React Components , States & Props

1). Explain Life cycle in Class Component and functional component with Hooks?

React lifecycle method explained : -

First, let's take a look at how it's been done traditionally. In order to do that, we're going to take a closer look at React components.

As you probably know, each React component instance has a lifecycle. The component's lifecycle consists of three phases:

Mounting lifecycle methods, that is inserting elements into the DOM.

Updating, which involves methods for updating components in the DOM.

Unmounting, that is removing a component from the DOM.

Each phase has its own methods, which make it easier to perform typical operations on the components. With class-based components, React developers directly extend from the React.Component in order to access the methods.

There is no need to address all of the React component lifecycle methods here. The best source of information on currently supported methods for class components is the Official React Documentation. Instead, let's take a closer look at just a couple examples of lifecycle methods. However, before we do, let's quickly address the topic that is likely on your mind right now.

Class Components v/s Functional Components :-

As you probably know, an alternative way of taking advantage of lifecycle methods is to use hooks. With the release of React 16.8 back in March 2019, it is now possible to create functional components that are not stateless and can use lifecycle methods.

It's all thanks to the *useState* and *useEffect* hooks – special functions that hook into React features that allow to set the initial state and use lifecycle events in functional components. Currently, it is possible to emulate the performance of almost any supported lifecycle method by skilfully applying these two hooks in your pure JavaScript functions.

Are hook-enhanced functional components superior to class based ones? Before we get to that, let's go over the lifecycle phases using the traditional approach.

React component lifecycle with hooks :-

You can take advantage of the *useEffect* hook to achieve the same results as with
the *componentDidMount*, *componentDidUpdate* and *componentWillUnmount* methods. *useEffect* accepts two parameters. The first one is a callback which runs **after render**, much like in *componentDidMount*. The second parameter is the effect dependency array. If you want to run it on mount and unmount only, pass an empty array *[]*.

To clean up, return the callback in *useEffect*:

```
useEffect(
() => {
document.addEventListener(“click”, someFunc);

return () => {
document.removeEventListener(“click”, someFunc);
};
},
[]
);
```

If you want it to behave like *componentDidUpdate*, put some dependencies into the array or don't pass the second argument at all.

You can also use *useState* instead of *this.State* in class components. Instead of:

```
this.state = {
        counter: 0,
        usersList: [],
}
```

You can do that:

```
const [counter, setCounter] = useState(0);
const [usersList, setUsersList] = useState([]);
```

As you can, it is possible to use hooks to achieve similar or the same end results.