**Q.1** Whats React and its pros and cons?

React is a popular JavaScript library used for building user interfaces. It provides a component-based architecture that allows developers to create reusable UI components and efficiently manage the application state. Here are some pros and cons of using React:

Pros of React:

1. Component-Based Architecture: React promotes a modular approach to building user interfaces. Components encapsulate their own logic, styles, and state, making them reusable and easier to maintain. This modularity enhances code reusability and allows for better organization of the application's UI.

2. Virtual DOM: React utilizes a virtual representation of the DOM (Document Object Model) called the Virtual DOM. This approach enables efficient rendering by updating only the necessary parts of the UI. React compares the previous and current states of the Virtual DOM and applies the minimum required changes to the actual DOM, resulting in improved performance.

3. Declarative Syntax: React uses a declarative syntax, which means you describe the desired UI state, and React takes care of updating the actual DOM to match that state. This approach makes it easier to understand and reason about the code, as you focus on the "what" rather than the "how."

4. Community and Ecosystem: React has a large and active community, which leads to a thriving ecosystem of third-party libraries, tools, and resources. This community support makes it easier to find solutions, get help, and stay up-to-date with the latest trends in React development.

Cons of React:

1. Learning Curve: React has a learning curve, especially for developers who are new to component-based architectures or JSX (a syntax extension used in React). Understanding React's concepts and best practices may require some initial effort.

2. Boilerplate: While React itself is lightweight, building a full-featured React application may involve setting up additional tooling and libraries, which can introduce some initial setup complexity.

3. Lack of Opinionated Structure: React doesn't provide a strict architectural pattern or enforce a specific project structure. While this flexibility allows developers to choose the best architecture for their application, it can also result in inconsistency across projects if not properly managed.

4. Performance Overhead: React's Virtual DOM introduces some performance overhead compared to directly manipulating the DOM. Although React is optimized to minimize this overhead, certain use cases with extensive UI updates or complex rendering may still require additional optimizations.

It's important to note that the cons mentioned above are relative and may not be significant issues for many developers. React's benefits often outweigh the drawbacks, and its popularity is a testament to its effectiveness in building scalable and maintainable user interfaces.

**Q.2** What do you understand by Virtual Dom?

The Virtual DOM (Document Object Model) is a concept used by React to improve the performance and efficiency of updating the user interface (UI). It is an abstraction or lightweight copy of the actual DOM, a tree-like structure that represents the HTML elements of a web page.

In React, when a component's state or props change, instead of directly updating the real DOM, React first updates the Virtual DOM. This process involves creating a new Virtual DOM tree that reflects the desired UI state after the changes.

Here's a simplified overview of how the Virtual DOM works in React:

1. Render Virtual DOM: React renders the components into a Virtual DOM representation, which is a JavaScript object hierarchy that mirrors the structure of the real DOM.

2. Compare with Previous Virtual DOM: When a component's state or props change, React generates a new Virtual DOM tree.

3. Diffing: React performs a process called "diffing" or reconciliation by comparing the previous Virtual DOM tree with the new one. It identifies the differences or updates required to bring the Virtual DOM in line with the desired UI state.

4. Update Real DOM: Once the differences are identified, React updates only the necessary parts of the actual DOM to reflect the changes. This selective updating is more efficient than re-rendering the entire DOM.

The Virtual DOM allows React to minimize the number of actual manipulations on the real DOM, which can be a costly operation in terms of performance. By leveraging the Virtual DOM, React is able to optimize the update process by calculating and applying only the necessary changes to the UI.

This approach improves performance by reducing the number of expensive operations involved in directly manipulating the DOM. Additionally, React's batched update process further enhances performance by grouping multiple updates together and applying them in a single pass.

Overall, the Virtual DOM is a key mechanism in React that contributes to its efficiency, performance, and ability to build complex and dynamic user interfaces.

**Q.3** Difference between Virtual Dom vs Real Dom?

The main difference between the Virtual DOM (VDOM) and the Real DOM lies in how they represent and update the user interface (UI) of a web application.

Virtual DOM:

1. Representation: The Virtual DOM is an abstraction or lightweight copy of the actual DOM. It is a JavaScript object hierarchy that mirrors the structure of the real DOM.

2. Updates: When a change occurs in the UI, React updates the Virtual DOM instead of directly manipulating the real DOM. This update process involves creating a new Virtual DOM tree that reflects the desired UI state.

3. Diffing: React performs a process called "diffing" or reconciliation, where it compares the previous Virtual DOM tree with the new one to identify the differences or updates required to bring the Virtual DOM in line with the desired UI state.

4. Selective Updating: Once the differences are identified, React updates only the necessary parts of the actual DOM to reflect the changes. It minimizes the number of actual manipulations on the real DOM, optimizing the update process.

5. Performance Benefits: The Virtual DOM helps improve performance by reducing the number of expensive operations involved in directly manipulating the real DOM. React calculates and applies only the necessary changes to the UI, reducing the impact on performance.

Real DOM:

1. Representation: The Real DOM is the actual tree-like structure that represents the HTML elements of a web page. It is the browser's internal representation of the rendered UI.

2. Updates: When a change occurs in the UI, traditional JavaScript approaches directly manipulate the Real DOM to reflect the updated state. This involves finding, modifying, adding, or removing individual DOM nodes.

3. Full Re-Rendering: In the Real DOM approach, making changes to the UI typically involves re-rendering the entire DOM tree, even if only a small part of it needs to be updated. This can lead to performance bottlenecks in complex applications.

4. Performance Implications: Directly manipulating the Real DOM can be an expensive operation, especially when dealing with a large number of DOM elements or frequent UI updates. Changes to the Real DOM trigger layout and repaint operations, which can impact performance.

In summary, the Virtual DOM provides an efficient and optimized way of updating the UI by introducing a lightweight representation of the DOM that minimizes direct manipulation of the real DOM. This approach helps improve performance and enables React to efficiently handle complex UI updates with minimal impact on the actual DOM.

**Q.4** Whats component? Types of component?

In React, a component is a reusable and self-contained piece of UI that encapsulates logic, structure, and styling. Components are the building blocks of a React application, allowing you to create modular and maintainable user interfaces. React applications are typically composed of multiple components working together to form the complete UI.

There are two types of components in React:

1. Functional Components:

- Also known as stateless components or presentational components.

- Implemented as JavaScript functions that accept props (input data) as arguments and return JSX (HTML-like syntax).

- They focus on presenting UI based on the given props and do not have their own state.

- Functional components are simpler and more lightweight compared to class components.

- With the introduction of React Hooks in React 16.8, functional components gained the ability to have local state and lifecycle methods using hooks.

Example of a functional component:

import React from 'react';

const MyComponent = (props) => {

return <div>Hello, {props.name}!</div>;

};

export default MyComponent;

2. Class Components:

- Also known as stateful components or container components.

- Implemented as ES6 classes that extend the `React.Component` class.

- They have their own internal state and can manage it using lifecycle methods like `componentDidMount`, `componentDidUpdate`, etc.

- Class components are suitable for more complex logic and managing stateful behavior.

- They can hold local state and also accept props.

Example of a class component:

import React, { Component } from 'react';

class MyComponent extends Component {

constructor(props) {

super(props);

this.state = { count: 0 };

}

incrementCount = () => {

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

};

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={this.incrementCount}>Increment</button>

</div>

);

}

}

export default MyComponent;

Both functional and class components can be used interchangeably in React applications. However, functional components are becoming more popular due to their simplicity and improved performance with the use of React Hooks. It's recommended to use functional components by default and reserve class components for cases that require complex state management or lifecycle methods.

**Q.5** Difference between class & function based component?

Here are the key differences between class-based components and function-based components in React:

Class-based Components:

1. Syntax: Class-based components are implemented as ES6 classes that extend the `React.Component` class.

2. State Management: They have their own internal state managed through `this.state` and can update it using `this.setState()`.

3. Lifecycle Methods: Class components have access to lifecycle methods such as `componentDidMount`, `componentDidUpdate`, `componentWillUnmount`, etc. These methods allow you to hook into different phases of a component's lifecycle and perform actions accordingly.

4. More Complex Logic: Class components are suitable for components that require more complex logic, state management, and access to lifecycle methods.

5. Component Hierarchy: Class components can define child components within their `render()` method using JSX, allowing for a nested component hierarchy.

Example of a class component:

```jsx

import React, { Component } from 'react';

class MyComponent extends Component {

constructor(props) {

super(props);

this.state = { count: 0 };

}

incrementCount = () => {

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

};

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={this.incrementCount}>Increment</button>

</div>

);

}

}

export default MyComponent;

```

Function-based Components:

1. Syntax: Function-based components are implemented as JavaScript functions. They accept `props` as input arguments and return JSX.

2. No Internal State: Function components were initially considered stateless and did not have their own internal state. However, with the introduction of React Hooks in React 16.8, function components gained the ability to have local state and lifecycle methods using hooks.

3. Simplicity and Readability: Function components are generally simpler and more lightweight compared to class components. They have a cleaner syntax and are easier to read and understand.

4. Improved Performance: Function components with React Hooks offer improved performance by eliminating the need for class-based overhead and optimizing rendering processes.

5. Reusability: Function components are highly reusable and can be easily composed to create complex UI structures by leveraging the concept of composition and custom hooks.

Example of a function component:

import React from 'react';

const MyComponent = (props) => {

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

const incrementCount = () => {

setCount(count + 1);

};

return (

<div>

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

<button onClick={incrementCount}>Increment</button>

</div>

);

};

export default MyComponent;

Overall, while class-based components provide more advanced features and control through state management and lifecycle methods, function-based components, especially with React Hooks, have become the preferred choice due to their simplicity, improved performance, and reusability. It's recommended to use function-based components by default and use class-based components when necessary for specific use cases.

**Q.6** Explain react component life cycle?

The React component lifecycle refers to the series of phases that a component goes through from its creation to its removal from the DOM. Each phase of the lifecycle offers specific methods that allow you to hook into and perform actions at different stages of a component's existence. The React component lifecycle can be divided into three main phases:

1. Mounting Phase:

- The mounting phase occurs when a component is being initialized and inserted into the DOM.

- Methods used in this phase:

- `constructor()`: Called when the component is created, used for initializing state and binding event handlers.

- `static getDerivedStateFromProps()`: Invoked just before rendering when new props are received. Used to update the state based on the new props.

- `render()`: Required method that returns JSX to be rendered to the DOM.

- `componentDidMount()`: Called after the component has been rendered to the DOM. Used for performing side effects such as API calls or setting up event listeners.

2. Updating Phase:

- The updating phase occurs when a component is re-rendered due to changes in its props or state.

- Methods used in this phase:

- `static getDerivedStateFromProps()`: Similar to the mounting phase, it is called just before rendering when new props are received.

- `shouldComponentUpdate()`: Called before rendering to determine whether the component should re-render. Used for performance optimization by avoiding unnecessary re-renders.

- `render()`: Required method that returns JSX to be rendered to the DOM.

- `getSnapshotBeforeUpdate()`: Called right before changes from the virtual DOM are reflected in the actual DOM. Used for capturing information from the DOM (e.g., scroll position) before it updates.

- `componentDidUpdate()`: Called after the component has been updated and re-rendered. Used for performing side effects like updating the DOM or fetching new data based on prop/state changes.

3. Unmounting Phase:

- The unmounting phase occurs when a component is being removed from the DOM.

- Method used in this phase:

- `componentWillUnmount()`: Called just before the component is unmounted and destroyed. Used for cleanup tasks such as removing event listeners or canceling timers.

In addition to these main phases, React also provides error handling methods that can be used to catch and handle errors that occur during rendering, updates, or lifecycle methods. These methods are `static getDerivedStateFromError()` and `componentDidCatch()`.

It's important to note that with the introduction of React Hooks, functional components can also utilize lifecycle-like functionality using the `useEffect()` hook, which allows you to perform side effects and cleanup operations. Hooks provide a more streamlined and declarative approach to handling component lifecycle in function-based components.

Understanding the React component lifecycle is crucial for managing component behavior, state, and performing actions at specific points in a component's existence. However, it's worth noting that some lifecycle methods, such as `componentWillMount()` and `componentWillReceiveProps()`, have been deprecated or removed in recent versions of React, so it's essential to refer to the React documentation for the most up-to-date information on lifecycle methods.

**Q.7** Explain Prop Drilling in React & Ways to avoid it?

Prop drilling in React refers to the process of passing props through multiple levels of nested components to reach a deeply nested component that needs access to those props. It occurs when intermediate components in the component tree don't need the props but have to receive and pass them down to their child components.

Prop drilling can become cumbersome and lead to code complexity and reduced maintainability. Here are some ways to avoid prop drilling:

1. Use React Context:

- React Context provides a way to share data between components without explicitly passing props through each level.

- You can create a context using the `React.createContext()` function and wrap the relevant components with a `Context.Provider` component to make the data accessible to them.

- Child components can then access the context data using the `Context.Consumer` component or the `useContext()` hook.

2. Use Redux:

- Redux is a state management library that centralizes the application state in a single store.

- It provides a global state accessible to all components, eliminating the need for prop drilling.

- Components can subscribe to the Redux store and access the required data using selectors.

3. Use React's Render Props pattern:

- The Render Props pattern involves passing a function as a prop to a component, allowing that component to render content using that function.

- By utilizing this pattern, you can avoid prop drilling by directly passing the required data and functions to the child component, bypassing intermediate components.

4. Use React custom Hooks:

- Custom Hooks enable you to encapsulate reusable logic and state in a custom function.

- By extracting common logic into a custom Hook, you can provide the required data and functions to any component that needs it, without the need for prop drilling.

5. Component Composition:

- Instead of passing data through intermediate components, you can compose components in a way that allows them to be self-contained and have their own data requirements.

- Break down your UI into smaller, more focused components that encapsulate their own logic and state, reducing the need for passing props through unnecessary levels.

6. Higher-Order Components (HOCs):

- HOCs are functions that take a component and return an enhanced version of that component.

- You can use HOCs to inject specific props into components, avoiding the need for manual prop drilling.

By utilizing these techniques, you can minimize or eliminate prop drilling in your React applications, leading to cleaner and more maintainable code. The choice of the approach depends on the complexity of your application, the nature of the shared data, and the specific requirements of your project.