Pure Component:

In React, **Pure Components** are a performance optimization technique used primarily with class components. They help prevent unnecessary re-renders by implementing a shallow comparison of props and state.

🧠 What Is a Pure Component?

A **Pure Component** is a class component that extends React.PureComponent instead of React.Component. It automatically implements the shouldComponentUpdate() lifecycle method with a shallow comparison of props and state.

```
import React from 'react';


class Greeting extends React.PureComponent {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

If name doesn't change, React skips re-rendering this component—even if the parent re-renders.

🔍 How It Works

- **Shallow Comparison**: It checks if the new props/state are *referentially* equal to the previous ones.
- **Avoids Re-rendering**: If nothing has changed, the component won't re-render, improving performance.

🆚 PureComponent vs Component

Feature	Component	PureComponent
Re-renders on every update	✅ Yes	🚫 No (only if props/state changed)
shouldComponentUpdate	❌ Must be implemented manually	✅ Built-in with shallow comparison
Performance	⚠️ May cause unnecessary renders	🚀 Optimized for performance

---

🧪 When to Use Pure Components

- When your component renders the same output for the same props and state.
- When working with **immutable data structures**.
- In large lists or tables where only a few items change.

⚠️ Caveats

- **Shallow comparison** only checks top-level values. If you mutate nested objects or arrays, changes might not be detected.
- Works only with **class components**. For functional components, use React.memo() instead:

```
import React from 'react';

const Greeting = React.memo(function Greeting({ name }) {

  return <h1>Hello, {name}!</h1>;

});
```

useMemo hook:

The useMemo hook in React is a powerful tool for **performance optimization**. It helps you avoid expensive recalculations by **memoizing** the result of a function—meaning React will reuse the cached value unless its dependencies change.

🧠 What Is useMemo?

useMemo lets you **cache the result of a computation** between renders. It's especially useful when:

- You have **expensive calculations** (e.g., filtering large arrays, sorting, or complex math).
- You want to **prevent unnecessary re-renders** by stabilizing object or array references.

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


const squared = useMemo(() => number * number, [number]);
```

🛠️ Syntax

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

- The first argument is a function that returns the computed value.
- The second is a dependency array—React will recompute the value only if one of these changes.

🚀 Real-World Example

```
import React, { useState, useMemo } from 'react';
function ExpensiveComponent({ numbers }) {
  const sum = useMemo(() => {
    console.log("Calculating sum...");
    return numbers.reduce((acc, num) => acc + num, 0);
  }, [numbers]);

  return <div>Sum: {sum}</div>;
}
```

Without useMemo, the sum would be recalculated on every render—even if numbers didn’t change.

⚠️ When *Not* to Use It

- Don’t use useMemo for **every** calculation—it adds complexity and memory overhead.
- Avoid it for **trivial operations** (like a + b).
- Use it **only when profiling shows performance issues**.

🧩 Bonus: useMemo vs useCallback

Hook	Returns	Use Case
useMemo	Memoized **value**	Expensive calculations
useCallback	Memoized **function**	Stable function references for props

Ref in React:

In React, a **ref** (short for "reference") provides a way to **access and interact with DOM elements or React components directly**—bypassing the usual data flow. Think of it as a way to “peek behind the curtain” when you need to do something outside the normal React rendering cycle.

🔧 How to Create a Ref

In **functional components**, you use the useRef hook:

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


function MyComponent() {
  const inputRef = useRef(null);
  const focusInput = () => {
    inputRef.current.focus();
  };
  return (
    <>
      <input ref={inputRef} type="text" />
      <button onClick={focusInput}>Focus Input</button>
    </>
  );
}
```

In **class components**, you use React.createRef():

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.inputRef = React.createRef();
  }
  focusInput = () => {
    this.inputRef.current.focus();
  };
  render() {
    return (
      <>
        <input ref={this.inputRef} type="text" />
        <button onClick={this.focusInput}>Focus Input</button>
      </>
    );
  }
}
```

🧠 What Refs Are Good For

- **Accessing DOM elements** (e.g., focusing an input, scrolling to a div)
- **Storing mutable values** that don’t trigger re-renders (like timers or previous state)
- **Integrating with third-party libraries** that require direct DOM manipulation

---

🆚 Refs vs State

Feature	useRef	useState
Triggers re-render?	❌ No	✅ Yes
Mutable?	✅ Yes (ref.current = value)	⚠️ Only via setState()
Use case	DOM access, timers, prev values	UI updates, rendering logic

---

⚠️ Best Practices

- Don’t overuse refs—they’re an **escape hatch**.
- Avoid reading or writing ref.current during rendering.
- Use state when the value affects rendering; use refs when it doesn’t.

ForwardRef:

In React, forwardRef is a technique that lets you **pass a ref from a parent component to a child component**, even if that child is a functional component. Normally, refs don’t work with function components—but forwardRef bridges that gap.

🔄 Why Use forwardRef?

- To **expose a DOM node** (like an <input>) inside a child component to its parent.
- To **trigger focus, scroll, or animations** from the parent.
- To **integrate with third-party libraries** that require direct DOM access.

🛠️ Basic Syntax

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


const MyInput = forwardRef((props, ref) => {
  return <input ref={ref} {...props} />;
});
```

Now the parent can do this:

```
import React, { useRef } from 'react';
function Parent() {
  const inputRef = useRef(null);
  const handleClick = () => {
    inputRef.current.focus();
  };
  return (
    <>
      <MyInput ref={inputRef} placeholder="Type here..." />
      <button onClick={handleClick}>Focus Input</button>
    </>
  );
}
```

🧩 How It Works

- forwardRef takes a **render function** with props and ref.
- It returns a component that can **accept a ref prop**.
- That ref is then **attached to a DOM element** inside the child.

⚙️ With useImperativeHandle

You can customize what the parent gets from the ref:

```
import { forwardRef, useImperativeHandle, useRef } from 'react';

const MyInput = forwardRef((props, ref) => {
  const localRef = useRef();

  useImperativeHandle(ref, () => ({
    focus: () => localRef.current.focus(),
    scrollIntoView: () => localRef.current.scrollIntoView(),
  }));
  return <input ref={localRef} {...props} />;
});
```

Now the parent can call ref.current.focus() or ref.current.scrollIntoView()—but not access the raw DOM node.

⚠️ Gotchas

- Only use forwardRef when **absolutely necessary**—it breaks encapsulation.
- Avoid using it in **high-level components** like modals or dashboards.
- Works best for **low-level UI elements** like buttons, inputs, or sliders.

Controlled Components:

In React, **controlled components** are form elements (like <input>, <textarea>, or <select>) whose values are **fully managed by React state**. This means the component's state is the *single source of truth* for the form data.

🧠 What Makes a Component "Controlled"?

A component is controlled when:

- The form element's value is set by React state.
- Any user input triggers an onChange handler that updates the state.

```
import { useState } from 'react';
function ControlledInput() {
  const [name, setName] = useState('');
  const handleChange = (e) => setName(e.target.value);
  return (
    <form>
      <label>Name:</label>
      <input type="text" value={name} onChange={handleChange} />
      <p>Hello, {name}!</p>
    </form>
  );
}
```

In this example, the input field is controlled because its value is tied to the name state.

✅ Benefits of Controlled Components

- **Predictable behavior**: State drives the UI, so you always know what's rendered.
- **Easy validation**: You can validate input before updating state.
- **Centralized control**: Great for complex forms or when integrating with libraries like Formik or Redux.

⚠️ Things to Watch Out For

- **Performance**: Too many controlled inputs in large forms can cause re-renders.
- **Boilerplate**: You'll write more code compared to uncontrolled components.

🆚 Controlled vs Uncontrolled

Feature	**Controlled Component**	**Uncontrolled Component**
Data source	React state	DOM (via ref)
Updates on input change	Yes (onChange updates state)	No (value read from DOM)
Validation	Easy and inline	Manual, often on submit
Use case	Complex forms, validation	Simple forms, quick prototypes

Uncontrolled Components:

In React, **uncontrolled components** are form elements where the **DOM itself manages the form data**, not React. Instead of syncing input values with state via useState, you use **refs** to access values when needed—typically on form submission.

🧪 Example: Uncontrolled Input

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

function UncontrolledForm() {

  const inputRef = useRef();

  const handleSubmit = (e) => {

    e.preventDefault();

    alert(`Input value: ${inputRef.current.value}`);

  };

  return (
  <form onSubmit={handleSubmit}>
   <input type="text" ref={inputRef} />
   <button type="submit">Submit</button>
  </form>
 );
}
```

Here, the input’s value is not tied to React state. Instead, we grab it directly from the DOM using ref.

⚙️ Key Characteristics

- **DOM-driven**: The browser handles the input’s internal state.
- **Ref-based access**: Use useRef() or createRef() to read values.
- **Less boilerplate**: No need for onChange handlers or state variables.
- **Validation**: Typically done on form submission, not in real-time.

⚠️ When to Use Uncontrolled Components

- For **quick prototypes** or **simple forms**.
- When integrating with **non-React libraries** that manipulate the DOM.
- When you don’t need real-time validation or dynamic behavior.

High Order Component:

In React, a **Higher-Order Component (HOC)** is an advanced pattern used to **reuse component logic**. It’s essentially a **function that takes a component and returns a new component** with enhanced behavior—without modifying the original component directly.

🧠 What Is a Higher-Order Component?

```
const EnhancedComponent = higherOrderComponent(OriginalComponent);
```

- higherOrderComponent is a function.
- It accepts OriginalComponent as input.
- It returns EnhancedComponent with added functionality.

🛠️ Example:

```
function App(){
  return(
    <div className="App">
      <h1 style={{ color: "red" }}>Hello React JS</h1>
      <HOC cmp = {Counter} />
    </div>
  )
}
function HOC(props){
    return <h2 style={{color:"brown", backgroundColor:"skyblue" }}><props.cmp /></h2>
}
function Counter(){
    const [count, setCount] = useState(0)
    return(
        <div>
            <h2>High Order Components</h2>
            <h3>{count}</h3>
            <button onClick={()=>setCount(count + 1)}>Update</button>
        </div>
    )
}
```

🔁 Real-World Use Cases

- **Authentication**: Wrap components to restrict access.
- **Logging**: Track lifecycle or user interactions.
- **Styling/Theming**: Inject consistent styles or themes.
- **Data Fetching**: Add API logic without cluttering UI components.

🧩 HOC vs Custom Hook

Feature	**HOC**	**Custom Hook**
Reusability	Wraps components	Reuses logic inside components

Feature	HOC	Custom Hook
Output	New component	Hook return values
Use case	Cross-cutting concerns (auth, log)	Stateful logic (form, fetch, etc.)

⚠️ Best Practices

- **Don’t mutate** the original component.
- **Pass all props** through unless intentionally overridden.
- **Name clearly**: e.g., withAuth, withLogger.
- **Avoid nesting too many HOCs**—it can get messy.