

# DAY 1: React – COMPLETE CONCEPTUAL FOUNDATION

---

## 1 Why should you learn React?

### ♦ a) Hype, Jobs, Trend

Why React is popular?

- React is used by **Facebook, Instagram, Netflix, WhatsApp, Uber**
- Huge **job market**
- Most frontend roles ask for React

👉 **Reality:**

React is popular **not because of hype**, but because it **solves real frontend problems**.

---

### ♦ b) Build UI Easily

Before React:

- HTML + CSS + JS
- Manual DOM manipulation
- Hard to manage large UI

With React:

- UI is **component-based**

- You think in **pieces**, not full pages

Example:

Page

```
|— Navbar
|— Sidebar
|— Content
└— Footer
```

Each is a **component**.

---

### ♦ c) Manage Complex Frontend

Modern apps have:

- Login state
- Notifications
- Real-time updates
- API data
- Forms

React helps:

- Keep **UI consistent**
  - Automatically update UI when data changes
- 

## 2 When should you learn React?

✗ NOT immediately

✓ Learn React AFTER mastering JavaScript

You must know:

- Variables (`let`, `const`)
- Functions
- Arrays & objects
- `map`, `filter`, `reduce`
- Promises, `async/await`
- DOM (`document.querySelector`)
- Events

👉 Why?

React is **JavaScript-heavy**.

If JS is weak → React becomes confusing.

---

## ! Important Truth

**Most projects DO NOT need React in the beginning**

Small websites:

- Portfolio
- Blog
- Static pages

👉 Use **HTML + CSS + JS**

React is needed when:

- UI becomes **dynamic**
  - App grows large
  - Many states & interactions
- 

### 3 Why React was created?

#### 🔥 The Core Problem: UI Inconsistency

##### Example: Ghost Message Problem

You see:

🔔 Notifications (3)

You open one message:

🔔 Notifications (2)

Open another:

🔔 Notifications (1)

Problem earlier:

- UI count not syncing properly
- Manual DOM updates
- Bugs everywhere

This is called **UI out of sync with data**

---

## ❌ Old Way (Before React)

- JS changes data
- Developer manually updates DOM
- Easy to forget updates
- UI becomes inconsistent

---

## ✅ React Solution

**State** → **UI**

If state changes → UI updates automatically

State (JS Data)



React



DOM (UI)

You **never touch DOM directly**.

---

## 4 Who created React?

- Created at **Facebook**
- One of the key creators: **Jordan Walke**
- Used internally first

## Adoption:

- **Khan Academy** adopted React
  - **Unsplash** adopted React
  - Then public release
  - React exploded 🚀
- 

## 5 State, DOM, User – Relationship

### Old Flow

User → DOM → JS

Messy, error-prone

---

### React Flow

User → Event → State

State → React → DOM

✅ Single source of truth = **State**

---

## 6 Don't learn React if...

❌ If you don't know:

- JavaScript fundamentals
- DOM working

- Browser internals
- 

## **Browser Inner Working (Basic)**

When you load a website:

1. HTML → DOM Tree
2. CSS → CSSOM
3. JS executes
4. DOM + CSSOM → Render Tree
5. Paint on screen

React sits **between JS & DOM**

It optimizes updates.

---

## **React Learning Process (Correct Way)**

### **Wrong Way**

- Watching tutorials only
  - Copy-paste code
  - No understanding
- 

### **Right Way**

**Step 1: Go Deep (Concepts)**

You must understand:

---

### ◆ Babel

- Browser doesn't understand JSX
- Babel converts:

```
<h1>Hello</h1>
```

into:

```
React.createElement("h1", null, "Hello")
```

---

### ◆ Virtual DOM

- React keeps a **virtual copy** of DOM
- Changes are applied there first

Why?

- Faster
  - Less direct DOM manipulation
- 

### ◆ Diffing Algorithm

- React compares old Virtual DOM with new one
- Finds **minimum changes**

- Updates only changed nodes

This is why React is fast ⚡

---

### ♦ **Fiber Architecture**

- New React engine
- Breaks rendering into small chunks
- Allows pause, resume, prioritize updates

Important for:

- Animations
  - Smooth UI
  - Large apps
- 

### ♦ **Hydration**

Used in frameworks like Next.js

- HTML comes from server
  - React attaches event listeners on client
  - Makes page interactive
- 

## **Step 2: Learn by Projects**

One topic → One project

Examples:

- Todo App → State
  - Calculator → Events
  - GitHub API → API + Async
  - Login App → Forms + Auth
- 

## 8 React is a Library (NOT a Framework)

### Library vs Framework

Feature	Library	Framework
Control	You control	Framework controls
Flexibility	High	Less
Example	React	Angular

👉 React only handles UI

---

## 9 Core Topics to Learn in React

---

### ♦ State (Heart of React)

State = data that changes

Example:

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

Change state → UI updates

---

### ◆ **JSX**

JS + HTML together

```
<h1>{count}</h1>
```

Makes UI readable & expressive

---

### ◆ **Component Reusability**

One component → use many times

```
<Button />  
<Button />  
<Button />
```

---

### ◆ **Props (Passing Data)**

Props = data passed to component

```
<User name="Vivekanand" />
```

Props are **read-only**

---

### ◆ **Hooks (Propagate Change)**

Hooks let you:

- Manage state

- Run side effects
- Access lifecycle

Important hooks:

- `useState`
- `useEffect`
- `useContext`
- `useRef`

---

## 10 Additional Add-ons to React

---

### ♦ Router

React has **NO** routing

Use:

- React Router

Allows:

```
/login  
/dashboard  
/profile
```

---

### ♦ State Management

React has **local state only**

For global state:

- Redux
- Redux Toolkit
- Zustand
- Context API

Used when:

- Many components need same data

---

## ♦ **Class Based Components**

Old way:

```
class App extends React.Component {}
```

Used in:

- Legacy projects
- Old codebases

---

## ♦ **BaaS Apps**

Backend as a Service:

- Firebase
- Supabase

Used for:

- Social media clones
  - E-commerce
  - Chat apps
- 

## Alternatives & Frameworks

### React Limitations

- No SEO
  - No server-side rendering
  - No routing
  - JS-heavy
- 

### Frameworks Built on React

- **Next.js** → SEO + SSR
  - **Gatsby** → Static sites
  - **Remix** → Modern full-stack
- 

## FINAL MINDSET (VERY IMPORTANT)

- 👉 **React is NOT magic**
- 👉 **JavaScript is KING**
- 👉 **React is a tool**

First:

- ✓ JavaScript
- ✓ Browser
- ✓ DOM

Then:

- ✓ React
- ✓ Projects
- ✓ Advanced concepts

=====

## **PART 1: CODE EDITOR & RUNTIME**

---

### **1 Code Editor → VS Code**

#### ♦ **What is VS Code?**

**Visual Studio Code** is a **code editor**, not a compiler.

#### ♦ **Why VS Code?**

- Fast & lightweight
- Huge extension support
- Best for JavaScript & React
- Auto-suggest, linting, formatting

### ♦ VS Code vs Notepad

Feature	VS Code	Notepad
Syntax Highlighting	✓	✗
Auto Complete	✓	✗
Extensions	✓	✗
Terminal	✓	✗

👉 Industry standard for React

---

## 2 Node.js (Important Concept)

### ♦ What is Node.js?

Node.js lets you **run JavaScript outside the browser**.

Before Node:

- JS only ran in browser

After Node:

- JS can run on server
  - JS can install tools (React, Vite)
- 

## 3 Compiler vs Interpreter (Very Important)

### ♦ Compiler

- Converts **whole code at once**
- Example: C, C++
- Faster execution
- Errors shown after compilation

#### ♦ Interpreter

- Executes **line by line**
- Example: JavaScript, Python
- Errors shown immediately

#### ♦ JavaScript?

JavaScript is **interpreted + JIT compiled**

- Browser & Node optimize it internally

👉 React tools need **Node.js**, not browser JS.



## PART 2: REACT DOCUMENTATION

---

### 4 React Documentation (Old vs New)

#### ✗ Old Docs

- reactjs.org (deprecated)
- Class-based focus

- Confusing lifecycle methods

## ✅ New Docs

### 👉 react.dev

- Hook-based
  - Practical examples
  - Beginner friendly
- 

## 5 How to Read React Documentation (VERY IMPORTANT)

### ❌ Wrong Way

- Random scrolling
- Copy-paste code
- No understanding

### ✅ Correct Way

Follow order:

1. **Thinking in React**
2. **Your First Component**
3. **State & Props**
4. **Hooks**
5. **Managing State**

## 6. Escape Hatches

👉 Read **concept first**, then **example**, then **try yourself**

---

# PART 3: PROJECT & GITHUB SETUP

---

## 6 Create Folder

```
mkdir for-freelance-react  
cd for-freelance-react
```

This folder will contain **all your React learning & freelance projects**.

---

## 7 Create GitHub Repository

### ♦ What is GitHub?

**GitHub** is used to:

- Store code online
- Version control
- Collaboration
- Portfolio for jobs/freelancing

Repository name:

```
for-freelance-react
```

---

## 8 Push Existing Project to GitHub (Terminal)

Step-by-step:

```
git init
```

👉 Initializes git in your project

```
git add .
```

👉 Stages all files

```
git commit -m "Initial React setup"
```

👉 Saves snapshot

```
git push
```

👉 Uploads code to GitHub



## PART 4: WHAT IS REACT ACTUALLY?

---

### 9 React Types

- ♦ **react**

Core UI library

- ♦ **react-dom**

Used for **websites**

- Connects React → Browser DOM

### ♦ **react-native**

Used for **mobile apps**

- React logic + native UI

👉 Same logic, different platform

---



## **PART 5: CREATING A REACT APP**

---

### ♦ **npm vs npx (IMPORTANT)**

#### **npm**

- Package manager
- Installs libraries

#### **npx**

- Executes packages without installing globally

Example:

```
npx create-react-app myApp
```

---

### ♦ **create-react-app (CRA)**

What is it?

- A **utility tool**

- Not software, not framework
- Pre-configured React environment

### **Problems:**

- Heavy
- Slow
- Old tooling
- Large build size

👉 **Avoid in 2025**

---

## ◆ **Vite / Parcel (Modern Tools)**

### **Why Vite?**

- Instant startup
- Smaller bundle
- Faster HMR (hot reload)
- Uses ES modules

### **Why Parcel?**

- Zero config
- Good but slower than Vite

👉 **Industry choice: Vite**



# PART 6: PROJECT STRUCTURE & package.json

---

## 10 package.json (Heart of Project)

Open `package.json`

---

### ◆ name & version

Project identity

---

### ◆ dependencies

Libraries needed in production:

- react
  - react-dom
- 

### ◆ devDependencies

Used only during development:

- linters
- bundlers
- formatters

✗ Not shipped to production

---

### ♦ **scripts (VERY IMPORTANT)**

Script	Meaning
start	Run dev server
build	Create production files
test	Run tests
eject	Expose internal config (dangerous)

---

### ♦ **react-scripts**

Used by CRA:

- Build
  - Start
  - Test
  - Bundle
- 

### ♦ **web-vitals**

Performance metrics:

- CLS
- LCP

- FID

Used to track real user performance

---

### ♦ **eslintConfig**

Linting rules

Red underline = **lint error**, not runtime error

---

### ♦ **browserslist**

Tells React:

- Which browsers to support
- Compatibility settings

Example:

`"last 2 chrome versions"`

---

## **PART 7: RUNNING & BUILDING PROJECT**

---

### **11** How to Run Project

Check `package.json` → scripts

`npm start`

or

```
npm run dev
```

---

## 12 npm install

What it does?

- Reads `package.json`
- Installs all dependencies
- Creates `node_modules`

Without this → project won't run

---

## 13 Build for Production

```
npm run build
```

Creates:

```
/build
```

This folder:

- Optimized
  - Minified
  - Used in hosting (Netlify, Vercel)
-



# FINAL SUMMARY (MUST REMEMBER)

- ✓ VS Code = Editor
- ✓ Node.js = JS runtime
- ✓ React = UI library
- ✓ Vite = Build tool
- ✓ npm = Package manager
- ✓ npx = Execute tools
- ✓ GitHub = Code hosting
- ✓ build folder = Production

=====



## REACT FILE NAMING, ENTRY POINT & RENDERING

*(From Beginner → Advanced)*

### 1 React is a Library, not a Framework — Why Naming is Flexible

- ◆ What does “React is a library” mean?

A **library** gives you tools, but **you decide structure**.

- React only cares about **UI**
- It does **not force**:
  - Folder structure

- File naming
- Routing
- Build process

👉 That's why React is **flexible**.

---

## 2 Why do we usually name files **App.js** or **App.jsx**?

### ♦ Convention vs Rule

Thing	Is it mandatory?
App.js	✗ No
App.jsx	✗ No
Default export	✓ Yes
Component function	✓ Yes

👉 **App** is just a convention, not a React rule.

You can name it:

- Main.jsx
- Root.jsx
- Home.jsx
- Vivek.jsx

React **does not care**.

---

### 3 .js vs .jsx — What is the real difference?

#### ♦ .js

- JavaScript file
- Can contain JSX
- Browser doesn't understand JSX directly

#### ♦ .jsx

- Signals that file contains JSX
- Better readability
- Better editor support
- Industry best practice

👉 Both work, but .jsx is recommended

---

### 4 Does Vite affect naming?

#### ♦ With Vite:

- JSX files should be .jsx
- Entry file is main.jsx

Vite expects:

```
ReactDOM.createRoot(...).render(...)
```

- 

👉 Naming is still **flexible**, but:

- `main.jsx` → entry point
  - `App.jsx` → root component (by convention)
- 

## PART 2: UNDERSTANDING `App.jsx`

---

### 5 Code inside `src/App.jsx`

```
function App(){  
  return (  
    <h1>code with vivek-freelancing youtube channel | HC</h1>  
  )  
}
```

```
export default App
```

Let's break it **line by line**.

---

#### ♦ `function App()`

- This is a **React Component**
- Component = JavaScript function that returns JSX
- Must start with **capital letter**

✗ Wrong:

```
function app() {}
```

✓ Correct:

```
function App() {}
```

Why?

- React treats lowercase tags as HTML
  - Capitalized names are treated as components
- 

#### ♦ **return (...)**

- A component **must return UI**
  - UI is written using **JSX**
  - JSX looks like HTML but is **JavaScript**
- 

#### ♦ **<h1>...</h1>**

This is **JSX**, not HTML.

Behind the scenes:

```
React.createElement("h1", null, "code with vivek...")
```

---

#### ♦ **Why only ONE parent element?**

This is valid:

```
return <h1>Hello</h1>
```

This is ❌ invalid:

```
return (  
  <h1>Hello</h1>  
  <p>World</p>  
)
```

Correct way:

```
return (  
  <>  
    <h1>Hello</h1>  
    <p>World</p>  
  </>  
)
```

---

#### ♦ export default App

- Makes App available to other files
- Required so main.jsx can import it

---

## PART 3: UNDERSTANDING main.jsx (MOST IMPORTANT)

---

### Code inside src/main.jsx

```
import React from 'react'
```

```
import ReactDOM from 'react-dom/client'  
import App from './App.jsx'
```

---

#### ♦ `import React from 'react'`

- Imports React core
  - Older versions **required this for JSX**
  - New versions don't strictly need it
  - Still good practice
- 

#### ♦ `import ReactDOM from 'react-dom/client'`

- This connects **React** → **Browser DOM**
  - Without this → React cannot render UI
- 

#### ♦ `import App from './App.jsx'`

- Imports your root component
  - `./` → same folder
  - Extension `.jsx` is optional but recommended
- 

## **7** `createRoot` — Modern React Rendering

```
ReactDOM.createRoot(document.getElementById('root'))
```

### ♦ What is **root**?

In `index.html`:

```
<div id="root"></div>
```

React **injects UI here**.

---

### ♦ Why **createRoot**?

Old React:

```
ReactDOM.render(<App />, root)
```

New React:

```
createRoot(root).render(...)
```

Benefits:

- Concurrent rendering
  - Better performance
  - Future-ready
- 

## 8 **.render()** — Rendering the App

```
.render(  
  <React.StrictMode>  
    <App />  
  </React.StrictMode>
```

)

---

### ♦ **<App /> — Component Usage**

- **<App />** means **execute App function**
- Returns JSX
- JSX gets rendered to DOM

---

### ♦ **Why <App /> not App() ?**

React manages:

- Re-rendering
- State updates
- Lifecycle

Calling **App()** manually breaks React rules.

---

## **9** **React.StrictMode — Advanced Concept**

### ♦ **What is StrictMode?**

- Development-only tool
- Helps find:
  - Unsafe lifecycle usage

- Side-effect bugs
- Deprecated APIs

---

◆ **What it does internally?**

- Renders components **twice** (only in dev)
- Helps catch bugs early

👉 It does **NOT** affect production

---

## 10 Complete Flow (VERY IMPORTANT)

Browser loads `index.html`

↓

`<div id="root"></div>`

↓

`main.jsx` runs

↓

`ReactDOM.createRoot()`

↓

`<App />` executes

↓

JSX → Virtual DOM

↓

Diffing

↓

Real DOM update

---

# ADVANCED INSIGHTS (INTERVIEW LEVEL)

---

? Can we rename `main.jsx`?

✗ No (Vite expects it)

? Can we rename `App.jsx`?

✓ Yes

? Is `React.StrictMode` mandatory?

✗ No, but recommended

? Is `JSX` required?

✗ No (but nobody avoids it)

---

## FINAL SUMMARY

Concept	Meaning
React is library	Flexible structure
App.jsx	Root UI component
main.jsx	Entry point
JSX	JS + HTML
createRoot	Modern rendering
StrictMode	Dev safety

=====

**line-by-line explanation** of your **Custom React (mini React)** implementation.  
This topic is **VERY IMPORTANT** because it teaches **how React actually works internally** instead of just using it.

I'll explain:

- **Why we build custom React**
- **HTML file role**
- **JS file role**
- **Virtual DOM idea**
- **customRender logic**
- **How this relates to real React**
- **Limitations & advanced insights**

---

## **WHY ARE WE BUILDING “CUSTOM REACT”?**

Before using real React, you must understand:

**? What does React actually do behind the scenes?**

At its core, React:

1. Takes an **object description of UI**

2. Converts it into **real DOM**
3. Injects it into the browser

Your **customReact** is a **mini React engine** that mimics this behavior.

---

## FOLDER STRUCTURE

```
customReact/  
|  
├─ index.html  
└─ customreact.js
```

This separation itself teaches an important concept:

- **HTML** → Entry point
  - **JS** → Rendering logic
- 

## PART 1: index.html (ENTRY POINT)

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <meta charset="UTF-8">  
  <meta name="viewport" content="width=device-width,  
initial-scale=1.0">  
  <title>Custom React App</title>  
</head>  
<body>  
  <div id="root"></div>
```

```
<script src="./customreact.js"></script>
</body>
</html>
```

Let's break this **line by line**.

---

## 1 <!DOCTYPE html>

- Tells browser: this is **HTML5**
  - Without it → browser may behave unpredictably
- 

## 2 <html lang="en">

- Root element
  - `lang="en"` helps:
    - Screen readers
    - SEO
    - Accessibility
- 

## 3 <meta charset="UTF-8">

- Allows all characters (₹, हिंदी, emojis, etc.)
- Without this → text may break

---

#### 4 <meta name="viewport">

content="width=device-width, initial-scale=1.0"

- Makes page **responsive**
- Important for mobile devices

---

#### 5 <div id="root"></div>

##### MOST IMPORTANT LINE

This is where:

- Your **entire app UI will appear**
- Same concept as React's root

In real React:

```
<div id="root"></div>
```

React attaches UI here.

Your custom React does **exactly the same thing**.

---

#### 6 <script src="./customreact.js"></script>

- Loads JavaScript **after HTML**
- JS will:

- Create elements
  - Inject into `#root`
- 

## PART 2: customreact.js (THE MINI REACT ENGINE)

---

### 1 customRender FUNCTION

```
function customRender(reactElement, container){
```

This function is equivalent to:

```
ReactDOM.render()
```

or

```
createRoot().render()
```

**Parameters:**

- `reactElement` → virtual description of UI
  - `container` → actual DOM node (`#root`)
- 

### 2 Creating Real DOM Element

```
const domElement = document.createElement(reactElement.type)
```

## What is happening?

If:

```
type: 'a'
```

This becomes:

```
document.createElement('a')
```

👉 This is **Virtual DOM** → **Real DOM** conversion

---

## 3 Adding Children (Text)

```
domElement.innerHTML = reactElement.children
```

If:

```
children: 'Click me to visit google'
```

Then:

```
<a>Click me to visit google</a>
```

⚠ Limitation:

- Only supports **text**
- Real React supports:
  - Nested elements
  - Arrays
  - Components

---

## 4 Setting Attributes (Props)

```
domElement.setAttribute('href', reactElement.props.href)
domElement.setAttribute('target', reactElement.props.target)
```

This mimics React props:

```
<a href="..." target="...">...</a>
```

🔥 Key idea:

**Props are just attributes on DOM elements**

React automates this process.

---

## 5 Append to DOM

```
container.append(domElement)
```

- Inserts element into `#root`
  - Browser finally displays it
- 

# PART 3: THE VIRTUAL ELEMENT OBJECT

```
const reactElement = {
  type: 'a',
  props: {
    href: 'https://google.com',
```

```
    target: '_blank'
  },
  children: 'Click me to visit google'
}
```

This is a **Virtual DOM object**.

---

## 1 type

```
type: 'a'
```

Means:

```
<a></a>
```

In real React:

```
<a />
```

---

## 2 props

```
props: {
  href: 'https://google.com',
  target: '_blank'
}
```

Props = configuration of element

Real React equivalent:

```
<a href="https://google.com" target="_blank">
```

---

### **3** children

```
children: 'Click me to visit google'
```

Content inside element

---

## **PART 4: SELECTING ROOT & RENDERING**

```
const mainContainer = document.querySelector('#root')
```

- Selects `<div id="root"></div>`
  - Same as ReactDOM targeting root
- 

```
customRender(reactElement, mainContainer)
```

 This is your **render pipeline**

---

## **COMPLETE FLOW (VERY IMPORTANT)**

```
index.html loads
```

↓

```
#root exists
```

↓

```
customreact.js runs
```

↓

```
Virtual Element Object created
```

↓

customRender() called



Real DOM element created



Props applied



Element appended to #root



Browser paints UI



## HOW THIS RELATES TO REAL REACT

### Your Custom React

### Real React

reactElement object

JSX

customRender

ReactDOM.render

document.createElement

Virtual DOM diffing

manual props

automatic props

single element

component tree

Real React adds:

- Diffing algorithm
- Re-rendering
- State
- Hooks
- Fiber architecture

---

## LIMITATIONS OF YOUR CUSTOM REACT

- ✗ No state
- ✗ No re-rendering
- ✗ No nested elements
- ✗ No event handling
- ✗ No diffing

But...

- ✓ Concept is 100% correct
- 

## ADVANCED THINKING (INTERVIEW GOLD)

### ? Why React uses objects?

Because:

- JS objects are cheap
- Easy to compare (diffing)
- Declarative UI

### ? Why not manipulate DOM directly?

Because:

- DOM is slow

- Error-prone
- Hard to scale



## FINAL SUMMARY

- ✓ React = object → DOM
- ✓ JSX = syntactic sugar for objects
- ✓ Rendering = DOM creation
- ✓ Props = attributes
- ✓ Root = injection point

=====

Great — this version of **customReact** is actually **much closer to how real React works**.

I'll explain **everything step by step**, starting from **why this change was made**, then **each line**, then **how this maps to real React internals**, and finally **advanced insights + limitations**.

---



## WHY THIS VERSION IS IMPORTANT

Earlier, you were doing this:

```
domElement.setAttribute('href', reactElement.props.href)
domElement.setAttribute('target', reactElement.props.target)
```

✗ Problem:

- Hard-coded attributes
- Works only for `<a>`
- Not scalable

✓ New version:

- **Generic**
- Works for **any HTML element**
- Very close to React's internal render logic

---

## FULL CODE (REFERENCE)

```
function customRender(reactElement, container){
  const domElement = document.createElement(reactElement.type)
  domElement.innerHTML = reactElement.children

  for (const prop in reactElement.props) {
    if (prop === 'children') continue;
    domElement.setAttribute(prop, reactElement.props[prop])
  }

  container.appendChild(domElement)
}

const reactElement = {
  type: 'a',
  props: {
    href: 'https://google.com',
    target: '_blank'
  }
}
```

```
    },  
    children: 'Click me to visit google'  
  }  
}
```

```
const mainContainer = document.querySelector('#root')  
customRender(reactElement, mainContainer)
```

---

## STEP-BY-STEP EXPLANATION (BASIC → ADVANCED)

---

### **1** customRender – Your Mini ReactDOM.render

```
function customRender(reactElement, container)
```

This function plays the role of:

```
ReactDOM.createRoot(container).render(<App />)
```

#### Parameters:

Parameter	Meaning
<code>reactElement</code>	Virtual description of UI
<code>container</code>	Real DOM node ( <code>#root</code> )

---

### **2** Creating the Real DOM Element

```
const domElement = document.createElement(reactElement.type)
```

## What happens here?

If:

```
reactElement.type === 'a'
```

Then browser executes:

```
document.createElement('a')
```

📌 This is the **core React idea**:

React never starts with HTML — it starts with **objects**

---

## 3 Adding Content (Children)

```
domElement.innerHTML = reactElement.children
```

If:

```
children: 'Click me to visit google'
```

DOM becomes:

```
<a>Click me to visit google</a>
```

### ⚠ Important limitation (for now):

- `children` only supports **text**
- Real React supports:
  - Arrays

- Elements
- Components
- Nested trees

We'll fix this later conceptually.

---

## 4 The **for...in** Loop – MOST IMPORTANT PART

```
for (const prop in reactElement.props) {
```

**What does this do?**

It loops through:

```
{  
  href: 'https://google.com',  
  target: '_blank'  
}
```

One by one:

- `prop = "href"`
- `prop = "target"`

This is how React handles **dynamic props**.

---

## 5 Why this line exists

```
if (prop === 'children') continue;
```

## Why skip `children`?

Because:

- `children` is **not an HTML attribute**
- This would be invalid:

```
<a children="Click me"></a>
```

📌 In real React:

- `children` is handled **separately**
- Attributes & children follow different paths

---

## 6 Applying Attributes Dynamically

```
domElement.setAttribute(prop, reactElement.props[prop])
```

This line translates object props → DOM attributes.

### Example:

```
setAttribute("href", "https://google.com")  
setAttribute("target", "_blank")
```

Which produces:

```
<a href="https://google.com" target="_blank">
```

🔥 This is exactly how React maps JSX → DOM.

---

## 7 Appending to the DOM

```
container.appendChild(domElement)
```

- Adds element inside `<div id="root"></div>`
- Browser paints the UI

📌 React does the same, but with:

- Diffing
  - Scheduling
  - Performance optimization
- 

## 🧠 UNDERSTANDING `reactElement` OBJECT (VIRTUAL DOM)

```
const reactElement = {  
  type: 'a',  
  props: {  
    href: 'https://google.com',  
    target: '_blank'  
  },  
  children: 'Click me to visit google'  
}
```

This is a **manual JSX replacement**.

**JSX equivalent:**

```
<a href="https://google.com" target="_blank">
```

`Click me to visit google`  
`</a>`

Behind the scenes, JSX compiles into **objects like this**.

---



## COMPLETE EXECUTION FLOW (VERY IMPORTANT)

`index.html` loads



`<div id="root"></div>`



`customreact.js` executes



Virtual element object created



`customRender()` called



DOM element created



Props applied



Children applied



DOM appended



Browser renders UI

---



# HOW CLOSE IS THIS TO REAL REACT?

Your Code	Real React
createElement object	JSX AST
customRender	Fiber render phase
createElement	Host config
setAttribute	DOM property mapping
appendChild	Commit phase

🔥 Conceptually: **90% correct**

---



## CURRENT LIMITATIONS (IMPORTANT)

- ✗ No re-rendering
- ✗ No state
- ✗ No events (`onClick`)
- ✗ No nested children
- ✗ No diffing

React exists to solve **these exact problems**.

---



## ADVANCED INSIGHT (INTERVIEW LEVEL)

? Why React uses objects instead of HTML?

Because:

- Objects are easy to compare
- Can detect changes
- Enables diffing
- Enables scheduling

### ? Why not manipulate DOM directly?

- DOM operations are expensive
- Hard to track UI consistency
- Error-prone in large apps

---

## FINAL SUMMARY

- ✓ This is a **generic renderer**
- ✓ Props handled dynamically
- ✓ Children handled separately
- ✓ Root injection same as React
- ✓ Concept matches real React

=====

This topic is **VERY IMPORTANT** because it answers:

? *What exactly can React render? JSX? Object? Function? Component?*

---



# STEP 7 — WHAT CAN REACT RENDER?

React can render **only one thing**:

👉 **A React Element**

Everything else is **converted into a React Element**.

We'll prove this step by step.

---

## PART **1** `src/main.jsx` — ROOT & RENDERING LOGIC

### ◆ Code

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
```

### What this means

- `react` → core library (JSX, hooks, components)
  - `react-dom/client` → connects React to browser DOM
  - `App` → root component (optional, replaceable)
- 

## PART **2** FUNCTION COMPONENT — `MyApp`

```
function MyApp(){
  return (
    <div>
      <h1>Custom App | hello</h1>
    </div>
  )
}
```

```
    </div>
  )
}
```

#### ♦ What is this?

- A **function component**
- Must start with **capital letter**
- Must **return JSX**

#### ♦ JSX here is NOT HTML

Behind the scenes:

```
React.createElement(
  "div",
  null,
  React.createElement("h1", null, "Custom App | hello")
)
```

👉 JSX → React Element → Virtual DOM

---

## PART 3 PLAIN OBJECT — **ReactElement**

```
const ReactElement = {
  type: 'a',
  props: {
    href: 'https://google.com',
    target: '_blank'
  },
  children: 'Click me to visit google'
}
```

#### ♦ What is this?

- A manual **React Element**
- Similar to what JSX compiles into
- **NOT JSX**, just a JS object

⚠ Important:

React **cannot render this object directly**

It must go through `React.createElement`.

---

## PART 4 JSX ELEMENT — `anotherElement`

```
const anotherElement = (  
  <a href="https://google.com" target="_blank">  
    Visit google  
  </a>  
)
```

#### ♦ What is this?

- JSX Element
- Already compiled into a **React Element**
- Safe to render directly

👉 This is the **most common React usage**

---

## PART 5 THE MOST IMPORTANT LINE — `.render(...)`

```
ReactDOM.createRoot(document.getElementById('root')).render(  
  anotherElement or ReactElement or MyApp()  
)
```

Let's break this carefully.

---

### ● CASE 1: `anotherElement` ✓ (CORRECT)

```
render(anotherElement)
```

- ✓ JSX element
  - ✓ Valid React Element
  - ✓ React renders it perfectly
- 

### ● CASE 2: `ReactElement` ✗ (INVALID)

```
render(ReactElement)
```

- ✗ This is a **plain JS object**
- ✗ React doesn't know how to convert it
- ✗ Error: *Objects are not valid as a React child*

👉 React **never renders raw objects**

---

### ● CASE 3: `MyApp()` ✗ (WRONG)

```
render(MyApp())
```

Why this is wrong:

- `MyApp()` executes the function

- React loses control
- Hooks break
- Lifecycle breaks

✗ React must **call components itself**

---

✓ **CORRECT WAY**

```
render(<MyApp />)
```

This allows React to:

- Track component
  - Re-render
  - Manage hooks
- 

🧠 **GOLDEN RULE (INTERVIEW LEVEL)**

! React renders **React Elements**, NOT functions, NOT objects

Thing	Can React Render?
JSX Element	✓
<Component />	✓
Component( )	✗

Plain object 

String /  
number 

---

## PART 6 `src/App.jsx` — COMPONENT COMPOSITION

```
import Hello from "../Hello"

function App(){
  return (
    <>
      <Hello />
      <h1>hello from app.jsx</h1>
      <p>test paragraph from app.jsx</p>
    </>
  )
}

export default App
```

---

### ◆ **Fragment** `<>...</>`

Why fragment?

- JSX needs **one parent**
- Avoid extra `<div>`

Equivalent to:

```
<React.Fragment>...</React.Fragment>
```

---

## ◆ `<Hello />` — COMPONENT REUSE

- Components can call other components
- Builds **component tree**

Tree:

```
App
├─ Hello
├─ h1
└─ p
```

This is how React apps scale.

---

## PART 7 `src/Hello.jsx` — SIMPLE COMPONENT

```
function Hello(){
  return (
    <h3>Hello is ready</h3>
  )
}
```

```
export default Hello
```

### ◆ Key rules

- Capitalized name
  - Returns JSX
  - Exported for reuse
- 

## COMPLETE RENDER FLOW (VERY IMPORTANT)

index.html

↓

<div id="root"></div>

↓

main.jsx

↓

ReactDOM.createRoot()

↓

render(React Element)

↓

Component executes

↓

JSX → React Elements

↓

Virtual DOM

↓

Diffing

↓

Real DOM update

---



# ADVANCED CONCEPTS YOU JUST LEARNED

- ✓ JSX is NOT HTML
  - ✓ Components return React Elements
  - ✓ React controls execution
  - ✓ Plain objects are not renderable
  - ✓ `<Component /> ≠ Component()`
- 

## ! COMMON MISTAKES (VERY IMPORTANT)

- ✗ `render(MyApp())`
  - ✗ Rendering raw objects
  - ✗ Lowercase component names
  - ✗ Multiple root JSX elements
- 



## FINAL SUMMARY

Concept	Meaning
JSX	Syntax sugar
React Element	Renderable unit
Component	Function returning JSX
Fragment	Wrapper without DOM

render()      Accepts React Element  
only

=====

Below is a **deep, structured explanation from absolute basics** → **React internals / open-source level** of **everything** you wrote.

This is exactly the point where **you stop being a React user and start becoming a React engineer**.

I'll go in **layers**:

1. Simple component (`Hello.jsx`)
2. Rendering without JSX (`createElement`)
3. What is an *evaluated expression*
4. Why JSX exists & where it comes from
5. How to read React's open-source code
6. Babel's role (VERY IMPORTANT)
7. Internal `ReactElement` implementation (advanced)

---

## PART 1 `Hello.jsx` — SIMPLE FUNCTION COMPONENT

```
function Hello(){  
  return (  
    <h3>Hello is ready</h3>  
  )  
}
```

```
    )  
  }  
  export default Hello
```

## ◆ What is this?

- A function component
- A normal JavaScript function
- Returns **JSX**

## ◆ Why function name must be capital?

React rule:

- Lowercase → HTML tag
- Capital → React component

`<hello />` ❌ (HTML tag)  
`<Hello />` ✅ (Component)

---

## ◆ What does this return actually mean?

JSX:

```
<h3>Hello is ready</h3>
```

Internally becomes:

```
React.createElement("h3", null, "Hello is ready")
```

So:

Every component returns a React Element

---

## PART 2 `main.jsx` — NO JSX, PURE REACT

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
```

These are standard imports.

Now comes the **important part** 📌

---

### ◆ This line is IMPORTANT (JSX runtime)

```
import { jsx as _jsx } from "react/jsx-runtime.js"
```

Why does this exist?

- New React **does NOT** require **React** import for JSX
- Babel converts JSX into `_jsx()` calls

Example:

```
<h1>Hello</h1>
```

Becomes:

```
_jsx("h1", { children: "Hello" })
```

This is why this import exists.

---

## PART 3 Variable inside render logic

```
const anotherUser = "vivek freelance"
```

This is just a normal JS variable.  
Now let's use it in React 👉

---

## PART 4 `React.createElement` — CORE REACT API

```
const reactElement = React.createElement(  
  'a',  
  { href: 'https://google.com', target: '_blank' },  
  'click me to visit google',  
  anotherUser  
)
```

### ♦ What is `React.createElement`?

This is the **lowest-level React API**.

Everything else (JSX, components, fragments) eventually becomes this.

---

### ♦ Arguments breakdown

```
React.createElement(  
  type,  
  props,
```

```
    children  
  )
```

### 1 type

```
'a'
```

Means:

```
<a></a>
```

---

### 2 props

```
{ href: 'https://google.com', target: '_blank' }
```

These become HTML attributes.

---

### 3 children

```
'click me to visit google',  
anotherUser
```

#### ♦ IMPORTANT:

Children are **variadic arguments**

So this is valid:

```
React.createElement('div', null, "Hello", "World", 123)
```

---

## PART 5 What is an *evaluated expression*?

You asked:

👉 Can we write `if(true)` here? What is evaluated expression?

## ❌ NOT ALLOWED

```
if (true) { ... } // statement ❌
```

## ✅ ALLOWED

```
true && "Hello"  
condition ? "Yes" : "No"  
anotherUser
```

### ♦ Why?

Because React children must be:

- Values
- Expressions
- NOT statements

---

### ♦ Definition (IMPORTANT)

**Evaluated expression** = anything that returns a value

Examples:

```
anotherUser           // string  
5 + 5                 // number  
isLoggedIn && "Welcome" // expression
```

❌ Statements don't return values:

```
if {}
```

```
for {}  
while {}
```

---

## PART 6 Rendering the element

```
ReactDOM.createRoot(document.getElementById('root')).render(  
  reactElement  
)
```

♦ What can `.render()` accept?

- ✓ React Element
- ✓ JSX
- ✓ `<Component />`

- ✗ Plain object
  - ✗ Function call
- 

## PART 7 Can JSX replace `createElement`?

YES — JSX EXISTS ONLY TO REPLACE THIS 📌

```
React.createElement(...)
```

**JSX version:**

```
<a href="https://google.com" target="_blank">  
  click me to visit google {anotherUser}  
</a>
```

**JSX is:**

- Easier
  - Readable
  - Safer
  - Declarative
- 

## PART 8 WHERE DOES JSX COME FROM? (OPEN SOURCE)

You asked:

*Where can I find this code? I want to jump into open source.*

You are asking the **RIGHT QUESTION**.

---

### ◆ React is a JavaScript open-source library

Location:

```
react/  
└─ packages/  
    └─ react/  
        └─ src/  
            └─ jsx/  
                └─ ReactJSXElement.js
```

This file defines **what a React Element actually is**.

---

# PART 9 INTERNAL **ReactDOM** FUNCTION (ADVANCED)

```
function ReactDOM(type, key, ref, self, source, owner, props) {
```

This is the **factory** that creates React Elements.

---

## ◆ Core object structure

```
const element = {  
  $$typeof: REACT_ELEMENT_TYPE,  
  type,  
  key,  
  ref,  
  props,  
  _owner: owner,  
}
```

## ◆ Why **\$\$typeof**?

- Security feature
  - Helps React identify valid elements
  - Prevents XSS attacks
- 

## ◆ **type**

- 'div'

- 'a'
  - MyComponent
- 

### ◆ props

Everything inside JSX attributes + children.

---

### ◆ \_owner

Tracks **which component created this element**

Used for:

- Warnings
  - Debugging
  - DevTools
- 

## PART 10 `__DEV__` block (VERY IMPORTANT)

```
if (__DEV__) {
```

### ◆ Why this exists?

Development vs Production

In dev mode:

- Extra warnings
  - Validation
  - Freezing objects
  - Debugging helpers
- 

#### ♦ **Object.freeze**

```
Object.freeze(element.props);  
Object.freeze(element);
```

Prevents:

- Accidental mutation
  - Bugs
  - Side effects
- 

## PART **11** ROLE OF BABEL (CRITICAL)

Babel does:

1. Converts JSX → `_jsx()`
2. Removes `console.log` (prod)
3. Removes dev warnings
4. Optimizes loops

5. Dead code elimination

6. Inlines constants

**Example:**

```
<h1>Hello</h1>
```

Becomes:

```
_jsx("h1", { children: "Hello" })
```

---



## BIG PICTURE (MENTAL MODEL)

JSX

↓ (Babel)

\_jsx()

↓

ReactDOM()

↓

Virtual DOM

↓

Fiber

↓

Real DOM

---



## FINAL SUMMARY (MUST REMEMBER)

- ✓ JSX is NOT magic
- ✓ JSX compiles to `createElement`
- ✓ React Elements are plain objects
- ✓ Statements are NOT allowed in JSX

- ✓ Expressions ARE allowed
- ✓ Babel is the compiler
- ✓ ReactElement is the core data structure

=====

Upto 2:30 in lecture of react1 by chai aur code which is part1