1. Component Lifecycle

React components go through different phases during their existence, known as the **component lifecycle**. These phases include:

Mounting Phase (Component is being created and inserted into the DOM)

- `constructor()` – Initializes state and binds methods.
- `static getDerivedStateFromProps()` – Updates state based on props (rarely used).
- `render()` – Returns JSX to be rendered.
- `componentDidMount()` – Runs after the component is mounted (API calls, subscriptions).

Updating Phase (Component is re-rendered due to state/prop changes)

- `static getDerivedStateFromProps()` – Updates state before rendering.
- `shouldComponentUpdate()` – Determines if the component should re-render (optimization).
- `render()` – Re-renders the component.
- `getSnapshotBeforeUpdate()` – Captures DOM info before update (rarely used).
- `componentDidUpdate()` – Runs after the component updates.

Unmounting Phase (Component is removed from the DOM)

- `componentWillUnmount()` – Cleanup (timers, subscriptions).

Example:

```
class LifecycleExample extends React.Component {
 constructor(props) {
  super(props);
  this.state = { count: 0 };
  console.log("Constructor");
 }
```

```
  componentDidMount() {
    console.log("Component mounted");
  }

  componentDidUpdate() {
    console.log("Component updated");
  }

  componentWillUnmount() {
    console.log("Component will unmount");
  }

  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    console.log("Render");
    return (
      <div>
        <p>Count: {this.state.count}</p>
        <button onClick={this.increment}>Increment</button>
      </div>
    );
  }
}
```

2. Working with Forms

Controlled Components

- Form elements whose values are controlled by React state.
- **Example:**

```
function ControlledForm() {
  const [name, setName] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Name: ${name}`);
  };
```

```
  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Uncontrolled Components

- Form elements that manage their own state (using `ref`).
- **Example:**

```
function UncontrolledForm() {
  const inputRef = useRef(null);

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Name: ${inputRef.current.value}`);
  };

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

3. React Router

Basic Routing (`Link`, `Routes`, `Route`, `Outlet`)

```
import { BrowserRouter, Routes, Route, Link, Outlet } from "react-router-dom";

function App() {
  return (
    <BrowserRouter>
      <nav>
```

```
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
    </nav>
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
    </Routes>
  </BrowserRouter>
 );
}

function Home() {
 return <h1>Home Page</h1>;
}

function About() {
 return (
  <div>
   <h1>About Page</h1>
   <Outlet /> {/* Renders nested routes */}
  </div>
 );
}
```

Passing Parameters

```
<Route path="/user/:id" element={<User />} />
```

useParams() & useNavigate()

```
import { useParams, useNavigate } from "react-router-dom";

function User() {
 const { id } = useParams();
 const navigate = useNavigate();

 return (
  <div>
   <p>User ID: {id}</p>
   <button onClick={() => navigate("/")}>Go Home</button>
  </div>
 );
}
```

4. Lazy Loading & Suspense

- Load components only when needed (improves performance).

```
import { lazy, Suspense } from "react";

const LazyComponent = lazy(() => import("./LazyComponent"));

function App() {
 return (
  <Suspense fallback={<div>Loading...</div>}>
   <LazyComponent />
  </Suspense>
 );
}
```

5. Sharing State Between Components

Parent and Child Components

```
function Parent() {
 const [count, setCount] = useState(0);
 return <Child count={count} onIncrement={() => setCount(count + 1)} />;
}

function Child({ count, onIncrement }) {
 return (
  <div>
   <p>Count: {count}</p>
   <button onClick={onIncrement}>Increment</button>
  </div>
 );
}
```

Lifting State Up

- Moving shared state to the closest common ancestor.

```
function Parent() {
 const [text, setText] = useState("");
 return (
  <>
```

```
    <ChildA text={text} onTextChange={setText} />
    <ChildB text={text} />
   </>
  );
}
```

6. Axios - HTTP Requests

Promise API Support

```
import axios from "axios";

axios.get("https://api.example.com/data")
  .then((res) => console.log(res.data))
  .catch((err) => console.error(err));
```

HTTP Methods (`get`, `post`, `put`, `delete`)

```
// GET
axios.get("/users");

// POST
axios.post("/users", { name: "John" });

// PUT
axios.put("/users/1", { name: "Updated John" });

// DELETE
axios.delete("/users/1");
```

7. Testing

Example Test

```
import { render, screen, fireEvent } from "@testing-library/react";
import App from "./App";

test("increments counter", () => {
  render(<App />);
  const button = screen.getByText("Increment");
  fireEvent.click(button);
  expect(screen.getByText("Count: 1")).toBeInTheDocument();
});
```

Key Testing Methods

- `render()` – Renders a React component.
- `fireEvent` – Simulates user interactions.
- `expect()` – Asserts expected outcomes.