {
            30
        }
        />;
    }
}
```

The child component receives the props (`name` and `age`) and displays them.

- **Functional Components**: In functional components, props are passed as arguments to the function. You can also **destructure** props for easy access.

```
const ChildComponent = ({
    name,
    age
}) => {
    return <div > Name: {
        name
    }, Age: {
        age
    } < /div>;
};
```

You can still pass props in a similar way from the parent component:

```
const ParentComponent = () => {
    return <ChildComponent name = "John"
    age = {
        30
    }
    />;
};
```

Difference between Class and Functional Components:

- In class components, props are accessed using `this.props` , while in functional components, props are passed as function arguments and can be destructured directly.

State

- **Definition**: State is **mutable** and represents the internal state of a component. The state is managed **locally** within the component and can be changed using the state update function.
- **Class Components**: In class components, state is defined using `this.state` and updated using `this.setState()`. Example:

```
class Counter extends React.Component {
    constructor() {
        super();
        this.state = {
            count: 0
        };
    }
    increment = () => {
        this.setState({
            count: this.state.count + 1
        });
    };
    render() {
        return ( <
            div >
            Count: {
                this.state.count
            } <
            button onClick = {
                this.increment
            } > Increment < /button> < /
            div >
        );
    }
}
```

- **Functional Components**: In functional components, **hooks** (specifically `useState`) are used to manage state. Example:

```
const Counter = () => {
    const [count, setCount] = useState(0);
    const increment = () => setCount(count + 1);
    return ( <
        div >
        Count: {
            count
        } <
        button onClick = {
            increment
        } > Increment < /button> < /
        div >
    );
};
```

`useState` returns an array with the current state value and a function to update it.

- **Key Differences Between Props and State**:
  - **Props** are **immutable** and passed down from a parent component to a child.
  - **State** is **mutable** and is managed within the component itself.
  - **Class Components**: Access props using `this.props` and state using `this.state`.
  - **Functional Components**: Access props directly as arguments and manage state using the `useState` hook.

Children Prop

- **Definition**: The **children prop** allows components to accept and render dynamic content passed between opening and closing tags. It is useful for creating reusable and flexible components.
- **Example**: A **Card** component can be created that accepts dynamic content via `children` .

```
const Card = (props) => {
    return ( <
        div className = "card" > {
            props.children
        } <
        /div>
    );
};
```

Now, in the parent component:

```
const App = () => {
    return ( <
        Card >
        <
        h1 > Card Title < /h1> <
        p > This is some content inside the card < /p> < /
        Card >
    );
};
```

The `children` prop makes it possible to pass any JSX elements between the `Card` tags, allowing for flexible and reusable components.

- **Use Case**: The `children` prop can be particularly useful in larger applications where components need to be adaptable and dynamic. You can also combine other JSX elements inside the `children` prop, such as making the content bold or adding other elements inside the card.

Summary

- **Props** are read-only, immutable, and used for communication between components, passed from parent to child.
- **State** is mutable, managed within the component, and represents the internal state of the component.
- The **children prop** allows dynamic content to be passed into components, making them reusable and flexible.

React Component Types

1. Smart/Stateful/Container Components

- **Definition**: These are components that manage their state, handle business logic, and pass data to child components. They are also known as "container components."
- **Characteristics**:
  - **State Management**: They handle state and contain the logic for their operations.
  - **Data Flow**: They pass down data to presentational components (dumb components) through props.
  - **Lifecycle Methods**: These components can have lifecycle methods to control their behavior during various stages (e.g., `componentDidMount`).

Example: In the lesson's counter example, the component manages state, business logic, and renders itself. If there are child components, it passes props to them.

2. Dumb/Stateless/Presentational Components

- **Definition**: These components are responsible solely for rendering UI based on the props they receive. They do not have state or business logic.
- **Characteristics**:
  - **No State**: They receive their data via props and render it.
  - **No Logic**: They don't have any business logic or lifecycle methods.
  - **Focus on UI**: The main purpose is to display UI elements, and they are commonly reused.

Example: A simple child component that takes in props and displays them without any additional logic.

3. Higher-Order Components (HOC)

- **Definition**: A higher-order component is a function that takes a component and returns an enhanced version of it, typically by adding additional logic or behavior.
- **Characteristics**:
  - **Reusability**: They allow sharing logic between components without repeating the code.
  - **Function-Based**: HOCs are functions that wrap and enhance a component by passing additional props or modifying its behavior.
  - **Naming Convention**: The convention is to prefix the HOC with `with` (e.g., `withLogger`).

Example:

```
const withLogin = (WrappedComponent) => {
    return class extends React.Component {
        login() {
            // login logic
        }
        logout() {
            // logout logic
        }
        render() {
            return <WrappedComponent {
                ...this.props
            }
            login = {
                this.login
            }
            logout = {
                this.logout
            }
            />;
```

```
        }
    };
};
```

By wrapping a component with `withLogin` , we can reuse the login and logout logic across multiple components without repeating the code.

4. Pure Components

- **Definition**: A pure component is a component that optimizes rendering performance by preventing unnecessary re-renders. It does so by shallowly comparing props and state to determine if a re-render is necessary.
- **Characteristics**:
  - **Shallow Comparison**: It only re-renders if the props or state have changed.
  - **Improved Performance**: By reducing unnecessary renders, it enhances performance, especially in large applications.

Example:

```
class MemoizedComponent extends React.PureComponent {
    render() {
        console.log("PureComponent re-rendered");
        return <div > Pure Component < /div>;
    }
}
```

For Functional Components: The equivalent of `React.PureComponent` is `React.memo` . It prevents unnecessary re-renders by memoizing the component's output.

```
const MemoizedFunctionalComponent = React.memo((props) => {
    console.log("Functional component re-rendered");
    return <div > Functional Component < /div>;
});
```

5. Controlled Components

- **Definition**: Controlled components are those in which React manages the value of form elements (like input fields) through state.
- **Characteristics**:
  - **State-Controlled**: The value of the input field is tied to the state in React.
  - **Two-Way Binding**: React updates the state whenever the input changes, and the input reflects the current state.

Example:

```
const ControlledInput = () => {
    const [value, setValue] = React.useState("");
    return ( <
        input type = "text"
```

```
            value = {
                value
            }
            onChange = {
                (e) => setValue(e.target.value)
            }
            />
        );
    };
```

6. Uncontrolled Components

- **Definition**: Uncontrolled components are those where the value of form elements is not directly controlled by React. Instead, the DOM itself manages the state, and React interacts with it using refs.
- **Characteristics**:
  - **DOM-Controlled**: The DOM controls the value, not React state.
  - **Interaction via Refs**: React uses refs to interact with the DOM elements and retrieve their values when necessary.

Example:

```
const UncontrolledInput = () => {
    const inputRef = React.useRef(null);

    const handleClick = () => {
        console.log(inputRef.current.value);
    };

    return ( <
        div >
        <
        input ref = {
            inputRef
        }
        type = "text" / >
        <
        button onClick = {
            handleClick
        } > Get Value < /button> < /
        div >
    );
};
```

In this case, React doesn't control the input value directly. Instead, we access it using `inputRef` .

Summary

- **Smart/Stateful/Container Components**: Manage state and pass props.
- **Dumb/Stateless/Presentational Components**: Render UI based on props.
- **Higher-Order Components (HOC)**: Enhance components with additional functionality.
- **Pure Components**: Prevent unnecessary re-renders for improved performance.

- **Controlled Components**: React manages input field state.
- **Uncontrolled Components**: DOM manages input field state, and React interacts via refs.