Q: 1. What is JSX, and how is it different from regular JavaScript?

A: JSX (JavaScript XML) is a syntax extension to JavaScript. It's not a separate language itself, but a way to write HTML-like code within your JavaScript. The key difference lies in how you structure and represent user interfaces.

Regular JavaScript: For creating user interfaces in plain JavaScript, you typically manipulate the DOM (Document Object Model) directly. This involves using JavaScript methods like `createElement`, `appendChild`, `setAttribute`, etc., to build and update the HTML structure dynamically. This can become verbose and difficult to manage for complex UIs.

JSX: JSX allows you to write HTML-like code within your JavaScript code. This code looks like HTML, but it's actually JavaScript expressions that are translated into JavaScript functions by a tool called a JSX transformer (like Babel). This translation creates the DOM manipulation for you.

Here's an illustration:

Regular JavaScript (DOM manipulation):

```javascript
const element = document.createElement('h1');
element.textContent = 'Hello, world!';
document.body.appendChild(element);
```

JSX (React example):

```javascript
const element = <h1>Hello, world!</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

Key Differences Summarized:

Feature	Regular JavaScript	JSX
Syntax	Standard JavaScript syntax	HTML-like syntax within JavaScript
DOM Manipulation	Explicit DOM manipulation using JavaScript methods	Implicit DOM manipulation; handled by a transformer
Readability	Can be verbose and less intuitive for UIs	More concise and easier to read for UIs
Compilation	No compilation required	Requires a JSX transformer (like Babel)
Use Case	General-purpose programming	Primarily used for building UIs (often with React)

In essence, JSX makes building user interfaces much more declarative and readable. You describe *what* you want the UI to look like, and the JSX transformer handles the underlying DOM manipulation. It's not mandatory for using JavaScript, but it's become incredibly popular because of its ease of use, especially in frameworks like React, Preact, and others that embrace it. Note that JSX itself isn't directly executable in a

browser; it must be compiled into plain JavaScript first.

Q: 2. What is the difference between a class component and a functional component in React?

A: The core difference between class and functional components in React lies in how they're defined and how they handle state and lifecycle methods. Functional components are simpler and generally preferred for most use cases in modern React, while class components are largely legacy.

Class Components:

* **Definition:** Defined using ES6 classes and extend the `React.Component` class.
* **State and Lifecycle:** Manage state using `this.state` and have access to a full lifecycle API (e.g., `componentDidMount`, `componentDidUpdate`, `componentWillUnmount`). These lifecycle methods allow you to perform actions at specific points in the component's existence (e.g., fetching data, cleaning up resources).
* **Methods:** Methods are defined as instance methods within the class using the `this` keyword.
* **Syntax:** More verbose and requires more boilerplate code.
* **Example:**

```javascript
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
```

```
    super(props);
    this.state = { count: 0 };
  }

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

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

export default MyComponent;
```

Functional Components:

* **Definition:** Defined as simple JavaScript functions.
* **State and Lifecycle (Hooks):** Don't directly manage state or lifecycle methods. Instead, they use React Hooks (introduced in React 16.8) like `useState`, `useEffect`, `useContext`, etc., to manage state and side effects. Hooks provide a more concise and

organized way to handle state and lifecycle concerns within functional components.

* **Methods:** Can use helper functions defined within the component.

* **Syntax:** More concise and easier to read.

* **Example:**

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

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

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

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

export default MyComponent;
```

Key Differences Summarized:

Feature	Class Component	Functional Component (with Hooks)
Definition	ES6 Class extending `React.Component`	JavaScript function
State Management	`this.state`	`useState` hook
Lifecycle	Lifecycle methods (`componentDidMount`, etc.)	`useEffect` hook
Syntax	More verbose	More concise
this Keyword	Used to access state and methods	Not needed (except in rare cases within closures)

In modern React development, functional components with Hooks are strongly preferred because they offer:

* **Improved readability:** Simpler and easier to understand.
* **Better maintainability:** Less boilerplate code leads to easier maintenance.
* **Testability:** Easier to test with standard JavaScript testing tools.
* **Reusability:** Hooks can be easily reused across different components.

Class components are mostly used for legacy codebases or in rare circumstances where a very specific, now-obsolete feature from the older lifecycle methods is strictly required. For almost all new development, functional components with Hooks are the

way to go.

Q: 3. Explain the concept of "state" in React. How do you manage state in functional

A: In React, "state" refers to the data that determines the behavior and rendering of a component. Whenever the state changes, the component re-renders to reflect those changes. Think of it as the component's internal memory or the data that the component directly controls. Unlike props, which are passed down from parent components, state is managed internally.

Managing State in Functional Components:

Before React Hooks (introduced in React 16.8), state management in functional components was cumbersome, requiring the use of higher-order components or class components. Hooks dramatically simplified this process. The primary hook used for managing state in functional components is `useState`.

`useState` Hook:

The `useState` hook is a function that takes an initial value as an argument and returns an array containing two elements:

1. **The current state value:** This is the data that your component is currently using.
2. **A function to update the state value:** This function, typically named `setState` (though you can name it anything), is used to change the state. Calling this function triggers a re-render of the component.

Here's how it works:

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

function MyComponent() {
  // Initialize the state with an initial value of 0
  const [count, setCount] = useState(0);

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

export default MyComponent;
```

In this example:

* `useState(0)` initializes the `count` state variable to 0.
* The hook returns an array `[count, setCount]`. `count` holds the current value (0 initially), and `setCount` is the function to update it.

* The `onClick` handler calls `setCount(count + 1)`, incrementing the count by 1. This triggers a re-render, updating the displayed value of `count`.

Important Considerations:

* **Immutability:** It's crucial to understand that you should *not* directly mutate the state value. Instead, create a *new* state object or value. For example, with objects or arrays, you should create copies and then modify the copies:

```javascript
const [myObject, setMyObject] = useState({ name: "Alice" });

// Incorrect: Directly mutates the state object
//myObject.name = "Bob";

// Correct: Creates a copy and updates the copy
setMyObject({ ...myObject, name: "Bob" });


const [myArray, setMyArray] = useState([1,2,3]);

//Incorrect: directly mutates the array
//myArray.push(4);

//Correct: Creates a copy and updates the copy
setMyArray([...myArray, 4]);
```

* **Asynchronous Updates:** State updates in React are asynchronous. If you rely on the updated state immediately after calling `setCount`, you might encounter unexpected behavior. Use the updated state only after a re-render.

* **Multiple States:** You can use multiple `useState` calls within a single component to manage multiple state variables.

* **State Lifting:** For managing state shared between multiple components, consider "lifting state up"—moving the state to their nearest common ancestor.

By understanding and utilizing the `useState` hook, you can efficiently manage state within your functional React components, leading to more dynamic and interactive user interfaces. For more complex state management scenarios (like managing data fetched from an API or dealing with a large amount of shared state), you might want to explore state management libraries like Redux, Zustand, Jotai, or Context API.

Q: 4. What is the purpose of the useEffect hook, and how does it work?
A: The `useEffect` hook in React allows you to perform side effects in your functional components. Side effects are actions that go beyond simply rendering the UI, such as:

* **Fetching data:** Making API calls to retrieve data from a server.
* **Manipulating the DOM directly:** Adding event listeners, updating the document title, or directly changing the styles of elements.
* **Setting timers:** Using `setTimeout` or `setInterval`.
* **Subscription management:** Subscribing to external data streams (like

WebSockets) and cleaning up those subscriptions.

How it works:

`useEffect` takes two arguments:

1. **A function:** This function contains the code that performs the side effect. It's executed after every render of the component.

2. **An optional dependency array:** This array lists the values that the effect depends on. React uses this array to determine when to re-run the effect.

    * **Empty array (`[]`):** If the dependency array is empty, the effect will only run once after the initial render, similar to `componentDidMount` in class components.

    * **With dependencies:** If the dependency array includes values, the effect will run after the initial render *and* whenever any of those values change. React compares the previous and current values of the dependencies. If they are different, the effect runs again. This is analogous to `componentDidMount` and `componentDidUpdate` combined in class components.

    * **No dependency array (omitted):** If the dependency array is omitted, the effect runs after every render. This is generally discouraged unless you intentionally want to perform an effect after *every* render, as it can lead to performance issues.

Example:

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

function MyComponent() {
  const [count, setCount] = useState(0);
  const [data, setData] = useState(null);

  useEffect(() => {
    // This effect runs only once after the initial render because of the empty array
    console.log("This runs only once!");
  }, []);



  useEffect(() => {
    //This effect runs whenever the 'count' value changes
    console.log("Count changed:", count);
  }, [count]);

  useEffect(() => {
      // Fetch data from an API when the component mounts, and clean up the subscription.
    const fetchData = async () => {
      const response = await fetch('/api/data');
      const jsonData = await response.json();
      setData(jsonData);
    };
```

```
    fetchData();
    return () => {
        // Cleanup function (optional):  Called before the next effect runs or component unmounts.
      console.log("Cleaning up...");
        //You would typically handle things like cancelling network requests, removing event listeners here.
    };
  }, []); //Empty array - runs only once

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
      {data && <pre>{JSON.stringify(data, null, 2)}</pre>}
    </div>
  );
}

export default MyComponent;
```

Cleanup Function (Return Value):

The return value of the effect function is a cleanup function. This function is called before the next effect runs or before the component unmounts. It's crucial for cleaning up after side effects to prevent memory leaks and unexpected behavior. For example,

you would use it to remove event listeners or cancel timers. If you don't need a cleanup function, you can omit the return statement.

In short, `useEffect` is a powerful tool for managing side effects in functional components, providing a clear and organized way to handle asynchronous operations, DOM manipulation, and other tasks that go beyond simply updating the UI. Careful consideration of the dependency array is vital for performance and avoiding unexpected behavior.

Q: 5. How do you pass data from a parent component to a child component in React?
A: In React, you pass data from a parent component to a child component using **props (properties)**. Props are read-only; the child component cannot modify them.

Here's how it works:

1. **In the Parent Component:** When you render the child component, you pass data as attributes to the child component's tag. These attributes become the child component's props.

2. **In the Child Component:** The child component receives these props as an object. You can access the data within the props object using the attribute names you assigned in the parent component.

Example:

Let's say we have a parent component called `App` and a child component called `Greeting`. We want to pass a name from `App` to `Greeting` to personalize the greeting.

Parent Component (App.js):

```javascript
import React from 'react';
import Greeting from './Greeting';

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

export default App;
```

Child Component (Greeting.js):

```javascript
import React from 'react';
```

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

export default Greeting;
```

In this example:

* The `App` component passes the `name` variable as a prop to the `Greeting` component using the attribute `name={name}`.
* The `Greeting` component receives this prop as `props.name` and uses it to display a personalized greeting.

Passing multiple props:

You can pass multiple pieces of data as props:

```javascript
//Parent Component
<Greeting name="Bob" age={30} city="New York" />
```

```
//Child Component
function Greeting(props) {
  return (
    <div>
      Hello, {props.name}! You are {props.age} years old and live in {props.city}.
    </div>
  );
}
```

Default Props:

You can define default values for props in the child component if a prop isn't provided by the parent:

```javascript
function Greeting(props) {
  const name = props.name || "Guest"; // Default to "Guest" if name is not provided
  return (
    <div>
      Hello, {name}!
    </div>
  );
}
```

In summary, props are the primary mechanism for passing data down the component tree in React, providing a clean and efficient way to share information between components. Remember that props are unidirectional – data flows from parent to child. For communication in the other direction, you'll need to use techniques like state and callbacks.

Q: 6. What is React's virtual DOM, and why is it important?
A: React's virtual DOM (VDOM) is a lightweight in-memory representation of the actual DOM (Document Object Model). It's essentially a JavaScript object that mirrors the structure of the real DOM. Instead of directly manipulating the browser's DOM, React first updates its virtual DOM. Then, React uses a sophisticated algorithm (called a diffing algorithm) to compare the previous virtual DOM with the updated one, identifying only the minimal changes needed. Finally, it applies only those changes to the actual DOM, making updates significantly faster and more efficient.

Why is it important?

* **Performance:** Directly manipulating the DOM is expensive. The browser has to repaint and reflow the entire page or affected sections every time something changes, leading to performance bottlenecks, especially in complex applications. The virtual DOM drastically reduces the number of direct DOM manipulations, improving performance significantly.

* **Efficiency:** The diffing algorithm cleverly minimizes the number of updates to the real DOM. It only updates the parts that have actually changed, avoiding unnecessary work and resulting in faster rendering.

* **Abstraction:** The virtual DOM provides an abstraction layer between the application's code and the actual DOM. This simplifies development and makes it easier to reason about how changes in the application's state affect the UI. Developers can focus on updating the virtual DOM, and React handles the complexities of updating the real DOM.

* **Cross-browser compatibility:** React handles the differences between different browsers' DOM implementations. Developers don't need to worry about the nuances of how each browser handles DOM updates. The virtual DOM acts as a consistent intermediary.

In short, the virtual DOM is a key component that contributes to React's speed, efficiency, and ease of use. It's a crucial part of what makes React a powerful and popular JavaScript library for building user interfaces.

Q: 7. What is the Context API, and how would you use it to manage global state?

A: The React Context API is a mechanism for passing data through the component tree without having to pass props down manually at every level. This is particularly useful for managing global state—data that needs to be accessible from many different parts of your application, such as user authentication status, theme settings, or language preferences. Instead of prop drilling (passing props down multiple levels), Context allows you to make this data available to any component that subscribes to it.

Here's how you would use it to manage global state:

1. Create a Context:

First, you create a Context object using `React.createContext()`. This returns an object with two properties: `Provider` and `Consumer`. In modern React, using the `useContext` hook is generally preferred over `Consumer`.

```javascript
import React, { createContext, useContext, useState } from 'react';

const MyContext = createContext(); // Creates the context object

// Context Provider Component
const MyProvider = ({ children }) => {
  const [globalState, setGlobalState] = useState({
    userName: 'Guest',
    theme: 'light',
  });

  const updateGlobalState = (newState) => {
    setGlobalState((prevState) => ({ ...prevState, ...newState }));
  };

  const contextValue = { globalState, updateGlobalState };

  return (
```

```
    <MyContext.Provider value={contextValue}>
      {children}
    </MyContext.Provider>
  );
};

// Custom Hook for easier useContext
const useMyContext = () => {
  const context = useContext(MyContext);
  if (context === undefined) {
    throw new Error('useMyContext must be used within a MyProvider');
  }
  return context;
};

export { MyProvider, useMyContext };
```

This code creates a context called `MyContext` and a provider component (`MyProvider`). The provider holds the global state (`globalState`) and a function to update it (`updateGlobalState`). The `useMyContext` hook simplifies accessing the context.

2. Wrap your application with the Provider:

Wrap your application's root component with the `MyProvider` to make the context available to all its descendants:

```javascript
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { MyProvider } from './MyContext';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <MyProvider>
      <App />
    </MyProvider>
  </React.StrictMode>
);
```

3. Consume the Context:

In any component that needs access to the global state, use the `useMyContext` hook:

```javascript
import React from 'react';
import { useMyContext } from './MyContext';

function MyComponent() {
```

```
  const { globalState, updateGlobalState } = useMyContext();

  const changeTheme = () => {
    updateGlobalState({ theme: globalState.theme === 'light' ? 'dark' : 'light' });
  };

  return (
    <div>
      <h1>Hello, {globalState.userName}!</h1>
      <p>Theme: {globalState.theme}</p>
      <button onClick={changeTheme}>Change Theme</button>
    </div>
  );
}

export default MyComponent;
```

This component accesses the `globalState` and `updateGlobalState` from the context and uses them to render information and update the theme.

Important Considerations:

* **Performance:** While Context is powerful, excessively large or frequently changing context values can impact performance. Consider using Context selectively for truly global state and explore other state management solutions (like Redux, Zustand, Jotai) for more complex state management needs.

* **Context updates are asynchronous:** Remember that state updates in React are asynchronous. Don't rely on immediately accessing the updated value after calling `updateGlobalState`.
* **Error Handling:** The example includes error handling in `useMyContext` to prevent runtime errors if the component is used outside of the provider.

This comprehensive example demonstrates how to effectively leverage React's Context API for efficient global state management within your React applications. Remember to choose the appropriate state management solution based on your application's complexity and scale.

Q: 8. What is the purpose of keys in React lists, and why are they important?
A: In React, keys are used to uniquely identify elements within an array that is being rendered as a list. They're crucial for React's efficient rendering and updating of lists. Without them, React's reconciliation algorithm can behave unpredictably, leading to performance issues and incorrect UI updates.

Here's a breakdown of their purpose and importance:

Purpose:

Keys act as a hint to React's reconciliation algorithm. When the list changes (items are added, removed, or reordered), React uses the keys to efficiently determine what has changed. Instead of re-rendering the entire list, React can intelligently update only the necessary parts.

Importance:

* **Efficient Updates:** Without keys, React has to resort to a much more expensive comparison process. It essentially compares every item in the old list to every item in the new list to figure out what has changed. This is O(n^2) complexity, meaning the performance degrades drastically as the list grows. With keys, React can perform a much more efficient O(n) comparison. It can quickly identify elements by their keys and update only the necessary elements.

* **Correct UI:** Incorrect updates can lead to inconsistencies in the UI. For example, if you're displaying a list of items, and an item is moved, without keys, React might not correctly reflect the change in position, or might reuse old state associated with a previous element in the wrong place. This can lead to unexpected behavior and bugs.

* **Component State Preservation:** React uses keys to track which components represent which data. If the data is rearranged in the list, React can preserve the state of components. If a component doesn't have a unique key, React may reuse the component, and therefore its state, in the wrong place.

Example:

Consider a list of to-dos. If you add a new to-do item, without keys, React will have difficulty determining whether the new item is truly new or just a reordered item. Using the `id` of each to-do as a key resolves this ambiguity.

Choosing Keys:

Ideally, keys should be unique, stable identifiers for each item in the list. Good

candidates include:

* **Database IDs:** If your data comes from a database, use the database ID.
* **Unique Identifiers:** Generate unique identifiers using a library like UUID.
* **Index as a last resort:** Only use the index as a key if the items in the list are never reordered, added, or removed and are truly static. Using index as a key can lead to unexpected behaviour and is generally discouraged.

In short, using keys in React lists is a best practice that significantly improves performance, ensures correctness, and simplifies maintenance. Always provide a unique key for each item in your list.

Q: 9. What is the difference between useState and useReducer ? When would you use one over
A: Both `useState` and `useReducer` are Hooks in React used for managing state, but they differ significantly in how they handle state updates:

useState:

* **Simple State Management:** `useState` is ideal for managing simple state values that don't involve complex logic or multiple interdependent sub-states. It directly updates a single state variable.
* **Direct Updates:** Updates are made directly using the setter function returned by `useState`. This is straightforward for simple scenarios.
* **Immutability:** While you might use the updater function with a callback to handle updates in a functional way (i.e. `setState(prevState => ({...prevState, count: prevState.count + 1}))`, it's generally simpler than `useReducer` for single-state

management.

useReducer:

* **Complex State Management:** `useReducer` is designed for managing more complex state that involves multiple values or requires complex logic to update. It uses a reducer function to manage the state.
* **Reducer Function:** The state updates happen through a reducer function which takes the current state and an action as input and returns the *new* state. This approach is functional and promotes better code organization and predictability for complex state interactions.
* **Centralized Logic:** The reducer function centralizes the state update logic, making it easier to reason about and test. This is advantageous when many different components might modify the same state variable in different ways.
* **Action Types:** Using a defined set of action types can improve readability and maintainability; the reducer determines how the state changes based on the action type dispatched.

When to Use Which:

* **`useState`:** Use `useState` when you have simple state that can be managed with direct updates. If the state is a single value or a few values that don't depend on each other, `useState` offers a simple and easy-to-understand approach. Examples include:
    * A form input value
    * A boolean flag for a component's visibility

    * A counter

* **`useReducer`:** Use `useReducer` when:
    * Your state is complex and has multiple interdependent values.
    * You have multiple ways to update the state, and those updates involve complex logic.
    * You need a centralized and easily testable way to manage state transitions.
    * You want to improve code readability and maintainability for more complex state changes. Examples include:
    * Managing a shopping cart with multiple items, quantities, and totals
    * Handling game state with many variables interacting with each other (e.g., player position, score, game over status)
    * Implementing undo/redo functionality

Example (Counter):

useState:

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

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

  return (
```

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

useReducer:

```javascript
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();
  }
}

function 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>
  );
}
```

In summary, choose `useState` for simplicity and `useReducer` for managing complex state with cleaner, more maintainable code. For very simple state, `useState` is often the preferred and more efficient approach. But as complexity grows, `useReducer` becomes increasingly beneficial.

Q: 10. What are React fragments, and why would you use them instead of a wrapper element like

A: In React, fragments are a way to group a list of children without adding extra nodes to the DOM. They're represented by empty tags `<></>` or, for keyed fragments (useful when you need to identify fragments individually), `<Fragment>`.

Why use fragments instead of a wrapper element?

Let's say you have a component that renders several elements:

```jsx
function MyComponent() {
  return (
    <div> {/* Wrapper div */}
      <p>Paragraph 1</p>
      <p>Paragraph 2</p>
      <ul>
        <li>Item 1</li>
        <li>Item 2</li>
      </ul>
    </div>
  );
}
```

Here, a `<div>` acts as a wrapper to group the child elements. While this works, the `<div>` adds an unnecessary node to the DOM. This can be insignificant in small applications but can lead to performance issues or unintended styling consequences in large, complex apps.

Using a fragment avoids this:

```jsx
function MyComponent() {
  return (
    <> {/* Fragment */}
      <p>Paragraph 1</p>
```

```
      <p>Paragraph 2</p>
      <ul>
        <li>Item 1</li>
        <li>Item 2</li>
      </ul>
    </>
  );
}
```

or with the named fragment:

```jsx
function MyComponent() {
  return (
    <Fragment> {/* Named Fragment */}
      <p>Paragraph 1</p>
      <p>Paragraph 2</p>
      <ul>
        <li>Item 1</li>
        <li>Item 2</li>
      </ul>
    </Fragment>
  );
}
```

The key benefits are:

* **Cleaner DOM:** Fragments don't appear in the rendered HTML, resulting in a cleaner and potentially more efficient DOM tree. This improves performance, especially with many nested components.
* **Improved readability:** Grouping related elements without unnecessary wrapper elements enhances the readability of your JSX. It makes the component's structure clearer.
* **Avoiding unnecessary styling:** If you're not using the wrapper element for styling or any other specific purpose, avoiding it prevents potential unintended styling conflicts.

In short, use fragments when you need to group JSX elements without introducing extra nodes into the actual DOM. They're a powerful tool for keeping your React code clean and efficient.