1. Parent-Child Communication

Real-Life Analogy

Imagine your **parent gives you instructions** (like "Go buy groceries"). Similarly, in React, a **parent component sends data or functions to child components** using **props**.

How it works

- **Parent:** Sends data using **props**.
- **Child:** Receives data via **props** and can use/display it.

Example

```
function Parent() {
  const myName = 'Venky';
  return <Child name={myName} />;
}

function Child(props) {
  return <h3>My name is {props.name}</h3>;
}
```

Key Point

Props = **Data gifts** from Parent to Child.

2. Lifting State Up

Real-Life Analogy

Imagine you and your friend are playing a game but need to **share the same scoreboard**. So, the **score is managed by your teacher (common parent)** and **both of you just view or update it**.

Why we do this?

When **two components need the same data**, it's better to **lift the data to the nearest parent**.

Example

```
function Parent() {
  const [score, setScore] = useState(0);

  return (
    <>
      <Player1 score={score} />
      <Player2 increaseScore={() => setScore(score + 1)} />
    </>
  );
}

function Player1({ score }) {
  return <h3>Score: {score}</h3>;
}

function Player2({ increaseScore }) {
  return <button onClick={increaseScore}>Increase
Score</button>;
}
```

Key Point

Lift state up when **multiple components need to share/update the same data**.

Lifting State Concept

- **Lifting state** means **declaring the state in a common parent component**, so that:
    - The **parent can pass the state value** to any child (via **props**).
    - The **parent can also pass the function (setState)** to modify the state in any child.

Example:

In **App.jsx**

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

import Login from './Login';

export default function App() {

  const [login, setLogin] = useState(false);

  return (

    <>

      <h1>Login Status: {login ? 'Logged In' : 'Logged Out'}</h1>

      <Login setLogin={setLogin} />

    </>

  );

}
```

In **Login.jsx**

```jsx
export default function Login({ setLogin }) {

  return (

    <button onClick={() => setLogin(true)}>Login</button>

  );

}
```

What Happened Here?

- App.jsx holds the **login state**.
- App.jsx sends setLogin function to Login.jsx via **props**.
- When the button is clicked in Login.jsx, the **App's state changes**, and the component re-renders with **login = true**.

Therefore:

App.jsx holds the state, and **Login.jsx** modifies it through the function passed down via props.

useRef Hook :

What is useRef?

useRef is a **basic React Hook** that returns a **mutable reference object** whose .current property is **persisted across the component's lifecycle**. It does **not cause re-renders** when updated, making it ideal for storing values that don't need to trigger UI updates.

Common Use Cases of useRef

1. **Creating Mutable References**

You can create a persistent reference using :

```
const count = useRef( initialValue ) {
```

- This count will not change across re-renders, unlike state variables.

2. **Accessing User Input Without State**

You can attach useRef to an input element to read its value directly, bypassing the need for controlled inputs:

```
const inputRef = useRef();

const handleClick = () => {

  console.log(inputRef.current.value);

};

return <input type="text" ref={inputRef} />;
```

3. **Direct DOM Manipulation**

useRef gives access to the **actual DOM element**, allowing you to **change styles, focus input, scroll, etc.**:

```
inputRef.current.focus();

inputRef.current.style.color = "red";
```

Example

```
import React, { useRef } from 'react';

export default function Example() {
 const inputRef = useRef(null);
 const handleFocus = () => {
  inputRef.current.focus();
  inputRef.current.style.backgroundColor = 'lightyellow';
 };
 return (
  <div>
<input ref={inputRef} type="text" placeholder="Typehere..." />
   <button onClick={handleFocus}>Focus & Highlight</button>
  </div>
 );
}
```

4. useMemo (Performance Optimization)

Real-Life Analogy

Imagine you are **solving a big math problem**. If you write the answer down, you don’t have to solve it again. That’s what **useMemo** does — it **remembers** expensive calculations!

Why use it?

- Prevents **re-calculating data unnecessarily**.
- Speeds up performance if the calculation is complex.

Example

```
const expensiveValue = useMemo(() => {
   let sum = 0;
   for (let i = 0; i < 100000000; i++) {
      sum += number;
   }
   return sum;
}, [number]);
```

Without useMemo, this heavy calculation would happen every time the component renders — even when it's not needed.

Key Point

useMemo = **Remember results** of a calculation unless the data needed changes.