React JS Questions

▼ 1. What is HOC with example?

A higher-order component is a function that takes a component and returns a new component.

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

const HigherOrderComponent = (Component, incrementValue = 0) => {

  const HOCFun = () => {
    const [value, setValue] = useState(0);
    const incrementHandler = () => {
      setValue(value + incrementValue);
    };

    return <Component value={value} incrementHandler={incrementHandler} />;
  };

  return HOCFun;
};

export default HigherOrderComponent;
```

```
import HigherOrderComponent from "./HigherOrderComponent";

const ComponentA = ({ value, incrementHandler }) => {
```

```
  return (
    <div>
      <button onClick={incrementHandler}>Increment by 2</button>
      <h2>{value}</h2>

    </div>
  );
};

export default HigherOrderComponent(ComponentA, 2);
```

▼ 2. **Why we use React? advantage of SPA?**

React is very Good at 3 things:

- Smooth(UX) - they offer a smooth user experience by not having to reload the entire page during navigation.
- Great Time to Interactivity - It uses Virtual DOM which makes rendering fast and hence there is increase performance.
- Scale - they scale seamlessly as everything takes place on the Client side.

▼ 3. **React Context with example.**

Sharing data among components without using props.

Here sharing data from A→B→(C, D)

1. Create a Context

```
import React from "react";

const DemoContext = React.createContext();

export default DemoContext;
```

2. Then, using this context, wrap the component tree in a Provider.

```
import DemoContext from "../DemoContext";
import B from "./B";

const A = () => {
  const obj = {
    a: 1,
    b: 2,
    c: 3,
```

```
  };
  return (
    <DemoContext.Provider value={{ obj }}>
      <div>
        <B />
      </div>
    </DemoContext.Provider>
  );
};

export default A;
```

3. Now, we can access the “obj” in components “C”. There are two ways for consuming the context - by using the Consumer and useContext hook. Prefer using the useContext hook because it is the modern and better way.

```
import React, { useContext } from "react";
import DemoContext from "../DemoContext";

const C = () => {
  const { obj } = useContext(DemoContext);
  const { a, b, c } = obj;

  return (
    <div>
      <h2>Component C</h2>
      <h3>{a}</h3>
      <h3>{b}</h3>
      <h3>{c}</h3>
    </div>
  );
};

export default C;
```

▼ 4. **What are React Hooks?**

React Hooks are a new addition in React version 16.8. They let you use state and other React features without converting functional components to a class.

Hooks does the same job with less code and with less code means less chances of producing bugs and increased performance.

Basic Hooks

useState

- returns a stateful value, and a function to update it.

useEffect

- lets us perform side effects in function components

useContext

- gives a simple function to access the data via value prop of the Context Provider in any child component

Additional Hooks

useReducer

- state management like redux for managing state in smaller applications rather than having to reach for a third-party state management library

useCallback

- memoizes callback functions, so they not recreated on every re-render.

useMemo

- stores the results of expensive operations

useRef

- lets us perform side effects in function components

useImperativeHandle

- Used together with forwardRef which allows you to modify the ref instance that is exposed from parent components

useLayoutEffect

- this runs synchronously immediately after React has performed all DOM mutations

useDebugValue

- allows you to display additional, helpful information next to your custom Hooks, with optional formatting.

▼ **5. How to pass data between components?**

1. To pass data from parent to child, use props
2. To pass data from child to parent, use callbacks
3. To pass data among siblings AND anywhere else
    a. use React's Context API also
    b. use state management libraries for mid - big sized applications that are stateful. **Example:** Redux, MobX, and Recoil

▼ **6. What are some limitations of React.**

First, JSX can make the coding complex. It will have a steep learning curve for the beginners

Second, React documentation is not user friendly and thorough as it should be.

Third, every React project are unique to engineers as they will rely on numerous technologies to incorporate in their projects.

▼ **7. What is prop drilling and how can you avoid it?**

Prop Drilling is the process by which data is passed from one component to deeply nested components. This becomes a problem as other components will contain data that they don't need.

Also, It will make the components hard to maintain.

A common approach to avoid prop drilling is to use React context and state management libraries.

Few disadvantage of prop drilling

1. Components that should not otherwise be aware of the data become unnecessarily complicated
2. Harder to maintain.

▼ **8. What is the use of dangerouslySetInnerHTML?**

This property is React's replacement for using innerHTML in the browser. It will render raw HTML in a component.

> One should limit its use because it can expose users to potential cross-site scripting attacks.

▼ 9. Name a few techniques to optimize React app performance.

First, Use React.Suspense and React.Lazy for Lazy Loading Components. This will only load component when it is needed.

```
import LazyComponent from './LazyComponent';

const LazyComponent = React.lazy(() => import('./LazyComponent'));
```

Second, Use React.memo for Component Memoization**React.memo** is a higher order component that will render the component and memoizes the result. Before the next render, if the new props are the same, React reuses the memoized result skipping the next rendering.

```
import React from 'react';

const MyComponent = React.memo(props =>  {
  /* render only if the props changed */
});
```

> Note: If React.memo has
> a useState, useReducer or useContext Hook in its implementation, it will still re-render when state or context change.

The more often the component renders with the same props,the heavier and the more computationally expensive the output is, the more chances are that component needs to be wrapped in React.memo().

Third, Use **React.Fragment** to Avoid Adding Extra Nodes to the DOM React Fragments do not produce any extra elements in the DOM Fragment's child

components will be rendered without any wrapping DOM node.

This is a cleaner alternative rather than adding divs in the code.

```
function App() {
  return (
    <React.Fragment>
      <h1>Best App</h1>
      <p>Easy as pie!</p>
    </React.Fragment>
  );
}
```

▼ 10. What is reconciliation?

When a component's props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM. This process is called reconciliation.

▼ 11. Why do we need to pass a function to setState()?

setState() is an asynchronous operation. React batches state changes for performance reasons. This means state may not change immediately after setState() is called.

```
// Wrong
this.setState({
  counter: this.state.counter + 1
})

// Correct
this.setState((prevState) => ({
  counter: prevState.counter + 1
}))
```

▼ 12. Why should we not update the state directly?

Updating the state directly, like below will not cause the component to re-render.

Instead, use setState() method. This method will schedule an update to a component's state object. When state changes, the component responds by re-rendering.

▼ 13. What are Controlled and Uncontrolled Component.

A Controlled Component is one that takes a value through props and notify changes through callbacks like `onChange` or `onClick`.

Form data is handled by React component.

Uncontrolled Component is one that stores its own state internally, and queries the DOM using a ref or reference to find the current value when it is needed.

Form data is handled by the DOM.

In most cases, Controlled components are recommended to be used when implement forms.

▼ 14. What is the use of refs?

The ref is used to return a reference to the element. They can be useful when you need direct access to the DOM element or an instance of a component.

▼ 15. What are the difference between a class component and functional component?

Class Components

- Class-based Components uses ES6 class syntax. It can make use of the lifecycle methods.
- Class components extend from React.Component.
- In here you have to use this keyword to access the props and functions that you declare inside the class components.

Functional Components

- Functional Components are simpler comparing to class-based functions.
- Functional Components mainly focuses on the UI of the application, not on the behavior.
- To be more precise these are basically render function in the class component.

- Functional Components can have state and mimic lifecycle events using react Hooks

▼ 16. What is Pure Component?

`React.PureComponent` is exactly the same as `React.Component` except that it handles the `shouldComponentUpdate()` method for you. When props or state changes, PureComponent will do a shallow comparison on both props and state. Component on the other hand won't compare current props and state to next out of the box. Thus, the component will re-render by default whenever shouldComponentUpdate is called.

▼ 17. What are portals in React?

Portal is a recommended way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

```
ReactDOM.createPortal(child, container);
```

The first argument is any render-able React child, such as an element, string, or fragment. The second argument is a DOM element.

▼ 18. How do you memoize a component?

Since React v16.6.0, we have a `React.memo`. It provides a higher order component which memoizes component unless the props change. To use it, simply wrap the component using `React.memo` before you use it.

```
const MemoComponent = React.memo(function MemoComponent(props) {
    /* render using props */
});

// OR

export default React.memo(MyFunctionComponent);
```

▼ 19. What are render props?

Render Props is a simple technique for sharing code between components using a prop whose value is a function. The below component uses render prop which returns a React element.

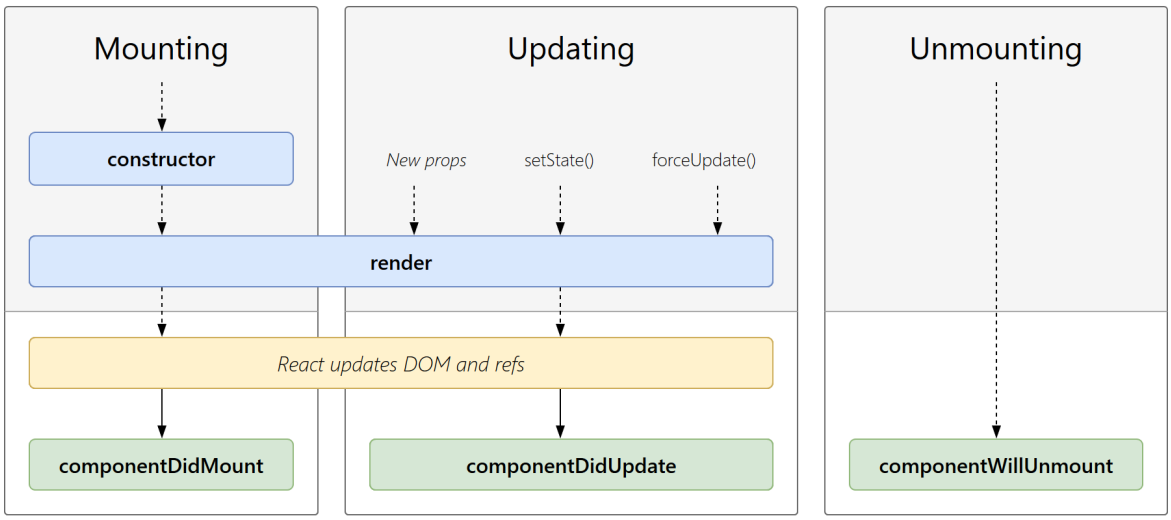
```
<DataProvider render={(data) => <h1>{`Hello ${data.target}`}</h1>} />
```

▼ 20. What is React memo?

Class components can be restricted from rendering when their input props are the same using **PureComponent or shouldComponentUpdate**. Now you can do the same with function components by wrapping them in **React.memo**.

```
const MyComponent = React.memo(function MyComponent(props) {
  /* only rerenders if props change */
});
```

▼ 21. Component Lifecycle Methods (in Order)



▼ ErrorBoundaries in React

Error boundaries are a basic fallback for a webapp / website if any error occurs (one way of error handling in reactjs)

```
//Sample Error Boundary component
```

```
import React from 'react';

import {
  ErrorImageOverlay,
  ErrorImageContainer,
  ErrorImageText
} from './error-boundary.styles';

class ErrorBoundary extends React.Component {
  constructor() {
    super();

    this.state = {
      hasErrored: false
    };
  }

  static getDerivedStateFromError(error) {
    // process the error
    return { hasErrored: true };
  }

  componentDidCatch(error, info) {
    console.log(error);
  }

  render() {
    if (this.state.hasErrored) {
      return (
        <ErrorImageOverlay>
          <ErrorImageContainer imageUrl='https://i.imgur.com/yW2W9SC.png' />
          <ErrorImageText>Sorry this page is broken</ErrorImageText>
        </ErrorImageOverlay>
      );
    }

    return this.props.children;
  }
}

export default ErrorBoundary;
```

We will just wrap this error boundary component in our parent component

```
//Example of a parent component where errorboundary is wrapped

const App = ({ checkUserSession, currentUser }) => {
  useEffect(() => {
    checkUserSession();
```

```
  }, [checkUserSession]);

  return (
    <div>
      <GlobalStyle />
      <Header />
      <Switch>
        <ErrorBoundary>
          <Suspense fallback={<Spinner />}>
            <Route exact path='/' component={HomePage} />
            <Route path='/shop' component={ShopPage} />
            <Route exact path='/checkout' component={CheckoutPage} />
            <Route
              exact
              path='/signin'
              render={() =>
                currentUser ? <Redirect to='/' /> : <SignInAndSignUpPage />
              }
            />
          </Suspense>
        </ErrorBoundary>
      </Switch>
    </div>
  );
};
```

▼ What are common use cases for the `useMemo`?

The primary purpose of `useMemo` hook is *"performance optimization"*.

- It returns a *memoized value*,
- It accepts *two arguments* - ***create*** *function* (which should return a value to be memoized) and ***dependency*** *array*. It will *recompute the memoized value only when one of the dependencies has changed.*

Using `useMemo` you achieve:

- referential equality of the values (to further send them to props of the components to potentially avoid re-renders)
- eliminate redoing of the computationally expensive operations for same parameters

```
function App() {
    const [data, setData] = useState([....]);

    function format() {
```

```
        console.log('formatting....'); // this will print only when data has changed
        const formattedData = [];
        data.forEach(item => {
            const newItem = //...do soemthing here,
            if (newItem) {
                formattedData.push(newItem);
            }
        })
        return formattedData;
    }

    const formattedData = useMemo(format, [data])

    return (
        <>
        {formattedData.map(item => (
            <div key={item.id}>
            {item.title}
            </div>
        ))}
        </>
    )
}
```

▼ What are production use cases for the `useRef`?

- `useRef` simply returns a *plain Javascript object*. Its value can be *accessed* and *modified* (mutability) as many times as you need without worrying about *rerender*.
- `useRef` value will *persist* (won't be reset to the *initialValue* unlike an ordinary object defined in your function component; it persists because `useRef` gives you the same object *instead of creating a new one* on subsequent renders) for the component lifetime and across re-renders.
- `useRef` hook is often used to store values instead of DOM references. These values can either be a state that does not need to change too often or a state that should change as frequently as possible but should *not trigger* full re-rendering of the component.

```
const refObject = useRef(initialValue);
```

▼