Q.1 What are hooks in React? How to identify hooks?

Hooks are functions introduced in React 16.8 that allow functional components to have state and access lifecycle methods. They provide a way to use state and other React features without writing a class. Hooks are identified by the naming convention starting with the prefix "use" (e.g., `useState`, `useEffect`, `useContext`). React provides built-in hooks and also allows creating custom hooks.

Q.2 Explain `useState` Hook & what can you achieve with it?

The `useState` hook is a built-in hook in React that allows functional components to manage state. It returns an array with two elements: the current state value and a function to update that state. By calling the state update function, React re-renders the component, reflecting the updated state.

With `useState`, you can achieve state management within functional components, allowing them to hold and update their own state.

Q.3 How to pass data from one component to another component?

In React, data can be passed from one component to another using props. The parent component can pass data as props when rendering the child component. The child component can then access the passed data through its props.

In the above example, the `data` variable in the parent component is passed as a prop to the `ChildComponent` by using the `data={data}` syntax. The child component can access the data through `props.data` and render it.

Q.4 What is the significance of the "key" prop in React lists, and why is it important to use it correctly?

In React, when rendering lists of components, it's important to assign a unique "key" prop to each item in the list. The "key" prop helps React identify which items have changed, been added, or been removed when updating the list.

The significance of the "key" prop in React lists:

- Efficient Updates: The "key" prop helps React optimize list updates by efficiently determining which items need to be modified or re-rendered. It allows React to identify specific items in the list and update them without affecting other items.

- Stable Component Instances: The "key" prop helps maintain component state and prevent unnecessary component re-mounting. When a list is re-rendered, React tries to match components with their previous instances based on the "key" prop. This ensures that component state is preserved correctly.

To use the "key" prop correctly, make sure:

- The "key" values are unique within the list.

- The "key" values remain consistent for the same item across re-renders (preferably using a unique identifier from the data).

Q.5 What is the significance of using "setState" instead of modifying state directly in React?

In React, it is important to use the `setState` method provided by React instead of modifying the state directly. Here's why:

- Triggering Re-render: React components re-render when their state changes. By using `setState`, React is made aware of the state change, and it can schedule the re-rendering of the component and its child components, optimizing performance.

- Immutability and State Updates: React enforces the concept of immutability when updating state. Directly modifying state can lead to unexpected behavior, as React relies on comparing the current and previous state to determine what needs to be updated in the component tree.

- Batched Updates: React batches multiple `setState` calls together for better performance. It means that if you call `setState` multiple times within the same event handler or lifecycle method, React will perform a single re-render instead of multiple individual renders.

Using `setState` correctly ensures proper state management, component updates, and avoids potential issues related to state mutation and re-rendering.

Q.6 Explain the concept of React fragments and when you should use them.

React fragments allow you to group multiple children elements without adding an extra wrapping element. They are useful when you need to return multiple elements from a component, but you don't want to introduce unnecessary parent elements in the DOM.

Instead of using a single wrapping element like a `<div>`, you can use a fragment to group the elements together. Fragments do not create an extra DOM node.

Use React fragments when:

- You need to return multiple elements from a component without introducing an additional wrapping element.
- You want to avoid unnecessary nesting in the component hierarchy.
- You want to improve the readability and cleanliness of your JSX code.

Q.7 How do you handle conditional rendering in React?

Conditional rendering in React is handled by using JavaScript expressions or conditional operators within JSX code.

Examples of conditional rendering in React:

- Using if-else statements:

```
function MyComponent({ isLoggedIn }) {
  if (isLoggedIn) {
    return <p>Welcome, user!</p>;
  } else {
    return <p>Please log in.</p>;
  }
}
```

- Using ternary operator:

```
function MyComponent({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn ? <p>Welcome, user!</p> : <p>Please log in.</p>}
    </div>
  );
}
```

- Using logical && operator for simple conditionals:

```
function MyComponent({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn && <p>Welcome, user!</p>}
    </div>
  );
}
```

These are just a few examples of how you can conditionally render components or elements based on certain conditions in React. The approach you choose depends on the complexity and requirements of your conditional rendering logic.