React Questions / Week 1 - The Compilation (15July-21July)

❓ Q #1 (🟢 Basic)

Tags: 🧩 General Interview , 🧱 Components

Question:

What is React?

✅ **Answer:**

React is an open-source JavaScript library primarily used for building user interfaces (UIs), especially for single-page applications (SPAs). It allows developers to create reusable UI components and efficiently update the DOM using a virtual DOM 1.

❓ Q #2 (🟢 Basic)

Tags: 🧩 General Interview , 🧱 Components

Question:

What are the key features of React?

✅ **Answer:**

Key features of React include:

- **Component-Based Architecture:** React applications are built using independent, reusable UI components 1.
- **Virtual DOM:** React uses a virtual DOM to optimize UI updates and improve performance 1.
- **JSX:** A syntax extension that allows writing HTML-like code within JavaScript 2.
- **Unidirectional Data Flow:** Data flows in one direction, typically from parent to child components 1.
- **Hooks:** Introduced in React 16.8, Hooks enable functional components to manage state and side effects 3.
- **Server-Side Rendering (SSR):** React supports SSR to improve SEO and initial load times 1.

❓ Q #3 (🟢 Basic)

Tags: 🧩 General Interview , 🧱 Components

Question:

Explain what JSX is and why it's used in React.

✅ **Answer:**

JSX (JavaScript XML) is a syntax extension that allows developers to write HTML-like code directly within JavaScript 2. It makes writing React components more intuitive and readable by combining UI structure with JavaScript logic. While browsers don't understand JSX natively, tools like Babel transpile it into standard JavaScript (e.g., `React.createElement` calls) that browsers can execute 4.

❓ Q #4 (🟢 Basic)

Tags: 🧩 General Interview , 🧱 Components

Question:

Why do multiple JSX tags need to be wrapped in a single root element or a Fragment?

✅ **Answer:**

JSX transforms into plain JavaScript objects representing UI elements. Similar to how a JavaScript function cannot return multiple objects without wrapping

them (e.g., in an array), multiple JSX elements must be enclosed within a single parent tag (like a `<div>`) or a `React.Fragment` to return a single root element. This ensures that the JSX structure maps correctly to the underlying JavaScript object representation 4.

❓ Q #5 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧱 Components , ⚙️ Performance

Question:

How does Babel contribute to React development?

✅ **Answer:**

Babel is a JavaScript compiler that plays a crucial role in React development by converting JSX into standard JavaScript that browsers can understand. It also transpiles newer JavaScript features (like ES6+) into backward-compatible versions. The process involves parsing the code into an Abstract Syntax Tree (AST), applying transformations, and then generating the output JavaScript code 4.

❓ Q #6 (🟢 Basic)

Tags: 🧩 General Interview , 📦 State Management

Question:

What is the difference between props and state in React?

✅ **Answer:**

Props (Properties) are read-only data passed from a parent component to a child component. They are immutable within the child component and are used to configure or reuse components 45.

State is mutable data managed within a component. It allows components to update their internal data dynamically and triggers re-renders when changed. State is typically managed using `useState` in functional components or `this.setState` in class components 45.

❓ Q #7 (🟡 Intermediate)

Tags: 🚀 MAANG-level , 🔧 Hooks , 📦 State Management

🔥 **EXTREMELY IMPORTANT**

Question:

Explain `useState` in functional components. How do you update state, and why are immutable updates important?

✅ **Answer:**

`useState` is a React Hook that allows functional components to manage and retain data between renders. It returns an array with two elements: the current state value and a setter function to update it (e.g., `const [count, setCount] = useState(0)`).

To update state, you call the setter function (e.g., `setCount(count + 1)`). Immutable updates are crucial because React relies on shallow comparisons of state to determine if a re-render is necessary. If you directly mutate an object or array in state, React might not detect the change, leading to the UI not updating. By returning a *new* object or array with the updated values, you ensure React recognizes the change and triggers a re-render of the component 45.

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

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

 const increment = () => {

// Immutable update: creating a new value
   setCount(count + 1);
 };

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

❓ Q #8 (🟡 Intermediate)

Tags: 🧩 General Interview , 📦 State Management , 💥 Pitfall

Question:

What is `this.setState` in class components, and what's a common pitfall when using it?

✅ **Answer:**

`this.setState` is the method used in class components to update the component's state. It merges the provided object into the current state and triggers a re-render. A common pitfall is directly mutating `this.state` (e.g., `this.state.count++`)

instead of using `this.setState`. Direct mutation will not trigger a re-render, leading to an outdated UI 4. `this.setState` also takes an optional callback function that fires after the state update is complete and the component has re-rendered.

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

 increment = () => {

// Correct way: using setState
   this.setState({ count: this.state.count + 1 });


// Incorrect way (will not trigger re-render):// this.state.count++;
 };

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

❓ Q #9 (🟡 Intermediate)

Tags: 🚀 `MAANG-level` , 🥾 `Components` , 🔍 `Real-World Scenario`

🔥 EXTREMELY IMPORTANT

Question:

Differentiate between controlled and uncontrolled components in React forms. When would you use each?

✅ Answer:

- **Controlled Components**: Form elements (like `<input>` , `<textarea>` , `<select>`) whose values are entirely controlled by React state. Their values are updated via `onChange` event handlers, which set the state, making React the "single source of truth" for the input data 4.
    - **Use when**: You need real-time validation, dynamic input disabling, formatted input, or when the form data needs to be immediately

reflected in the UI or passed to other components. They offer more control and predictability 4.

- **Uncontrolled Components**: Form elements whose values are managed by the DOM itself, rather than by React state. You access their values directly from the DOM using `refs` (e.g., `useRef`) typically when the form is submitted 4.
    - **Use when**: You need a simpler solution for basic forms, for integrating with non-React code, or when you only need the value on demand (e.g., on form submission). They can sometimes offer better performance by avoiding re-renders on every input change, but validation is harder to implement in real-time 4.

❓ Q #10 (🟡 Intermediate)

Tags: 🧩 `General Interview` , 🔧 `Hooks`

🔥 **EXTREMELY IMPORTANT**

Question:

What is `useEffect` in React, and what are its common use cases?

✅ **Answer:**

`useEffect` is a React Hook that allows functional components to perform side effects. A side effect is any operation that impacts something outside the current function's scope, such as data fetching, directly manipulating the DOM, setting up event listeners, or cleaning up resources 46.

`useEffect` accepts two arguments:

1. A function containing the side effect logic.
2. An optional dependency array.

Common use cases include:

- **Fetching data from APIs** when the component mounts or when specific data dependencies change 4.
- **Setting up subscriptions or event listeners** (e.g., for `resize` events) 4.
- **Directly manipulating the DOM** (e.g., focusing an input) 4.
- **Performing cleanup operations** when the component unmounts or before the effect re-runs (via a return function in `useEffect`) 46.

```
javascriptimport React, { useState, useEffect } from 'react';

function DataFetcher() {
  const [data, setData] = useState(null);

  useEffect(() => {

// Side effect: fetching data
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(json => setData(json));

// Cleanup function: runs on unmount or before re-run
    return () => {

// e.g., cancel fetch request, remove event listener
    };
  }, []);
// Empty dependency array: runs only once after initial render

  return (
    <div>
      {data ? <p>{data.message}</p> : <p>Loading...</p>}
    </div>
  );
}
```

❓ Q #11 (🟡 Intermediate)

Tags: 🚀 MAANG-level , 🔧 Hooks , 💥 Pitfall

Question:

Explain the purpose of the dependency array in `useEffect`. What happens if it's omitted or empty?

✅ **Answer:**

The dependency array (`[]`) in `useEffect` specifies a list of values that the effect depends on. The effect function will re-run only if any of these values change between renders 46.

- **Omitted (no dependency array)**: The `useEffect` callback will run after *every* render of the component. This is rarely desired and can lead to performance issues or infinite loops if not handled carefully 46.
- **Empty dependency array (`[]`)**: The `useEffect` callback will run only once after the initial render (mount) of the component. It's often used for effects that only need to be set up once, like initial data fetching or setting up global event listeners 46.

- **With dependencies (`[dep1, dep2]`)**: The `useEffect` callback will run after the initial render, and then *only* when any of the specified dependencies change their value 46. This is essential for preventing stale closures and ensuring your effects always work with the latest values.

❓ Q #12 (🟡 Intermediate)

Tags: 🧩 `General Interview` , 🔧 `Hooks` , ⚙️ `Performance`

Question:

When would you use `useRef` in React? Provide an example.

✅ **Answer:**

`useRef` is a Hook that returns a mutable ref object whose `.current` property can hold any mutable value. This value persists across renders without causing a re-render when it changes 7.

You use `useRef` when your component needs to "step outside" of the React rendering flow, typically for:

- **Accessing and interacting with DOM elements directly**: Such as focusing an input, playing media, or measuring element dimensions 7.
- **Storing mutable variables that don't trigger re-renders**: Useful for keeping track of a value that changes but shouldn't cause the component to update, like a timer ID or a previous state value 7.
- **Keeping a mutable value across re-renders**: The ref object itself is stable across renders.

Example: Focusing an input field

```
javascriptimport React, { useRef } from 'react';

function MyForm() {
 const inputRef = useRef(null);

 const handleClick = () => {
   inputRef.current.focus();
// Direct DOM manipulation
 };

 return (
   <div>
     <input type="text" ref={inputRef} />
     <button onClick={handleClick}>Focus Input</button>
   </div>
```

```
);
}
```

❓ Q #13 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , 🔧 `Hooks` , 💥 `Pitfall`

Question:

Explain what a "stale closure" is in the context of React's `useEffect`, and how it can be resolved.

✅ **Answer:**

A **stale closure** occurs when a function (the closure) "captures" a variable's value from its lexical scope at the time it was created, but later expects the *latest* value of that variable, which may have since changed. In `useEffect`, this typically happens when an effect depends on state or props but doesn't list them in its dependency array, causing the effect to run with an outdated snapshot of those variables 48.

Example of stale closure:

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

 useEffect(() => {
   const interval = setInterval(() => {
     console.log(`Count is: ${count}`);
// 'count' will always be 0 if [] is the dependency
   }, 1000);
   return () => clearInterval(interval);
 }, []);
// Problem: `count` is captured as 0 on initial render
}
```

Resolution:

The primary way to resolve stale closures in `useEffect` is by correctly listing all dependencies in the dependency array. When `count` is added to the array, the effect will re-run whenever `count` changes, creating a new `setInterval` with the updated `count` value 48.

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

 useEffect(() => {
   const interval = setInterval(() => {
     console.log(`Count is: ${count}`);
// 'count' will be the latest value
   }, 1000);
   return () => clearInterval(interval);
```

```
}, [count]);
// Solution: `count` is now a dependency
}
```

Another solution, especially for values that shouldn't trigger a re-run but need to be current, is to use `useRef` to store the latest value 48.

❓ Q #14 (🔴 Advanced)

Tags: 🚀 `MAANG-level` , ⚙️ `Performance` , 🔍 `Real-World Scenario`

🔥 EXTREMELY IMPORTANT

Question:

What is debouncing, and how would you implement it in a React component for an input field (e.g., search bar)?

✅ Answer:

Debouncing is a technique used to delay the execution of a function until a specified amount of time has passed since its last invocation. This is particularly useful for events that fire rapidly (like keystrokes in a search input or window resizing), preventing the function from being called too frequently and improving performance 48.

Implementation in React (for a search bar):

The typical approach involves using `useEffect` with a `setTimeout` to delay setting the debounced value. If the input value changes before the timeout, the previous timeout is cleared and a new one is set.

```
javascriptimport React, { useState, useEffect } from 'react';

function SearchBar() {
 const [query, setQuery] = useState('');
// User-typed search input
 const [debouncedQuery, setDebouncedQuery] = useState('');
// Debounced value// Effect to debounce the query
 useEffect(() => {
   const handler = setTimeout(() => {
     setDebouncedQuery(query);
   }, 500);
// Debounce for 500ms// Cleanup: clear the timeout if query changes before the delay
   return () => {
     clearTimeout(handler);
   };
 }, [query]);
// Re-run effect only when 'query' changes// Effect to perform API call with debounced query
 useEffect(() => {
   if (debouncedQuery) {
     console.log("Making API call with:", debouncedQuery);
```

```
    // In a real app, you would trigger your API call here// e.g., fetch(`/api/search?q=${debouncedQuery}`);
    }
  }, [debouncedQuery]);
  // Re-run effect only when 'debouncedQuery' changes

  return (
    <input
      type="text"
      placeholder="Search..."
      value={query}
      onChange={(e) => setQuery(e.target.value)}
    />
  );
}
```

This implementation ensures that the API call (or any expensive operation) is only triggered after the user has stopped typing for a brief period, significantly reducing unnecessary requests and improving responsiveness 4.

❓ Q #15 (🔴 Advanced)

Tags: 🚀 MAANG-level , 🔁 Rendering , ⚙️ Performance

🔥 EXTREMELY IMPORTANT

Question:

What is React Reconciliation and the Diffing Algorithm? How do they work together to optimize UI updates?

✅ Answer:

Reconciliation is the process React uses to efficiently update the actual DOM to match the desired UI represented by the virtual DOM. When a component's state or props change, React builds a new virtual DOM tree 49.

The **Diffing Algorithm** is the core part of reconciliation. It's the highly optimized algorithm React uses to compare the newly generated virtual DOM tree with the previous one 49. Instead of re-rendering the entire UI, the diffing algorithm identifies the minimal set of changes needed to update the real DOM 49.

How they work together:

1. **Virtual DOM Creation:** When state or props change, React creates a new Virtual DOM tree representing the updated UI 49.
2. **Diffing:** The diffing algorithm compares this new Virtual DOM with the old one. It makes two key assumptions to optimize this comparison:

- Elements of different types (e.g., `<div>` to `<span>`) result in a complete re-build of the subtree 109.
- Keys, when used in lists, help React identify which items have changed, been added, or removed, allowing it to efficiently update individual elements without re-rendering the entire list 109.

3. **Real DOM Update:** Once the differences are determined, React performs only the necessary updates on the actual DOM. This minimizes direct DOM manipulations, which are typically slow, leading to improved performance and a fast user experience 49.

❓ Q #16 (🟡 Intermediate)

Tags: 🧩 General Interview , 🔁 Rendering , 💥 Pitfall

Question:

Why is it recommended to use a stable and unique `key` prop when rendering lists in React, and why should you avoid using `index` as a key if the list items can change order?

✅ **Answer:**

The `key` prop helps React identify which items in a list have changed, been added, or removed. It allows React to efficiently track and update individual elements when the list is re-rendered 410.

Why avoid `index` as a key if order changes:

If you use the array `index` as a key and the order of items in the list changes (e.g., due to sorting, adding/removing items in the middle), React might incorrectly reuse or reorder existing DOM elements. This can lead to:

- **Performance issues:** React may re-render more than necessary or perform inefficient DOM manipulations.
- **State bugs:** If list items have internal state (e.g., an input field's value), using `index` as a key can cause that state to be incorrectly associated with a different item if the order changes, leading to unexpected behavior and data inconsistencies 410.

It's best to use a stable and unique ID that comes from your data (e.g., `item.id` from a database) as the key 410. Only use `index` as a key if the list is static and its order will never change 10.

```javascript
// Good practice: using a unique ID as key
{items.map((item) => (
  <li key={item.id}>{item.name}</li>
))}

// Bad practice if list order changes: using index as key
{items.map((name, index) => (
  <li key={index}>{name}</li>
))}
```

❓ Q #17 (🟢 Basic)

Tags: 🧩 General Interview , 🧰 Components

Question:

What are the different phases of a React component's lifecycle?

✅ **Answer:**

In a broad sense, a React component typically goes through four main phases 13:

1. **Initialization:** The component prepares by setting up initial props and state.
2. **Mounting:** The component is created and inserted into the DOM for the first time.
3. **Updating:** The component re-renders due to changes in its state or props.
4. **Unmounting:** The component is removed from the DOM.

❓ Q #18 (🟡 Intermediate)

Tags: 🧩 General Interview , ⚙️ Performance

Question:

How can you optimize React component re-renders? Name a few strategies.

✅ **Answer:**

Optimizing re-renders is crucial for performance. Several strategies can be employed 4:

- `shouldComponentUpdate` **(Class Components)**: A lifecycle method that allows you to prevent a re-render if the component's props or state haven't relevantly changed 4.

- `React.memo` **(Functional Components)**: A higher-order component (HOC) that memoizes a functional component, preventing re-renders if its props haven't changed 4.
- `React.PureComponent` **(Class Components)**: A base class that automatically implements `shouldComponentUpdate` with a shallow comparison of props and state 4.
- **Efficient** `key` **Prop Usage**: Using stable and unique keys for list items helps React update lists efficiently 410.
- **Avoiding Inline Functions and Objects in JSX**: Creating new function or object instances on every render can cause unnecessary re-renders of child components. Use `useCallback` for functions and `useMemo` for objects to memoize them 4.
- **State Colocation**: Moving state down to the lowest possible component that needs it to limit the scope of re-renders.

❓ Q #19 (🟢 Basic)

Tags: 🧩 General Interview , 🧰 Components

Question:

What is the `children` prop in React?

✅ **Answer:**

The `children` prop is a special prop in React that allows you to pass components or elements as data to other components by nesting them inside the JSX tag. It enables content to be passed directly to a component, similar to how HTML elements contain their content 5. This is commonly used for layout components or generic containers.

```
javascriptfunction Card({ title, children }) {
 return (
   <div className="card">
     <h3>{title}</h3>
     {children} {
/* This renders whatever is passed inside <Card> tags */  }
   </div>
 );
}

 // Usage:
<Card title="Welcome">
 <p>This is content passed via the children prop.</p>
```

```
<button>Click Me</button>
</Card>
```

❓ Q #20 (🟡 Intermediate)

Tags: 🧩 General Interview , 📦 State Management , 💥 Pitfall

Question:

Can you directly modify state in React? Why or why not?

✅ **Answer:**

No, you should **never directly modify state** in React (e.g., `this.state.count = 1` or `myArray.push(item)` for `useState`). State should only be updated using the provided setter functions (`setState` for class components or the `set` function from `useState` for functional components) 45.

The reason is that React relies on these setter functions to detect changes in state and trigger the reconciliation process, which eventually leads to a UI re-render. If you directly modify the state, React won't be notified of the change, and the component's UI will not update, leading to inconsistencies between the actual state and what is displayed on the screen 45.

❓ Q #21 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧰 Debugging

Question:

What are React Developer Tools, and how are they useful?

✅ **Answer:**

React Developer Tools is a browser extension (available for Chrome, Firefox, etc.) that allows developers to inspect the React component hierarchy, state, props, and performance of a React application. It's incredibly useful for debugging by [search_web_response]:

- **Inspecting component trees:** Visualizing the nested structure of components.
- **Modifying state and props:** Directly changing values to test different UI states without reloading.
- **Tracing re-renders:** Identifying why components are re-rendering unnecessarily to optimize performance.

- **Profiling performance:** Analyzing render times and component updates.

❓ Q #22 (🟢 Basic)

Tags: 🧩 General Interview , 🧰 Components

Question:

What are functional components in React?

✅ **Answer:**

Functional components are JavaScript functions that accept a single `props` object argument and return React elements (JSX). They are simpler to write and read than class components and became the preferred way to write React components after the introduction of Hooks, which allowed them to manage state and side effects 4.

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

❓ Q #23 (🟢 Basic)

Tags: 🧩 General Interview , 🧰 Components

Question:

What are class components in React?

✅ **Answer:**

Class components are ES6 classes that extend `React.Component` and have a `render()` method that returns React elements (JSX). They were traditionally used to manage state and lifecycle methods before the introduction of Hooks. While still supported, they are generally less common in new React development 4.

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

❓ Q #24 (🟡 Intermediate)

Tags: 🧩 General Interview , 🔧 Hooks

Question:

When would you choose a functional component over a class component in modern React?

✅ **Answer:**

In modern React, functional components are generally preferred over class components for several reasons 4:

- **Simplicity and Readability:** They are simpler JavaScript functions, making the code cleaner and easier to understand.
- **Hooks:** Hooks (like `useState`, `useEffect`, `useContext`) allow functional components to manage state and side effects, making them as capable as class components for most use cases 3.
- **Better Performance Optimization Potential:** With `React.memo` and `useCallback` / `useMemo`, functional components can be easily optimized to prevent unnecessary re-renders.
- **Less Boilerplate:** They require less boilerplate code compared to class components (e.g., no `constructor`, `this` binding issues).

❓ Q #25 (🟢 Basic)

Tags: 🧩 `General Interview` , 🧱 `Components`

Question:

What is the purpose of `render()` method in class components?

✅ **Answer:**

The `render()` method is a required method in class components. Its primary purpose is to return the React elements (JSX) that describe the UI structure of the component. When the component's state or props change, the `render()` method is called again to produce a new set of React elements, which React then uses to update the actual DOM via reconciliation 3.

❓ Q #26 (🟡 Intermediate)

Tags: 🧩 `General Interview` , ⚙️ `Performance`

Question:

What is the Virtual DOM, and why does React use it?

✅ **Answer:**

The **Virtual DOM (VDOM)** is a lightweight, in-memory representation (a JavaScript object tree) of the actual DOM elements 41. React creates a virtual copy of the UI.

React uses the Virtual DOM to optimize updates because directly manipulating the Real DOM is slow and computationally expensive 4. When the state or props of a component change:

1. React creates a new Virtual DOM tree.
2. It compares this new Virtual DOM with the previous one using its diffing algorithm.
3. It then calculates the minimal set of changes required.
4. Finally, React applies only these necessary changes to the Real DOM, avoiding costly full re-renders and improving performance 41.

❓ Q #27 (🟢 Basic)

Tags: 🧩 General Interview , ⚙️ Performance

Question:

What is `npm start` and `npm create vite@latest` used for in React development?

✅ **Answer:**

- `npm start`: This command is typically used in `create-react-app` based projects to start the development server. It compiles the React application and serves it on a local port, usually with hot reloading, allowing developers to see changes in real-time 4.
- `npm create vite@latest`: This command is used to scaffold a new project using Vite, a modern front-end build tool. Vite is often preferred over `create-react-app` for its faster development server and build times, especially for smaller projects, due to its use of ES modules and native browser support for module imports 4.

❓ Q #28 (🟢 Basic)

Tags: 🧩 General Interview , 🧱 Components

Question:

What are props used for?

✅ **Answer:**

Props (short for properties) are used to pass data from a parent component to a child component in React. They allow parent components to configure and customize their child components. Props are immutable within the child component, meaning a child component cannot directly modify the props it receives 45.

❓ Q #29 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧰 Components

Question:

Are props objects? Explain.

✅ **Answer:**

Yes, `props` are indeed objects. When you pass multiple attributes to a child component in JSX (e.g., `<Child name="Alice" age={30} />`), React collects all these attributes and bundles them into a single JavaScript object. This `props` object is then passed as the first argument to the child functional component or accessed via `this.props` in a class component 5.

❓ Q #30 (🟡 Intermediate)

Tags: 🧩 General Interview , 🔧 Hooks

Question:

How does `useEffect` with a cleanup function work, and when is it necessary?

✅ **Answer:**

The `useEffect` hook can return a cleanup function. This function runs when the component unmounts or, importantly, *before* the effect re-runs (if its dependencies change). It's essential for "cleaning up" side effects to prevent memory leaks or unexpected behavior 46.

When it's necessary:

- **Removing event listeners:** To avoid multiple listeners accumulating or listening after the component is gone.
- **Clearing timers/intervals:** To prevent timers from running indefinitely after a component unmounts.

- **Canceling network requests:** To prevent updating state on an unmounted component.
- **Disconnecting from subscriptions:** To prevent memory leaks.

```
javascriptuseEffect(() => {

// Setup: add event listener
 window.addEventListener('resize', handleResize);

 return () => {

// Cleanup: remove event listener
   window.removeEventListener('resize', handleResize);
 };
}, []);
// Empty array, so setup and cleanup run once
```

❓ Q #31 (🟢 Basic)

Tags: 🧩 General Interview , 🧱 Components

Question:

What is unmounting in the React component lifecycle?

✅ **Answer:**

Unmounting is the final phase of a React component's lifecycle. It occurs when a component is removed from the DOM (e.g., when a user navigates away from a page, a component is conditionally rendered to `null`, or a parent component unmounts). During this phase, the `componentWillUnmount` lifecycle method (for class components) or the cleanup function returned from `useEffect` (for functional components) is called, allowing you to clean up any resources (like event listeners or timers) that were set up during the component's life 4111.

❓ Q #32 (🟡 Intermediate)

Tags: 🧩 General Interview , 📦 State Management

Question:

How does state trigger re-renders in React?

✅ **Answer:**

When a component's state changes (by calling `setState` in class components or the setter function from `useState` in functional components), React is notified of this change. React then re-executes the `render()` method for class components

or the functional component's body. This process creates a new virtual DOM tree, which is then compared with the previous one (reconciliation), and only the necessary updates are applied to the actual DOM 45.

❓ Q #33 (🟡 Intermediate)

Tags: 🚀 `MAANG-level` , 💥 `Pitfall`

Question:

Explain the concept of "lexical scoping" and "closure" in JavaScript, and how they relate to React.

✅ **Answer:**

- **Lexical Scoping**: In JavaScript, a function's scope is determined by where it is defined (lexically), not where it is called. Inner functions have access to variables declared in their outer (parent) functions 8.
- **Closure**: A closure is a function that remembers and has access to its lexical environment (i.e., variables from its outer scope) even after the outer function has finished executing 8. The inner function "closes over" those variables.

Relation to React:

Closures are fundamental to how React works, especially with Hooks. For example, when you define a function inside a functional component, that function forms a closure over the state and props variables of that component's render. This allows event handlers or `useEffect` callbacks to access the values of state and props. However, this is also where "stale closures" can arise if dependencies are not properly managed in `useEffect`, leading to functions capturing outdated variable values 48.

❓ Q #34 (🟡 Intermediate)

Tags: 🧩 `General Interview` , ⚙️ `Performance`

Question:

What are `React.memo` , `useCallback` , and `useMemo` used for, and how do they help optimize performance?

✅ **Answer:**

These are performance optimization tools in React that leverage memoization to prevent unnecessary re-renders:

- `React.memo`: A Higher-Order Component (HOC) used to memoize functional components. If a component is wrapped with `React.memo`, it will only re-render if its props have shallowly changed. This prevents child components from re-rendering when their parent re-renders but their own props are the same 4.
- `useCallback`: A Hook that memoizes functions. It returns a memoized version of the callback function that only changes if one of the dependencies in its dependency array has changed. This is crucial when passing callbacks to optimized child components (`React.memo`) to prevent unnecessary re-renders of the child 4.
- `useMemo`: A Hook that memoizes computed values. It only re-computes the memoized value when one of the dependencies in its dependency array has changed. This is useful for expensive calculations or for memoizing objects/arrays that are passed as props to child components, preventing unnecessary re-renders 4.

How they help: By memoizing components, functions, or values, they reduce the number of times React performs costly re-renders and computations, leading to improved application performance, especially in large and complex applications.

❓ Q #35 (🔴 Advanced)

Tags: 🚀 MAANG-level, 🏗️ Architecture, ⚙️ Performance

Question:

Explain the concept of "lifting state up" in React. When and why would you apply this pattern?

✅ **Answer:**

"Lifting state up" is a React pattern where the state that is shared or needs to be synchronized between multiple sibling components is moved up to their closest common ancestor component. Instead of each child managing its own portion of shared state, the parent component becomes the "single source of truth" for that shared state 12.

When to apply:

- **Sibling Communication**: When two or more sibling components need to share data or interact with each other based on a common state.
- **Parent Control**: When a parent component needs to control or react to changes in its children's UI or data.

Why apply:

- **Single Source of Truth**: Centralizes shared data, making it easier to reason about the application's data flow.
- **Synchronization**: Ensures that all relevant components are always in sync with the latest shared data.
- **Data Flow Predictability**: Adheres to React's unidirectional data flow, where data flows down via props and events bubble up via callbacks 1.
- **Simpler Debugging**: Easier to debug issues related to shared state as it's managed in one place.

The common ancestor then passes the state down to the relevant children as props, and children communicate changes back to the parent via callback functions passed as props.

❓ Q #36 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧱 Components

Question:

What is the purpose of `super(props)` in a class component's constructor?

✅ **Answer:**

In a React class component, the constructor is called before the component is mounted. When you define a constructor in a subclass, you must call `super()` before using `this`. Specifically, `super(props)` ensures that the `React.Component` base constructor is called with the component's props. This makes `this.props` available in the constructor and allows the component to properly initialize itself within the React ecosystem 4. If `super(props)` is not called, `this.props` will be `undefined` in the constructor, leading to issues.

❓ Q #37 (🟡 Intermediate)

Tags: 🧩 General Interview , ⚙️ Performance

Question:

When would you *not* use `index` as a key in a React list?

✅ **Answer:**

You should **not** use `index` as a key in a React list if the order of the items in the list can change (e.g., items can be added, removed, or reordered). This is because React uses keys to efficiently identify items during reconciliation. If the order changes and keys are based on indices, React might misidentify items, leading to incorrect UI updates, potential state bugs (e.g., wrong data associated with a component), or performance issues 410.

The only scenario where using `index` as a key is generally acceptable is when the list is static and will never change its order or contents 10. For dynamic lists, a stable and unique ID associated with each item is always the best practice 410.

❓ Q #38 (🟡 Intermediate)

Tags: 🧩 `General Interview` , 🧰 `Components`

Question:

How do you pass data from a parent component to a child component in React?

✅ **Answer:**

Data is passed from a parent component to a child component using **props**. In the parent component, you add attributes to the child component's JSX tag, which become the `props` object received by the child 5.

Parent Component:

```
javascriptfunction ParentComponent() {
 const message = "Hello from Parent!";
 return <ChildComponent data={message} />;
}
```

Child Component (Functional):

```
javascriptfunction ChildComponent(props) {
 return <p>{props.data}</p>;
}
```

Child Component (Functional with destructuring):

```
javascriptfunction ChildComponent({ data }) {
 return <p>{data}</p>;
}
```

❓ Q #39 (🟡 Intermediate)

Tags: 🧩 General Interview , 🧰 Components

Question:

How do you pass a function from a parent component to a child component, and why would you do this?

✅ **Answer:**

You pass a function from a parent to a child component in the same way you pass any other data: via **props**. The child component can then call this function, often passing arguments back to the parent 5.

Why do this?

This pattern is essential for **unidirectional data flow** and for handling events or actions that originate in a child component but need to affect the parent's state or trigger behavior in the parent. It allows the child component to communicate "up" to its parent without directly modifying the parent's state, adhering to React's principle of data flowing downwards 45.

Example:

Parent Component:

```
javascriptfunction ParentComponent() {
 const handleClick = (childData) => {
   console.log("Button clicked in child! Data:", childData);

 // Update parent state or perform action
 };

 return <ChildComponent onButtonClick={handleClick} />;
}
```

Child Component:

```
javascriptfunction ChildComponent({ onButtonClick }) {
 return (
   <button onClick={() => onButtonClick("Data from child")}>
     Click me
   </button>
 );
}
```

❓ Q #40 (🟢 Basic)

Tags: 🧩 General Interview

Question:

What is a "side effect" in React?

✅ **Answer:**

In React, a "side effect" refers to any operation that has an impact outside the scope of the current function component's rendering. Essentially, it's anything your component does *besides* just rendering UI 6. Common examples include:

- Fetching data from an API.
- Directly manipulating the DOM (e.g., changing a document title, focusing an input).
- Setting up event listeners (e.g., `window.resize`).
- Timers (e.g., `setTimeout`, `setInterval`).
- Logging to the console.

`useEffect` is the Hook specifically designed to handle these side effects in functional components 46.

❓ Q #41 (🟡 Intermediate)

Tags: 🧩 General Interview, ⚙️ Performance

Question:

What is the difference between `event.stopPropagation()` and `event.preventDefault()`? When would you use each?

✅ **Answer:**

Both are methods available on the synthetic event object in React, but they serve different purposes 6:

- `event.stopPropagation()`: This method stops the event from "bubbling up" (or propagating further down in the capturing phase) the DOM tree to parent elements. It prevents parent event handlers from being triggered by an event that originated in a child element 6.
  - **Use when**: You want to isolate an event to the element it occurred on and prevent it from affecting parent elements that might have their own event handlers (e.g., clicking a button inside a card, and you only want the button's click handler to fire, not the card's).
- `event.preventDefault()`: This method prevents the browser's default action for a given event. 6

- **Use when**: You want to override the browser's default behavior for an element (e.g., preventing a form from submitting and reloading the page on `onSubmit`, preventing a link from navigating on `onClick`, or stopping default scroll behavior).

```javascript
// Example: stopPropagation
<div onClick={() => console.log('Parent clicked')}>
  <button onClick={(e) => {
    e.stopPropagation();
// Prevents div's onClick from firing
    console.log('Button clicked');
  }}>
    Click me
  </button>
</div>

// Example: preventDefault
<form onSubmit={(e) => {
  e.preventDefault();
// Prevents page reload
  console.log('Form submitted');
}}>
  <input type="text" />
  <button type="submit">Submit</button>
</form>
```

❓ Q #42 (🟡 Intermediate)

Tags: 🧩 `General Interview` , ⚙️ `Performance`

Question:

Explain `React.createElement` and how it relates to JSX.

✅ **Answer:**

`React.createElement` is the core function that JSX compiles down to. While JSX is a syntactic sugar that makes it easier to describe UI elements with an HTML-like syntax, browsers don't understand JSX directly. Tools like Babel transpile JSX code into `React.createElement` calls 2.

`React.createElement` takes three main arguments:

1. The type of the element (e.g., a string like `'div'` or a React component reference).
2. An object containing the element's props (e.g., `className` , `onClick` , `style`).
3. Any children of the element (can be a single child, an array of children, or `null`).

Example:

```jsx
// JSX:
<div>
  <h1>Hello</h1>
</div>

// Transpiles to React.createElement calls:
React.createElement("div", null,
  React.createElement("h1", null, "Hello")
);
```

`React.createElement` returns a JavaScript object, known as a React Element, which is a lightweight description of what should be rendered to the DOM 2. React then uses these elements to build its Virtual DOM and perform efficient updates.

❓ Q #43 (🟡 Intermediate)

Tags: 🧩 `General Interview` , ⚙️ `Performance`

Question:

What are some alternatives to JSX if you don't want to use it?

✅ **Answer:**

While JSX is highly recommended and widely adopted, there are alternatives if you prefer not to use it, although they are less common in modern React development 2:

- **`React.createElement`**: You can directly use the `React.createElement` function to create React elements. This is what JSX transpiles into anyway 2.

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

- **`react-hyperscript`**: This library provides a more concise JavaScript function (often aliased as `h`) to create React elements, mimicking a JSX-like syntax without the need for a transpilation step for the HTML-like structure 2.

  ```javascript
  import h from 'react-hyperscript';
  h('div', [
    h('h1', 'Hello, World')
  ]);
  ```

- **`htm`**: This is a JSX alternative that uses tagged template literals to write HTML-like code within JavaScript, which is then processed into React elements without a separate JSX transpilation step 2.

```
javascriptimport htm from 'htm';
import React from 'react';
const html = htm.bind(React.createElement);

html`<div><h1>Hello, World</h1></div>`;
```

These alternatives demonstrate that JSX is a convenience layer, not a mandatory part of React itself.

❓ Q #44 (🟢 Basic)

Tags: 🧩 General Interview , 🧱 Components

Question:

What is the primary difference in how `class` attributes are handled in JSX compared to standard HTML?

✅ **Answer:**

In standard HTML, you use the `class` attribute to assign CSS classes to an element (e.g., `<div class="my-class">`). However, in JSX, `class` is a reserved keyword in JavaScript. Therefore, to assign CSS classes, you must use `className` instead (e.g., `<div className="my-class">`) 4. This is one of the key differences to remember when writing HTML-like code in JSX. Similarly, other attributes often follow `camelCase` (e.g., `tabindex` becomes `tabIndex`) 4.

❓ Q #45 (🟡 Intermediate)

Tags: 🧩 General Interview , ⚙️ Performance

Question:

How does Vite differ from Create React App in terms of development server and bundling?

✅ **Answer:**

Vite and Create React App (CRA) are both tools for setting up React projects, but they differ significantly in their approach to development and bundling 4:

- **Development Server (Speed)**:
  - **CRA (Webpack)**: Uses Webpack, which bundles the entire application before serving it. For larger applications, this can lead to slower cold start times and slower hot module replacement (HMR) updates as the entire bundle needs to be rebuilt or re-evaluated for small changes 4.

    - **Vite (Native ES Modules)**: Leverages native ES modules in the browser during development. It serves source files directly and only bundles code when requested by the browser. This results in significantly faster cold starts and near-instant HMR updates because only the changed modules are invalidated and re-sent to the browser 4.

- **Bundling (Production)**:
    - **CRA (Webpack)**: Uses Webpack for production bundling, which is robust but can be slower for large projects.
    - **Vite (Rollup)**: Uses Rollup for production bundling, which is generally faster and produces optimized bundles, often with smaller sizes.
- **Transpilation**:
    - **CRA**: Uses Babel for JSX transpilation 2.
    - **Vite**: Uses ESBuild for JSX transpilation, which is written in Go and significantly faster than Babel 2.

In essence, Vite aims for a much faster developer experience by optimizing the development server and build process 4.

❓ Q #46 (🟢 Basic)

Tags: 🧩 General Interview , 🥾 Components

Question:

What are events in React, and how do you handle them?

✅ **Answer:**

Events in React are synthetic events that provide a cross-browser wrapper around the browser's native event system. They are triggered by user interactions (like clicks, key presses, form submissions) or browser actions 6.

You handle events in React by passing a function as a prop to the event attribute (e.g., `onClick`, `onChange`, `onSubmit`) on a JSX element. The function receives a synthetic event object as its argument 6.

```
javascriptfunction MyButton() {
 const handleClick = (event) => {
   console.log("Button clicked!", event);

// event.target refers to the DOM element that triggered the event
 };
```

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

It's important to pass a *reference* to the function (`handleClick`) and not call it directly (`handleClick()`) in the JSX, otherwise, it will execute immediately on render instead of on the event 6.

❓ Q #47 (🟡 Intermediate)

Tags: 🧩 General Interview , 📦 State Management

Question:

Can a child component directly modify the props it receives from a parent? Why or why not?

✅ **Answer:**

No, a child component cannot directly modify the props it receives from a parent. Props are **read-only** within the child component 45.

This adheres to React's principle of **unidirectional data flow**, where data flows downwards from parent to child. This predictability makes it easier to understand how changes affect the application and helps prevent unintended side effects and bugs. If a child needs to communicate a change back to the parent, it should do so by calling a callback function that was passed down as a prop from the parent, allowing the parent to update its own state 45.

❓ Q #48 (🟡 Intermediate)

Tags: 🧩 General Interview , 🔧 Hooks

Question:

When would you use `useEffect` with an empty dependency array (`[]`) versus omitting the dependency array entirely?

✅ **Answer:**

- `useEffect` **with an empty dependency array (`[]`)**: The effect function will run only **once** after the initial render (component mounts) and will not re-run on subsequent updates. This is ideal for side effects that should only be set up or fetched once, such as initial data fetching, setting up global event listeners, or initializing third-party libraries 46. The cleanup function (if any) will run only when the component unmounts.

- `useEffect` **with no dependency array (omitted)**: The effect function will run after **every single render** of the component. This is rarely the desired behavior and can lead to performance issues, infinite loops (if the effect updates state that triggers another render), or unnecessary re-calculations. It effectively makes the effect behave like `componentDidMount` and `componentDidUpdate` combined, but without the benefit of dependency tracking 46.

Therefore, almost always include a dependency array to explicitly control when your effect runs, even if it's empty.

❓ Q #49 (🟡 Intermediate)

Tags: 🧩 `General Interview` , 📦 `State Management`

Question:

How do you pass multiple pieces of data from a parent to a child using props?

✅ Answer:

You pass multiple pieces of data from a parent to a child by adding multiple attributes to the child component's JSX tag. These attributes are then collected into a single `props` object that the child component receives. In the child component, you can access these pieces of data using dot notation (e.g., `props.name`) or by destructuring the `props` object 5.

Parent Component:

```
javascriptfunction ParentComponent() {
 const userName = "Alice";
 const userAge = 30;
 const isLoggedIn = true;

 return (
   <UserProfile
     name={userName}
     age={userAge}
     status={isLoggedIn ? "Online" : "Offline"}
   />
 );
}
```

Child Component (Functional, accessing via props object):

```
javascriptfunction UserProfile(props) {
 return (
   <div>
     <p>Name: {props.name}</p>
     <p>Age: {props.age}</p>
     <p>Status: {props.status}</p>
```

```
  </div>
 );
}
```

Child Component (Functional, with object destructuring for cleaner access):

```
javascriptfunction UserProfile({ name, age, status }) {
 return (
  <div>
   <p>Name: {name}</p>
   <p>Age: {age}</p>
   <p>Status: {status}</p>
  </div>
 );
}
```

❓ Q #50 (🟡 Intermediate)

Tags: 🧩 General Interview , 📦 State Management

Question:

What does it mean for `useState` to be "immutable via its setter"?

✅ Answer:

When we say `useState` is "immutable via its setter," it means that when you update state using the setter function (e.g., `setCount` from `const [count, setCount] = useState(0)`), you should provide a *new* value or a *new* object/array, rather than directly modifying the existing one 4.

For primitive values (numbers, strings, booleans), this is straightforward: `setCount(count + 1)` creates a new number.

For objects and arrays, direct mutation (e.g., `myArray.push(item)` or `myObject.property = value`) won't trigger a re-render. Instead, you must create a new array or object with the desired changes and pass that new instance to the setter:

```
javascript // Incorrect (direct mutation)
const [arr, setArr] = useState([1, 2]);
arr.push(3);
setArr(arr);
 // React might not detect change// Correct (immutable update for array)
const [arr, setArr] = useState([1, 2]);
setArr([...arr, 3]);
 // Creates a new array// Correct (immutable update for object)
const [obj, setObj] = useState({ name: 'Alice', age: 30 });
setObj({ ...obj, age: 31 });
 // Creates a new object
```