⚛⚛⚛⚛ REACT ⚛⚛⚛⚛

Why react came in picture

🧠 First, Understand Two Styles of Programming

Style	What it means	How it feels
Imperative	You tell the computer how to do things step-by-step.	Like giving cooking instructions.
Declarative	You tell the computer what you want, and it figures out the how.	Like saying "Make me a pizza"—you care about the result, not the steps.

🔍 So Why Use React When We Have JS?

Feature	JavaScript (DOM)	React
DOM updates	Manual	Automatic
Code style	Imperative	Declarative
State management	You do it yourself	`useState`, `useEffect` etc.
Reusability	Harder	Easy (Components)
Scaling apps	Difficult	Much easier
Performance	Okay	Fast with Virtual DOM
Maintainability	Can become messy	Cleaner structure

📜 JavaScript vs React (Summary Table)

Feature	JavaScript (Imperative)	React (Declarative)
Programming Style	Imperative (step-by-step instructions)	Declarative (describe what UI should look like)
DOM Manipulation	Manual (e.g., `document.getElementById`)	Automatic via Virtual DOM
UI Updates	You handle state + DOM updates yourself	React handles re-rendering automatically
Component Reuse	Harder to structure and reuse	Easy with components
State Management	Manual (global vars, event listeners)	Built-in with `useState`, `useReducer`, etc.
Code Maintenance	Becomes complex as app grows	Cleaner and more manageable
Learning Curve	Lower initially	Slightly higher, but pays off in large apps
Performance	Depends on how you write code	Efficient updates with Virtual DOM
Best For	Small/simple scripts or legacy projects	Modern, scalable, dynamic web apps

PROPS

Props (short for "properties") are read-only inputs that you pass from a parent component to a child component in React.

They allow components to be reusable and dynamic by giving them custom data.

💡 Key Points about Props

Feature	Explanation
Read-only	Props cannot be changed inside the child component.
Passed down	Always flow from parent → child.
Reusable	Make components flexible with different data.
Destructuring	Common to use: `function Component({ title }) {}`

🔒 Can I Modify Props?

- ❌ No — Props are immutable inside the component.
- If you need to change data, you should use state (useState) instead.

```jsx
function Menu() {
  return (
    <main className="menu">
      <h2>Our Menu</h2>
      <Pizza
        name="Pizza Funghi"
        ingredient="Tomato, mozarella, mushrooms, and onion"
        photoName="pizzas/funghi.jpg"
        price={100}
      />
      <Pizza
        name="Pizza Prosciutto"
        ingredient="Tomato, mozarella, ham, aragula, and burrata cheese"
        photoName="pizzas/prosciutto.jpg"
        price={180}
      />
    </main>
  );
}
```

```jsx
function Pizza(props) {
  console.log(props);
  return (
    <div className="pizza">
      <img src={props.photoName} alt={props.name} />
      <div>
        <h3>{props.name}</h3>
        <p>{props.ingredient}</p>
        <span>{props.price + 3}</span>
      </div>
    </div>
  );
}
```

Destructuring Props

Props destructuring in React is the practice of extracting specific properties from the props object directly, so you can access them without repeatedly writing props. in your component.

- Props destructuring makes React components more concise, readable, and maintainable by letting you directly extract needed values from the props object.

✅ Why We Need It:

We use props destructuring to:

1. **Write cleaner and shorter code**
   → Avoid repeating `props.name`, `props.age`, etc.
2. **Improve readability**
   → You see exactly which props the component uses — right in the function signature.
3. **Avoid clutter inside JSX**
   → Especially useful when there are multiple props.
4. **Make code more maintainable**
   → Easier to update, refactor, and debug.

🔁 Comparison Table

Aspect	Without Destructuring	With Destructuring
Access syntax	`props.name`, `props.age`	`name`, `age`
Readability	👎 Cluttered with `props.` prefix	👍 Clear and focused
Code length	Longer	Shorter
Best in JSX?	No — looks noisy	Yes — looks clean
Common use?	Less common	✅ Preferred in modern React

🛠️ Without Destructuring

jsx Copy Edit

```jsx
function Welcome(props) {
  return (
    <h1>
      Hello {props.name}, you are {props.age} years old!
    </h1>
  );
}
```

- You access values with `props.name`, `props.age`, etc.
- Works fine — but becomes **verbose** with many props.

✨ With Destructuring

✅ Inside the function body:

```jsx
function Welcome(props) {
  const { name, age } = props;

  return (
    <h1>
      Hello {name}, you are {age} years old!
    </h1>
  );
}
```

- You pull out `name` and `age` from `props` into their own variables.
- This makes the JSX cleaner.

✅ Even Cleaner: Destructure in the parameter list

```jsx
function Welcome({ name, age }) {
  return (
    <h1>
      Hello {name}, you are {age} years old!
    </h1>
  );
}
```

- This is the **most common** and cleanest way.
- You skip `props` entirely and pull out only what you need.

LIST RENDERING

📚 Rendering a List in React

To render a list of items in React, you typically:

1. Use the `.map()` function to loop over an array.
2. Return **JSX** for each item.
3. Provide a **unique** `key` **prop** to each list item.

Why Use a key Prop?

The key helps React identify which items changed, were added, or removed, which improves performance and avoids rendering bugs.

- 🔑 Key should be unique and stable.
- Avoid using index as key if the list can change.

🎯 Key Concepts Explained

1. `.map()`

- A JavaScript array method.
- Takes a callback and returns a new array.
- Used to convert an array of data → array of JSX.

```js
array.map(item => <JSX />)
```

2. `key` Prop

React needs a unique key to track each element:

```jsx
<li key={book.id}>...</li>
```

```jsx
function Menu() {
  return (
    <main className="menu">
      <h2>Our Menu</h2>
      <div>
        <ul className="pizzas">
          {pizzaData.map((pizza) => (
            <Pizza pizzaObject={pizza} key={pizza.name} />
          ))}
        </ul>
      </div>
    </main>
  );
}

function Pizza(props) {
  console.log(props);
  return (
    <li className="pizza">
      <img src={props.pizzaObject.photoName} alt={props.pizzaObject.name} />
      <div>
        <h3>{props.pizzaObject.name}</h3>
        <p>{props.pizzaObject.ingredients}</p>
        <span>{props.pizzaObject.price + 3}</span>
      </div>
    </li>
  );
}
```

🔁 `.map()` is **Pure**

What does "pure" mean?

> A **pure function** is one that **does not change anything outside itself** and **always returns a new value** based only on its input.

✅ `.map()`:

- Takes an array.
- Processes each element.
- **Returns a new array** with the transformed results.
- **Does not change** the original array.
- **No side effects.**

🧠 Example:

```
js                                                    Copy  Edit

const numbers = [1, 2, 3];
const doubled = numbers.map(n => n * 2);
console.log(doubled); // [2, 4, 6]
console.log(numbers); // [1, 2, 3] - original array unchanged
```

✔ This is **pure and declarative** — you declare what the new data should look like.

🔄 `.forEach()` is **Imperative**

What does "imperative" mean?

> An **imperative function** tells the program **how to do something**, often involving **side effects** (modifying something outside the function).

❌ `.forEach()`:

- Loops over an array.
- **Does not return anything.**
- Meant for running side effects (e.g., logging, changing variables).
- Often used when you're **doing something**, not **producing a new result**.

🧠 Example:

```js
const numbers = [1, 2, 3];
const doubled = [];

numbers.forEach(n => {
  doubled.push(n * 2); // Side effect: modifying `doubled` array
});

console.log(doubled); // [2, 4, 6]
```

✔ This is **imperative** — you manually push items, m ge state, and give instructions.

Why Use map() Instead of forEach() in React for Rendering Lists

Fundamental Difference

- `map()` returns a new array with transformed elements
- `forEach()` returns undefined and only executes a function for each element
- Always use map() when you need to render a list in JSX
- Use forEach only for side effects when you don't need the return value

🧠 Why Use `.map()` in React (Not `forEach`):

1. **React needs JSX to be returned**
   When you're rendering a list in React, you need to **return JSX elements** (like `<li>`, `<div>`, `<Card />`) so React knows what to display on the screen.
   - `.map()` creates and **returns a new array** of these elements.
   - `.forEach()` does **not return anything**. It just runs code for each item — which is useful for side effects, but not for rendering.

---

2. `.map()` **fits naturally in JSX**
   You can use `.map()` **directly inside JSX**:

```jsx
{users.map(user => <li>{user}</li>)}
```

   But `.forEach()` doesn't return a value, so you **can't use it like this.**

3. **Cleaner, more concise code**
   `.map()` allows you to transform data into UI in a **single line.**
   With `.forEach()`, you'd need to:
   - create an empty array,
   - manually push items into it,
   - and then return it.

---

4. **React community standard**
   Using `.map()` is the **standard and recommended** way to render lists in React. It's used in almost every React codebase and documentation, so it keeps your code **idiomatic and familiar** to other developers.

✅ Summary (in one sentence):

> We use `.map()` in React to render lists because it returns an array of JSX elements, which React uses to display content — while `.forEach()` does not return anything useful for rendering.

Conditional Rendering?

Conditional rendering means showing different UI / div / some portion based on a condition — like whether a user is logged in, or if a shop is open.

- React uses JavaScript logic (like if, ?:, &&) inside JSX to decide what to render.

1) Ternary Operator condition ? trueUI : falseUI

```
{isLoggedIn ? <p>Welcome back!</p> : <p>Please log in.</p>}
```

2) Logical AND (&&) — render only if condition is true

```
{isOpen && <p>The shop is open!</p>}
```

3) if...else (outside JSX)

- Use this if the condition is complex or the JSX is long.

```
let content;
if (isOpen) {
  content = <p>We are open</p>;
} else {
  content = <p>We are closed</p>;
}

return <div>{content}</div>;
```

4) Early return (guard clause)

```
if (!isOpen) return <p>Closed</p>;

return <p>We're open!</p>;
```

🧩 What are React Fragments?

A React Fragment lets you group multiple elements without adding an extra node to the DOM.

🤔 Why Use Fragments?

Normally in React, components must return a single parent element. If you want to return multiple siblings, you usually wrap them in a `<div>` — but that adds unnecessary HTML to the DOM.

🚫 Without Fragment:

jsx Copy Edit

```jsx
return (
  <div>
    <h1>Hello</h1>
    <p>Welcome</p>
  </div>
);
```

This adds an **extra** `<div>` to the DOM.

✅ With Fragment:

jsx Copy Edit

```jsx
return (
  <>
    <h1>Hello</h1>
    <p>Welcome</p>
  </>
);
```

This adds **no extra HTML** — just the `<h1>` and `<p>`.

🔑 Where to Use Fragments?

1. Inside .map() when rendering a list of elements that don't need a wrapper

```jsx
const items = ["Pizza", "Pasta"];

return (
  <ul>
    {items.map((item, index) => (
      <React.Fragment key={index}>
        <li>{item}</li>
        <hr />
      </React.Fragment>
    ))}
  </ul>
);
```

2. Inside a component when returning multiple sibling elements.

```jsx
function Welcome() {
  return (
```

```
    <>
      <h1>Hi!</h1>
      <p>Welcome to our store.</p>
    </>
  );
}
```

3. In tables, where adding extra
   would break structure — instead use fragments to return and properly.

```
function Row() {
  return (
    <div>
      <tr>
        <td>Pizza</td>
        <td>$10</td>
      </tr>
    </div>
  );
}
```