Feature	Use / Benefit
JSX (JavaScript XML)	Lets you write HTML inside JavaScript for easier UI creation.
Components	Break UI into reusable pieces (Functional & Class components).
Virtual DOM	Efficiently updates only changed parts of the UI, improving performance.
One-way Data Binding	Data flows from parent to child, making the app predictable and easier to debug.
State Management	Tracks component data and updates UI when state changes.
Props	Pass data from parent to child components.
Lifecycle Methods / Hooks	Control what happens during mounting, updating, and unmounting. (useEffect for functional components)
Conditional Rendering	Render components or elements based on conditions.
Event Handling	Handle user interactions like clicks, input, and form submissions.
Forms & Controlled Components	Manage form input values via state.
Fragments	Group multiple elements without adding extra nodes to the DOM.
Portals	Render components outside the main DOM hierarchy (e.g., modals, tooltips).
Hooks	Use state, context, and other React features in functional components.
Context API	Share global data like themes, auth, or language without props drilling.
Performance Optimization	Tools like React.memo, useMemo, and useCallback help avoid unnecessary re-renders.
React Router	Handle routing in single-page applications (SPA).
Error Boundaries	Catch JavaScript errors in components to prevent the whole app from crashing.

Feature	Real DOM	Virtual DOM
Definition	The actual HTML DOM in the browser	A lightweight copy of the real DOM in memory
Update	Updates the entire DOM when something changes	Updates only the changed elements using diffing
Performance	Slower, because re-rendering is costly	Faster, because only minimal updates are made
Memory	Uses more memory during updates	Uses less memory, stored in JS memory
Manipulation	Directly manipulates the browser DOM	Manipulates virtual DOM first, then syncs with real DOM
Use in React	React interacts with real DOM indirectly via virtual DOM	React's virtual DOM improves efficiency and performance
Re-rendering	Can be heavy for large apps	Efficient; only changed elements re-render

1. One-Way Data Binding

- **Flow: Parent → Child**
- **What it means: Data goes only from parent to child through props.**
- **Child cannot change parent's data directly.**
- **Example:**

```
function Child({ name }) {
  return <h1>Hello, {name}</h1>;
}

function Parent() {
  return <Child name="Alice" />;
}
```

- **Here, Parent sends name to Child, and Child just displays it.**

---

2. Two-Way Data Binding

- **Flow: UI ↔ State**
- **What it means: Data can move both ways: when the user types something, it updates the state, and when state changes, the UI updates automatically.**
- **Example:**

```
const [name, setName] = useState('');

<input value={name} onChange={e => setName(e.target.value)} />

<p>Hello, {name}</p>
```

- **When you type in the input, name state updates, and <p> shows the updated value automatically.**

✅ **Simple way to remember:**

- **One-way: Data flows down from parent → child.**
- **Two-way: Data flows both ways between UI and state.**

Functional vs. Class Components

Feature	Class Components	Functional Components
Syntax	Class-based with render() method	Function that returns JSX
State	this.state and this.setState()	useState() Hook
Lifecycle	Built-in lifecycle methods	useEffect() Hook
this Keyword	Used to access state, props, and methods	Not used
Performance	Can be optimized with shouldComponentUpdate()	Can be optimized with React.memo() and useCallback()
Adoption	Legacy method; less common in modern React	The standard for modern React development

Functional components are generally preferred for modern React development because:

- **Simplicity & Readability:** They are shorter, cleaner, and easier to maintain than class components.
- **Hooks:** Allow use of state and lifecycle methods directly, replacing the need for classes.
- **No this Issues:** Avoid manual binding and reduce bugs.
- **Performance Optimization:** useMemo and useCallback make preventing unnecessary re-renders easier.
- **Future-Proof:** React is focusing on functional components; all new features and optimizations target them.

Conclusion: For modern projects, functional components are simpler, more flexible, and aligned with React's future direction.

Do Both Components Have a Lifecycle?

Yes, but **class components** use **special methods**, and **functional components** use the **useEffect hook**.

Class Components

1. **Mounting (Added to screen)**
    - constructor() → setup state
    - render() → show UI
    - componentDidMount() → run after component appears (e.g., fetch data)
2. **Updating (State or props change)**
    - render() → update UI
    - componentDidUpdate() → run after update (e.g., do something when data changes)
3. **Unmounting (Removed from screen)**
    - componentWillUnmount() → cleanup (e.g., stop timers, remove event listeners)

Functional Components

All lifecycle phases are handled with **useEffect**:

1. **Mounting**

```
useEffect(() => {
  console.log('Mounted!');
}, []); // runs once
```

2. **Updating**

```
useEffect(() => {
  console.log('Updated!');
}, [count]); // runs when 'count' changes
```

3. **Unmounting**

```
useEffect(() => {
  const timer = setInterval(() => {}, 1000);
  return () => clearInterval(timer); // cleanup on unmount
```

}, []);

Summary:

- **Class:** Uses separate methods for each phase.
- **Functional:** Uses **one hook (useEffect)** for all phases**.**

Phase	**Class Component**	**Functional Component (useEffect)**
Mounting	constructor() → render() → componentDidMount()	useEffect(() => { ... }, []) (runs once on mount)
Updating	render() → componentDidUpdate()	useEffect(() => { ... }, [deps]) (runs when deps change)
Unmounting	componentWillUnmount()	useEffect(() => { return () => { ... } }, []) (cleanup on unmount)

Hook	**Use Case**
useState	Store and update state
useEffect	Side effects, lifecycle methods replacement
useContext	Access context data globally
useRef	DOM references or persistent values
useMemo	Memoize expensive calculations
useCallback	Memoize functions to prevent re-renders
useReducer	Manage complex state
useLayoutEffect	DOM measurements & synchronous updates
useImperativeHandle	Customize ref exposure in child components

useDebugValue	Debugging hooks in React DevTools

1. useState

- **Purpose:** Store and update state in functional components.
- **Example:**

```
const [count, setCount] = useState(0);

setCount(count + 1);
```

- **Use:** Any variable that can change over time (like form input, counters, toggles).

---

2. useEffect

- **Purpose:** Handle side effects like fetching data, subscriptions, or timers.
- **Example:**

```
useEffect(() => {
  console.log('Component mounted or updated!');
}, [count]); // runs when count changes
```

- **Use:** Lifecycle methods replacement in functional components (mount, update, unmount).

---

3. useContext

- **Purpose:** Access data from a **React context** without passing props manually.
- **Example:**

```
const theme = useContext(ThemeContext);
```

- **Use:** Share global data like theme, user info, or language across components.

---

4. useRef

- **Purpose:** Store a **reference to a DOM element** or **persistent value** that doesn't trigger re-renders.
- **Example:**

```
const inputRef = useRef();

<input ref={inputRef} />;

inputRef.current.focus();
```

- **Use:** Focus input, measure DOM elements, store timers or previous values.

---

5. useMemo

- **Purpose:** Optimize performance by **memoizing expensive calculations**.
- **Example:**

```
const expensiveValue = useMemo(() => computeHeavyTask(count), [count]);
```

- **Use:** Avoid recalculating values unless dependencies change.

---

6. useCallback

- **Purpose:** Memoize a function to prevent unnecessary re-creations.
- **Example:**

```
const handleClick = useCallback(() => { console.log(count); }, [count]);
```

- **Use:** Pass functions to child components without causing re-renders.

---

7. useReducer

- **Purpose:** Manage complex state logic (alternative to useState).
- **Example:**

```
const [state, dispatch] = useReducer(reducer, initialState);

dispatch({ type: 'INCREMENT' });
```

- **Use:** For multiple state values or complex updates, like forms or counters.

---

8. useLayoutEffect

- **Purpose:** Like useEffect, but runs **synchronously after DOM changes**.
- **Use:** Measure DOM size, update scroll positions, or make adjustments before the browser paints.

---

9. useImperativeHandle

- **Purpose:** Customize which values or functions are exposed when using ref with a child component.
- **Use:** Give parent access to specific child functions (like focus a custom input).

---

10. useDebugValue

- **Purpose:** Display custom labels for React DevTools for debugging hooks.
- **Use:** Mostly for library developers; shows values in DevTools.

Method	**Direction**	**Use Case**
Props	Parent → Child	Most common, one-way data flow
Callback Functions	Child → Parent	Send data from child to parent
Context API	Any component	Global/shared data without props drilling
Redux / State Management	Any component	Large-scale global state
useReducer + Context	Any component	Medium complexity global state
Local / Session Storage	Any component	Persist data across reloads

URL / Query Parameters	Any component	Routing-based data sharing
Refs	Parent → Child	Imperative child methods access

1. Props (Parent → Child)

- Pass data from a **parent component to its child** using props.
- **Example:**

```
function Child({ name }) {
  return <h1>Hello, {name}</h1>;
}

function Parent() {
  return <Child name="Alice" />;
}
```

- **Use:** Most common way for one-way data flow.

---

2. Callback Functions (Child → Parent)

- Pass a **function from parent to child**. Child calls it to send data back.
- **Example:**

```
function Child({ onClick }) {
  return <button onClick={() => onClick('Hello')}>Send</button>;
}

function Parent() {
  const handleData = (msg) => console.log(msg);
  return <Child onClick={handleData} />;
}
```

- **Use:** Communicate **upward** from child to parent.

---

3. Context API (Global / Any Component)

- Provides a way to **share data across multiple components** without props drilling.
- **Example:**

```
const ThemeContext = React.createContext('light');


function Child() {
  const theme = useContext(ThemeContext);
  return <div>{theme}</div>;
}


function Parent() {
  return <ThemeContext.Provider value="dark"><Child /></ThemeContext.Provider>;
}
```

- **Use:** Themes, authentication, language settings, or global data.

---

4. Redux / State Management Libraries

- Manage **global state** accessible anywhere in the app.
- **Use:** For complex apps where many components need the same state.
- Example libraries: Redux, Zustand, MobX, Recoil.

---

5. useReducer + Context (Combined)

- Use **useReducer with Context** to manage global state in React **without external libraries**.
- **Use:** Alternative to Redux for medium complexity apps.

---

6. Local Storage / Session Storage

- Store data in browser storage and read it in any component.
- **Use:** Persisting data across page reloads or sharing between distant components.

---

7. URL / Query Parameters

- Pass data via **URL params or query strings** (React Router).
- **Example:** /profile/123 → use useParams() to get id = 123.

---

8. Refs (less common)

- **Pass refs** to access child component methods or DOM elements.
- **Use:** Trigger child behavior imperatively (not typical for data flow).

What is Redux?

- Redux is a **state management library** for React.
- It helps manage **global state** that multiple components can share.

Core Concepts

1. **Store** – Holds the **entire app state**.
2. **Action** – An object describing **what happened** (type + optional data).
3. **Reducer** – A function that **updates state** based on the action.
4. **Dispatch** – Sends an action to the reducer.
5. **Selector** – Reads state from the store in components.

Redux Workflow

1. Component triggers an event →
2. Dispatch an action →
3. Reducer updates the state →
4. Store holds new state →
5. Components re-render with updated state

Diagram:

Component --dispatch--> Action --> Reducer --> Store --> Component re-renders

---

Why Redux?

- Centralized and predictable state
- Easy to share data across components
- Debug-friendly with DevTools
- Great for large apps

Feature	Controlled Component	Uncontrolled Component
Definition	Input value is controlled by React state	Input value is handled by the DOM itself
Data Handling	State stores and updates the input value	Ref is used to access input value when needed
onChange	Required to update state on input change	Not required; DOM manages value automatically
Use Case	Forms where you need **validation, dynamic updates, or instant feedback**	Simple forms where you just read value on submit
Example	<input value={name} onChange={e => setName(e.target.value)} />	<input ref={inputRef} /> and read inputRef.current.value
React Control	React **fully controls** the input value	React **does not control** input value
Advantages	Easy to validate, manipulate, and submit	Less code for simple inputs
Disadvantages	More code and boilerplate	Harder to validate or dynamically manipulate input

1. Controlled Components

- **Definition:** The form input's value is **controlled by React state**.
- **How it works:** The component's state stores the input value, and updates happen via onChange.
- **Benefit:** Easier to **validate, manipulate, and submit form data**.

Example:

```
const [name, setName] = useState('');


<input type="text" value={name} onChange={e => setName(e.target.value)} />

<p>You typed: {name}</p>
```

- Here, the input value is **controlled by the name state**.

---

2. Uncontrolled Components

- **Definition:** The form input's value is **handled by the DOM**, not React state.
- **How it works:** Use a **ref** to get the value when needed.
- **Benefit:** Simpler for small forms where you don't need to constantly track input changes.

Example:

```
const inputRef = useRef();


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

<button onClick={() => console.log(inputRef.current.value)}>Submit</button>
```

- Here, the input value is **managed by the DOM**, and we read it via ref.

Difference between Redux and Context API

Feature	Redux	Context API
Purpose	Global state management for large apps	Share data globally without props drilling
Complexity	More setup and boilerplate required	Simple and easy to use
Data Flow	Central store, actions, reducers	Provider → Consumer pattern
Performance	Optimized with middleware & DevTools	Can cause unnecessary re-renders if overused
Middleware Support	Yes (e.g., redux-thunk, redux-saga)	No middleware support
Use Case	Large apps with complex state	Small to medium apps with simple global data
Learning Curve	Steeper	Easier

Summary:

- **Redux:** Best for large-scale apps with complex state logic.
- **Context API:** Best for small/medium apps or sharing simple data (theme, auth, language).

- What is useReducer and When to Use It

  Definition: A React Hook that manages **state using a reducer function**, similar to Redux but built-in.

- **How it works:** You define a reducer (state, action) => newState and dispatch actions to update the state.

Syntax:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

Example:

```
const initialState = { count: 0 };


function reducer(state, action) {
  switch(action.type) {
    case 'INCREMENT': return { count: state.count + 1 };
    case 'DECREMENT': return { count: state.count - 1 };
    default: return state;
  }
}


const [state, dispatch] = useReducer(reducer, initialState);


// Usage
dispatch({ type: 'INCREMENT' });
```

When to use useReducer:

- When you have **complex state logic** with multiple sub-values.
- When state updates **depend on previous state**.
- When you want a **Redux-like pattern** without installing Redux.

How to conditionally render components?

1. Using if Statement

- You can use an if statement **inside the component** to decide what to render.

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Welcome back!</h1>;
  } else {
    return <h1>Please sign in.</h1>;
  }
}
```

---

2. Using Ternary Operator

- Quick way to render **one thing or another** in JSX.

```
function Greeting({ isLoggedIn }) {
  return (
    <h1>{isLoggedIn ? 'Welcome back!' : 'Please sign in.'}</h1>
  );
}
```

---

3. Using Logical && Operator

- Render something **only if a condition is true**.

```
function Notification({ message }) {
  return (
    <div>
      {message && <p>{message}</p>}
    </div>
  );
}
```

- If message is empty or false, nothing is rendered.

---

4. Using switch Statement

- Useful for multiple conditions.

```
function Status({ status }) {
  switch(status) {
    case 'loading':
      return <p>Loading...</p>;
    case 'success':
      return <p>Data loaded!</p>;
    case 'error':
      return <p>Error occurred!</p>;
    default:
      return null;
  }
}
```

✅ **Summary:**

- **if/else:** For complex conditions.
- **Ternary ? : :** Quick inline rendering.
- **Logical && :** Render only if true.
- **switch:** Multiple cases.

Feature	**if Statement**	**switch Statement**
Purpose	Handles **conditional logic** for one or more conditions	Handles **multiple specific values** for a single variable
Syntax	if (condition) { ... } else if (...) { ... } else { ... }	switch(expression) { case value1: ...; break; case value2: ...; break; default: ...; }
Use Case	Simple or complex **range-based conditions**	When you need to check **specific discrete values** of a variable
Readability	Can get long with many else if blocks	Cleaner for many discrete cases
Performance	Slightly slower if many else if blocks	Faster for multiple discrete cases in most cases
Default Case	Use else block	Use default block

1. What is Lifting State Up?

- Moving state from a child component to a **common parent** so multiple children can share or update it.
- **Example:** Two input fields need to share the same value. Instead of each storing its own state, move it to the parent and pass via props.

2. Difference Between State and Props

Feature	State	Props
Definition	Local data of a component	Data passed from parent to child
Changeable?	Yes, using setState or useState	No, read-only in child
Purpose	Manages dynamic data	Share data between components

3. Can Props Be Changed Inside a Child Component?

 No. Props are **read-only**.
 Child cannot modify props directly; it can only **ask parent** to change via a callback function.

4. How useEffect Works in Functional Components

- **It replaces lifecycle methods in class components.**
- **Runs after render and can be used for side effects like fetching data, timers, or subscriptions.**
- **Example:**

```
useEffect(() => {
```

```
  console.log("Component mounted or updated");
}, [dependency]); // runs when 'dependency' changes
```

5. Difference Between useEffect and useLayoutEffect

- **useEffect:** Runs **after DOM updates** are painted to the screen. Non-blocking.
- **useLayoutEffect:** Runs **before the browser paints**. Useful for measuring DOM elements or updating layout synchronously.

---

6. What is Strict Mode?

- <React.StrictMode> is a tool to **highlight potential problems** in an app.
- It **does not render anything in the UI**, but helps detect:
  - Unsafe lifecycle methods
  - Deprecated APIs
  - Side effects in development

---

7. Lists & Keys

- **Lists:** Render multiple items using .map().
- **Keys:** Unique identifier for each element in a list to help React **track changes efficiently**.
  ```
  const items = ['Apple','Banana'];
  items.map((item, index) => <li key={index}>{item}</li>)
  ```

---

8. How to Implement Routing in React

- Use **React Router** (react-router-dom).
- Example:
  ```
  import { BrowserRouter as Router, Route, Routes } from 'react-router-dom';

  <Router>
   <Routes>
    <Route path="/" element={<Home />} />
    <Route path="/about" element={<About />} />
  ```

```
  </Routes>
</Router>
```

---

9. Difference Between BrowserRouter and HashRouter

Feature	BrowserRouter	HashRouter
URL	Clean URLs (example.com/about)	URL has hash (example.com/#/about)
Server Required?	Yes, server should handle all routes	No, works on static file servers
Use Case	Modern web apps	Apps hosted on GitHub Pages or static hosting

---

10. How to Pass Params and Query Strings

- **Params:** path="/user/:id" → access via useParams()
  <Route path="/user/:id" element={<User />} />
  const { id } = useParams();
- **Query Strings:** example.com?name=Alice → access via useLocation() or URLSearchParams
  const query = new URLSearchParams(location.search);
  const name = query.get("name");

---

11. How to Optimize Performance in React

- Avoid unnecessary re-renders with:
  - React.memo for components
  - useMemo to memoize expensive calculations
  - useCallback to memoize functions
- Lazy load components with React.lazy and Suspense.

---

12. React.memo, useMemo, useCallback

Hook/Feature	Use

React.memo	Wrap a component to **prevent re-render** if props haven't changed
useMemo	Memoize **expensive calculations** so they run only when dependencies change
useCallback	Memoize **functions** so they don't get recreated on every render

---

1. What are Fragments in React?

- **Definition:** A way to group multiple elements **without adding extra nodes to the DOM**.
- **Why use it:** Avoid unnecessary wrappers that clutter the HTML.
- **Example:**

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

function List() {
  return (
    <Fragment>
      <li>Item 1</li>
      <li>Item 2</li>
    </Fragment>
  );
}
```

- Short syntax: <>...</>

---

2. What are Portals and Their Use?

- **Definition:** A way to **render a component outside the main DOM hierarchy**.
- **Use Case:** Modals, tooltips, or popups that need to appear above other content.
- **Example:**

```
import ReactDOM from 'react-dom';
```

ReactDOM.createPortal(<Modal />, document.getElementById('modal-root'));

- Modal will render inside #modal-root instead of parent component.

---

3. What are Error Boundaries?

- **Definition:** Components that **catch JavaScript errors** in their child components and display a fallback UI.
- **Use Case:** Prevent the entire app from crashing when an error occurs.
- **Example:**

```
class ErrorBoundary extends React.Component {
  constructor(props) { super(props); this.state = { hasError: false }; }
  static getDerivedStateFromError(error) { return { hasError: true }; }
  render() { return this.state.hasError ? <h1>Something went wrong.</h1> : this.props.children; }
}
```

---

4. What is a Higher-Order Component (HOC)?

- **Definition:** A function that **takes a component and returns a new component** with added functionality.
- **Use Case:** Code reuse, adding props, authentication checks, etc.
- **Example:**

```
function withLogger(Component) {
  return function WrappedComponent(props) {
    console.log('Props:', props);
    return <Component {...props} />;
  };
}
```

---

5. What is React Fiber?

- **Definition:** React's **reconciliation engine** that determines how to update the UI efficiently.
- **Key Points:**
  - Introduced in React 16.
  - Makes rendering **incremental**, which improves performance for large apps.
  - Supports features like **concurrent mode**, allowing React to pause and resume rendering.