React

Introduction

1. Getting started
    a. Importing Libraries and rendering the root element

```
var React = require("react");
var ReactDOM = require("react-dom");
// or
//import React from "react";
//import ReactDOM from "react-dom";

ReactDOM.render(<h1>Jai Shree Ram</h1>,document.getElementById("r
//ReactDOM.render(What to Show,Where to Show);
//What to show - Single HTML elememt only (Use everything inside singl
```

    b. JSX
        i. **JSX (JavaScript XML)** is a syntax extension for JavaScript. It looks like HTML but works inside JavaScript. React uses it to describe what the UI should look like.
        ii. <h1>Jai Shree Ram</h1>
        This is JSX, and it means:
        👉 Create an HTML <h1> element with the text “Jai Shree Ram”.
        iii. JSX allows to write HTML inside JS but it also allows to write JS→HTML→JS
            1. { expression} - Using this any expression can be written but not statements but ternary and logical expressions are allowed.
                a. ex - 9+10 is allowed but if(cond) is not .
            2. ES6 - {`${expression}`}
    c. Babel

i. **Babel** is a tool (a compiler) that **converts JSX into regular JavaScript** so that browsers can understand it.

ii. converts it to : React.createElement("h1", null, "Jai Shree Ram")

2. Styling

a. Inline - style attr takes JS object (Preferred for dynamic styling)

```
import React from "react";
import ReactDOM from "react-dom";
const customStyle = {
  color: "red",
  fontSize: "20px",//CamelCase
};
customStyle.color = "orange";
ReactDOM.render(
  <div>
    <p style={customStyle}>Jai Shree Ram</p>
    <p style={{color:"red"}}>Jai Shree Ram</p>
  </div>,
  document.getElementById("root")
);
```

b. External - Add class to tag (Used for major styling of website)

Here instead of class, **className** is used.

💡 Convetions-

1. camelCase - Attributes are always written in camelCase
2. No self closing tags - Always close each tag (<img/>)

Components

1. A **React component** is a reusable, self-contained piece of UI.
2. It's like a **function that returns JSX**.
3. Importing/Exporting a component
    a. `index.js`

```
js
CopyEdit
import React from "react";
import ReactDOM from "react-dom";
import List from "./List.jsx"; // ✅ Importing the component
//import List from "./List"; //This can also be used
ReactDOM.render(
  <div>
    <List /> //Here component is embedded
  </div>,
  document.getElementById("root")
);
```

1. `List.jsx` (Component - PanelCase)

```
js
CopyEdit
import React from "react";

function List() {
  return (
    <div>
      <ul>
        <li>A</li>
        <li>B</li>
        <li>C</li>
      </ul>
```

```
    </div>
  );
}

export default List;
//It makes the component available to be used in another file.
```

4. Methods to import/export

Type	Syntax (Export)	Syntax (Import)	Notes
Default Export	`export default value`	`import name from "./file"`	Only one per file, name is custom
Named Export	`export { name1, name2 }`	`import { name1 } from "./file"`	Multiple per file, name must match
Namespace Import	*use with named exports*	`import * as obj from "./file"`	All named exports under `obj` (obj.name())

Props

1. **Props** (short for **"properties"**) are **read-only inputs** passed from one React component (usually a parent) to another (usually a child).They allow you to **pass data** into components so they can display dynamic content.
2. Props Flow :

```
css
CopyEdit
App (parent)
 └── sends props 👉 to Card (child)
        👇
  Card uses props to display info
```

Part	Example from your code	Explanation

Passing props	`<Card name="Ram" phone="123" />`	Props are added as **attributes** in JSX
Receiving props	`function Card(props) { return <p> {props.name}</p>; }`	Function receives a `props` object
Using props	`props.name` , `props.phone` inside JSX	Used to display or work with passed-in values
Dynamic rendering	`name={Contacts[0].name}`	Makes component dynamic based on array/object data

Arrow Functions

Arrow functions are a
shorter syntax for writing functions in JavaScript.

Basic Syntax

```
// Traditional function
function add(a, b) {
  return a + b;
}

// Arrow function (same as above)
const add = (a, b) => {
  return a + b;
};

//Anonymous function
const doubled = numbers.map(function(num) {
  return num * 2;
});

const doubled = numbers.map((num) => {return num * 2});
```

Map()

```
js
CopyEdit
array.map(function(currentValue, index, array) {
  // return the new value for the new array
})
```

- Loops through every element of an array.
- Runs a function for each element.
- Returns a **new array** with the results.

ex-

used the `map()` method to loop over the `contacts` array and render a `Card` component for each contact.

```
jsx
CopyEdit
{contacts.map(createCard)}
```

🔍 Breaking it Down

Let's break this part:

- `contacts` is your array of contact objects.
- `.map(createCard)` means: for **each object** in the `contacts` array, call the `createCard()` function and use its return value (a `<Card />` JSX element).
- The `createCard()` function looks like this:

```
js
CopyEdit
function createCard(contact) {
  return (
    <key={contact.id}
      id={contact.id}
      name={contact.name}
```

```
      img={contact.imgURL}
      tel={contact.phone}
      email={contact.email}
    />
  );
}
```

💡 key - It is the differentiating attribute for each component.
Without a unique key, React
re-renders all list items instead of just the changed ones.
It is
not a **prop.**

filter()

Returns a **new array** containing elements that **pass a test** (i.e., return `true` in a callback).

🔷 Syntax:

```
array.filter(function(value, index, array) {return condition});
```

🧪 Example:

```
js
CopyEdit
const numbers = [1, 2, 3, 4, 5];

const even = numbers.filter(function(num){ return num % 2 === 0});
```

```
console.log(even); // ➤ [2, 4]
```

Reduce()

Reduces the array to a **single value** (like sum, product, etc.).

🔷 Syntax:

```
array.reduce((accumulator, currentValue, index, array) ⇒ {}, initialValue);
```

🧪 Example:

```
const numbers = [1, 2, 3, 4];

const sum = numbers.reduce((acc, num) ⇒ acc + num, 0);

console.log(sum); // ➤ 10
```

- `acc` starts as `0`
- Then: `0 + 1 = 1`, `1 + 2 = 3`, `3 + 3 = 6`, `6 + 4 = 10`

Find()

Returns the **first element** that satisfies a condition.

Stops searching after the first match.

🔷 Syntax:

```
array.find((value, index, array) ⇒ condition);
```

🧪 Example:

```
const numbers = [5, 10, 15, 20];

const found = numbers.find(num ⇒ num > 10);

console.log(found); // ➤ 15 (first one greater than 10)
```

FindIndex()

Returns the **index** of the **first element** that satisfies a condition.

Returns `-1` if none foun

🔷 Syntax:

```
array.findIndex((value, index, array) ⇒ condition);
```

🧪 Example:

```
const numbers = [5, 10, 15, 20];

const index = numbers.findIndex(num ⇒ num > 10);
```

```
console.log(index); // ➤ 2 (15 is at index 2)
```

State

- **State** is a built-in object that stores **dynamic data** for a component.
- When **state changes**, the component **re-renders** automatically.
- It's like the component's **memory**.

🔶 Declarative vs Imperative

Concept	Declarative (React way)	Imperative (Manual DOM way)
Meaning	Tell **what** you want	Tell **how** to do it
React example	`<h1>{count}</h1>` – UI updates on state change	`document.getElementById("h1").innerText = count;`
Style	Clear, predictable	Manual, harder to manage

Destructuring

Destructuring is a syntax that allows you to **unpack values** from arrays or properties from objects **into variables**.

Type	Syntax	Example Value
Array	`const [a, b] = arr`	`[10, 20]`
Object	`const {x, y} = obj`	`{x: 1, y: 2}`
Nested Array	`const [a, [b, c]] = nestedArr`	`[1, [2, 3]]`
Nested Obj	`const {a: {b}} = nestedObj`	`{a: {b: 5}}`

Hooks

Hooks are special **functions** in React that let you **"hook into"** React features like **state**, **lifecycle**, or **context** — **without writing a class**.

💡 If we do not use hooks we have to rerender the whole component for updating a single state

Most Common Hooks:

Hook	Purpose
`useState()`	Add **state** to functional components
`useEffect()`	Run code on **mount/update** (like lifecycle methods)
`useContext()`	Use **context API** (global state)
`useRef()`	Refer to DOM elements or persist data
`useReducer()`	Advanced state management (like Redux)

1. useState()
   a. `useState()` lets a functional component have **state**.
   b. Syntax:

```
const [state, setState] = useState(initialValue);
```

Part	Meaning
`useState`	React Hook that gives you state in function components
`state`	Current value of the state
`setState`	Function to **update** the state
`initialValue`	The **starting value** of the state

c. Example: Counter with `useState`

```
import React, { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0); // 0 is the initial value
```

```
  return (
    <div>
      <h1>{count}</h1>
      <button onClick={() => setCount(count + 1)}>
        Increase
      </button>
    </div>
  );
}

export default Counter;
```

d. setState()

```
setState((prevValue) => {
  return newValue;
});
```

- This is a **callback function**.
- React **passes the current (previous) state** into it automatically.
- You decide what to name it.(Here it is prevValue)

e. Spread Operator

The spread operator lets you **copy or merge** arrays/objects **without mutating** the original.

✅ For arrays:

```
const newArr = [...oldArr, newItem];
```

✅ For objects:

```
const newObj = { ...oldObj, newProp: "value" };
```

ex-Here's a real use-case where both are used together:

```jsx
import React, { useState } from "react";

function ProfileUpdater() {
  const [user, setUser] = useState({
    name: "Alice",
    age: 25,
    location: "Delhi"
  });

  const updateLocation = () => {
    setUser(prevUser => ({
      ...prevUser,
      location: "Mumbai" //We can use it like [name] = event.target.value wh
en particular input is triggered so other inputs will note change only [nam
e] = event.target.name
    }));
  };

  return (
    <div>
      <h2>{user.name}</h2>
      <p>Age: {user.age}</p>
      <p>Location: {user.location}</p>
      <button onClick={updateLocation}>Change Location</button>
    </div>
  );
}
```

🔁 What is `useEffect`?

`useEffect` is a **React hook** that lets you **run code when something happens** in your component — like:

- When the component **shows up (mounts)**

- When a **value changes**
- When the component **goes away (unmounts)**

📦 Basic Syntax

```
jsx
CopyEdit
useEffect(() => {
  // This code runs AFTER the component renders

  return () => {
    // (Optional) This cleanup code runs when the component is removed
  };
}, [dependency]);
```

🧠 Think of it like:

Situation	Real Life Example
No dependencies `[]`	Run once when the page loads (like opening a file)
With dependencies `[count]`	Run again whenever `count` changes (like reacting to a setting)
No array	Run after **every** render (not common)

✅ Simple Example

```
jsx
CopyEdit
import { useEffect } from "react";

function Welcome() {
  useEffect(() => {
    console.log("Component mounted! 🎉");
```

```
    return () => {
      console.log("Component will unmount 😢");
    };
  }, []);

  return <h1>Hello!</h1>;
}
```

✅ Output:

- "Component mounted!" when the component first appears
- "Component will unmount" when the component disappears

🧃 Example with Data Fetch

```
jsx
CopyEdit
useEffect(() => {
  fetch("https://api.example.com/data")
    .then(res => res.json())
    .then(data => console.log(data));
}, []);
```

✅ This fetches data **once** when the component loads.

💡 Summary

- `useEffect` = "Run this code **after** the component shows up or updates"
- Can be used for:
  - API calls
  - Changing the page title

- Setting timers
- Cleaning up (like unsubscribing)

📘 React Router (React Router DOM)

React Router is a library that helps you **navigate between pages (components)** in a React app without refreshing the page (Single Page Application - SPA).

✅ Key Concepts:

Term	Meaning
`BrowserRouter`	Wraps your app; enables routing using the browser URL
`Routes`	A container for all the different routes
`Route`	Defines a path and which component to show
`Link`	Navigation without reloading the page (like `<a>` tag but SPA-friendly)
`useNavigate()`	Hook to programmatically navigate to another route
`useParams()`	Hook to get route parameters (e.g. `/user/:id`)

📦 Example Setup:

```
jsx
CopyEdit
import { BrowserRouter, Routes, Route, Link } from "react-router-dom";
import Home from './Home';
import About from './About';

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>
  );
}
```

Feature	React Router	useEffect
Purpose	Navigation between components	Perform side effects
Core Component	`Route`, `Link`, `BrowserRouter`	`useEffect()`
Common Use	Page switching	Fetching API data, DOM manipulation
Executes	On URL change	On component mount/update (depends on deps)