Component Composition

- Composition is a technique which allows us to combine one or more components into a newer, enhanced component capable of greater behavior. Instead of extending another class or object, composition only aims at adding new behavior.
- By using composition, we can implement other components into the component.
- In programming, composition allows you to build more complex functionality by combining small and focused functions.
- The component composition is one of the most essential patterns in React related to the components model.
- Composition is also a familiar concept in Object Oriented Programming. Instead of inheriting properties from a base class, it describes a class that can reference one or more objects of another class as instances.
- React composition is a pattern that can be used to break a complex component down to smaller components, and then composing those smaller components to structure and complete your application.
- Facebook uses thousands of React components, and they haven't found any use cases where they would recommend creating component inheritance hierarchies.
- We have seen how multiple functions can be composed together to achieve something bigger. The same applies to HTML elements and beyond to React components too.

Composition Vs Inheritance

Composition in ReactJS

- Composition is not OOPS concept.
- Composition is simple & easy to implement.
- Composition is preferred for React Components.
- In Composition, properties & methods are passed from parent to child components.
- In Composition, code is expressed through patterns.

Inheritance in ReactJS

- Inheritance is OOPS concept.
- Inheritance is sometimes complex to implement.
- Inheritance is not recommended for React Components.
- In Inheritance, Child inherits properties & methods from Parent Components.
- In Inheritance, code is expressed through relationships.

Example -- Composing Single Component

```
import React from 'react'
const PrntMessage = () => {
  return(
    <h1>Hello ReactJS</h1>
  )
}
const App = () => {
  return <PrntMessage/>
}
export default App;
```

Example -- Composing Multiple Component

```
import React from 'react'
const FirstComponent = () => {
  return(
    <h1>ReactJS First Component</h1>
  )
}
const SecondComponent = () => {
  return(
    <h1>ReactJS Second Component</h1>
  )
}
const App = () => {
  return(
    <>
      <FirstComponent/>
      <SecondComponent/>
    </>
  )
}
export default App;
```

Props & States

- Components need data to work with. so, in order to maintain data inside components we use "props" & "states".
- There are two different ways that we can combine components and data: either as props or state.
- 'props' and 'state' determine what a component renders and how it behaves.

Props

- The 'props' are used as short for properties.
- 'props' allows us to pass information/ data from one component to another component.
- 'props' are immutable(read-only) objects and cannot be modified from inside the component.
- 'props' is similar to function arguments. Props are passed to the component in the same way as arguments passed in a function.

Example – Simple prop

```
import React from 'react'
const PrintInfo = (props) => {
  return(
    <h1>Hello {props.name}</h1>
  )
}
 const App = () => {
  return (
    <>
      <PrintInfo name="Shahzaib"/>
      <PrintInfo name="Nitin"/>
      <PrintInfo name="Rahul"/>
    </>
  )
}
export default App
```

Example -- Destructuring

```
import React from 'react'
const PrintInfo = ({ name, age }) => {
  return(
    <h1>Hello {name} your age is {age}</h1>
  )}
 const App = () => {
  return (
    <>
      <PrintInfo name="Niharika" age={27}/>
      <PrintInfo name="Rahul" age={28}/>
      <PrintInfo name="Ravi" age={29}/>
    </>
  )
}
export default App
```

Example -- Destructuring

```
import React from 'react'
const PrintInfo = (props) => {
  const { name, age } = props
            return(
              <h1>Hello {name} your age is {age}</h1>
            )
}
 const App = () => {
  return (
    <>
      <PrintInfo name="Niharika" age={27}/>
      <PrintInfo name="Rahul" age={28}/>
      <PrintInfo name="Ravi" age={29}/>
    </>
  )
}
```

```
export default App
```

Example -- Destructuring

```
import React from 'react'
const PrintInfo = ({...props}) => {
  return(
    <h1>Hello {props.name} your age is {props.age}</h1>
  )
}
 const App = () => {
  return (
    <>
      <PrintInfo name="Niharika" age={27}/>
      <PrintInfo name="Rahul" age={28}/>
      <PrintInfo name="Ravi" age={29}/>
    </>
  )
}
export default App
```

Example -- Destructuring

```
import React from 'react'
const PrintInfo = () => {
  return (
    <div style={{ width: "100%", height: "100vh", backgroundColor: "blueviolet" }}>
      <h1>Hello ReactJS</h1>
    </div>
  )}
 const App = (props) => {
  return (
    <>
      <PrintInfo {...props}/>
    </>
  )
}
export default App
```

States

- States are similar to props, but it is private & fully controlled by the component.
- State is an object that is owned by the component where it is declared. Its scope is limited to the current component.
- States are mutable objects can be accessed anywhere inside of component.
- In order to create states in ReactJS App we can use "useState" hook.

Example – Simple State

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

const App = () => {
  const [name, setName] = useState("ReactJS")
  return (
    <h1>Hello {name}</h1>
  )
}

export default App;
```

Lifting State Up in ReactJS

- We all know that the 'state' is local so, if we want to access our component state using other components then we need to 'lift up' our component state.
- In order to "lift up" states from one component to other we need to pass our component state to other component props.

Examples of Lifting States

```
import React,{useState} from 'react'
const PrntInfo = ({name}) => {
  return(
    <h1>Hello {name}</h1>
  )
}
const App = () => {
  const [name, setName] = useState("ReactJS")
  return (
    <PrntInfo name={name}/>
  )
}
export default App;
```