Introduction to ReactJS

Dr. L.M. Jenila Livingston
Professor
VIT Chennai

Overview

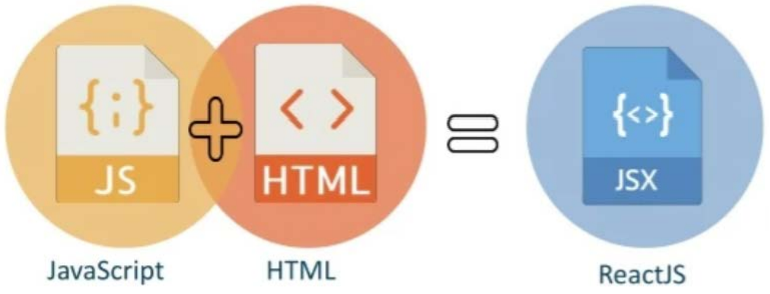
- React environment setup
- ReactJS basics
- React JSX
- JSX Vs HTML
- Rendering React Components
- React components:
  - React Functional component
  - React class component
  - API-Life Cycle
    - React constructors
    - Mounting
    - Updating
    - Unmounting
- React Dev Tools
- React Native Vs ReactJS

What is ReactJS?

- React is a **JavaScript library** for building user interfaces.
- React, sometimes referred to as a frontend JavaScript framework
- Developed by **Facebook** in 2013.
- Current version of create-react-app is **v5.0.1** (April 2022) which includes built tools such as webpack, Babel, and ESLint.
- Used for building **single-page applications** (SPA).
- **Component-based architecture** for building **reusable UI components**.

Prerequisites

- Knowledge of **HTML**, **CSS**, **JavaScript**, and **Git**
- Knowledge of package management with **Node.js** and **npm**
- **Node.js** and **Node Package Manager (npm)** locally installed
- A code editor, such as **Visual Studio Code**



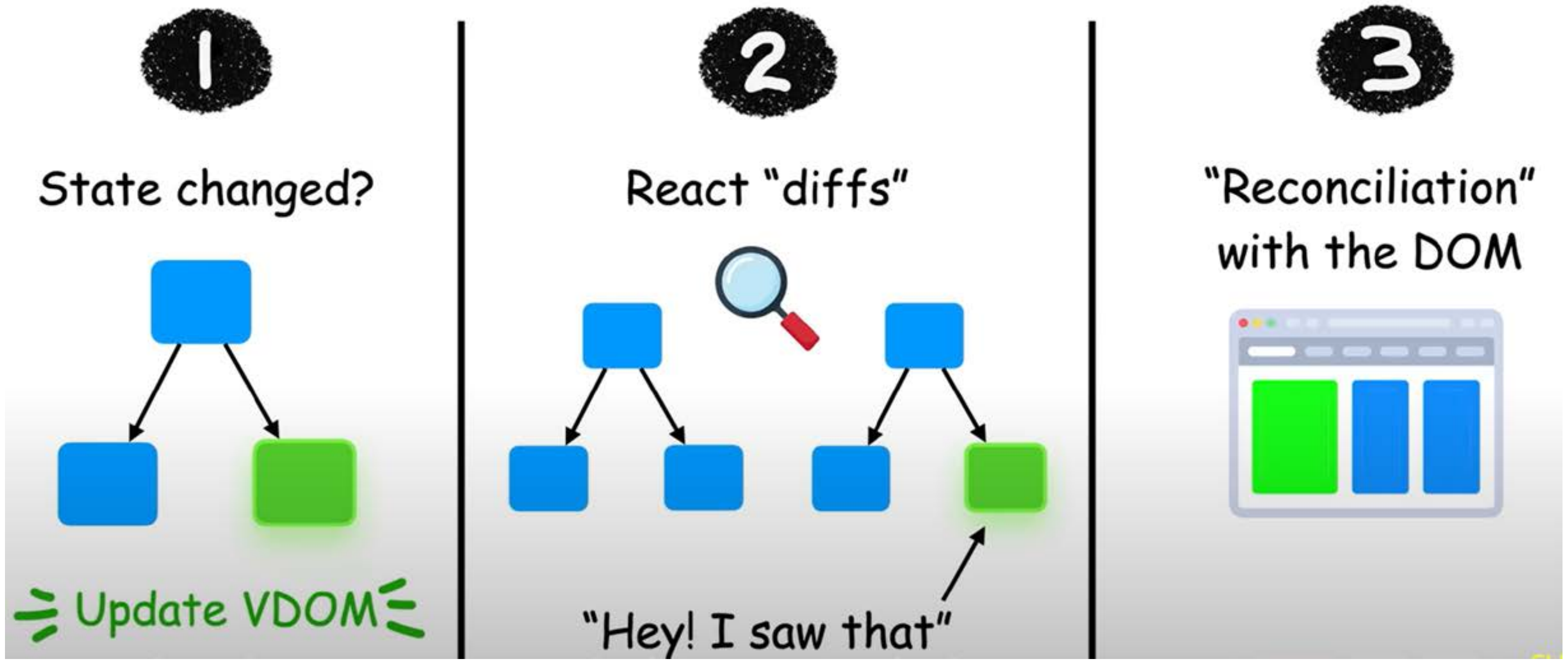
ReactJS Basics

1. Components are the building blocks.
   - Components make UI modular and manageable.
   - They are reusable, self-contained pieces of UI.
2. Virtual DOM improves performance.
   - React updates the Virtual DOM (lightweight copy of the actual DOM) first and then syncs changes to the real DOM efficiently.
   - This minimizes unnecessary re-rendering and improves performance.
3. One-way data binding ensures better control.
   - Data flows in a single direction, making debugging easier.
   - State updates trigger UI re-renders automatically.
4. JSX is used to write HTML in JavaScript.

VDOM - How does React Work?

- React creates a **VIRTUAL DOM** in memory.
- To use React in production, you need npm which is included with Node.js.

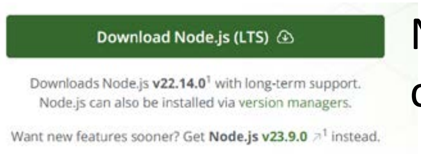
VDOM: React Rendering



- ✓ **Virtual DOM optimizes performance.** State Changed? (Update VDOM)
- ✓ **React updates only changed parts:** React **compares** (or "diffs") the new Virtual DOM with the previous one.
- ✓ **Efficient rendering improves app speed:** React applies only the **necessary updates** to the actual DOM. This is called **reconciliation**

Setting up ReactJS Environment

1. Install Node.js and npm.



Downloads Node.js **v22.14.0**[1] with long-term support.
Node.js can also be installed via version managers.

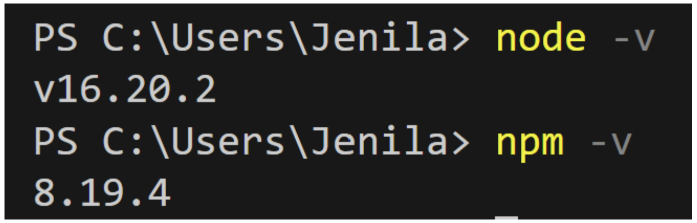
Want new features sooner? Get **Node.js v23.9.0**[1] instead.

Note: Do not enable chocolatey for fast installation

   – https://nodejs.org/en
   – After installation, check if Node.js and npm are installed.
   – Type the following **VS Code terminal (Press Ctrl + ~):**

     node -v

     npm –v



```
PS C:\Users\Jenila> node -v
v16.20.2
PS C:\Users\Jenila> npm -v
8.19.4
```

   – This should return the installed versions of Node.js and npm.
   – If npm is not installed, install it with npm init/ npm install
   – If npx is not installed, install it with npm install -g npx

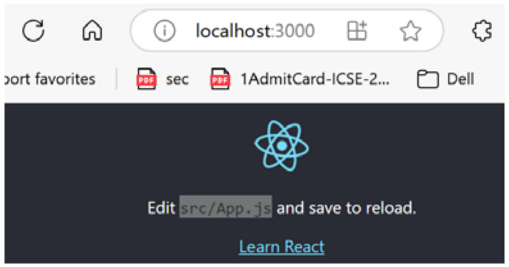
2. Install the Required Extensions in VS Code.
   – VS Code -> go to **Extensions (Ctrl+Shift+X)**
   – Search for and install:
     - **ES7+ React/Redux/React-Native snippets** (for faster coding)
     - **Prettier -** Code formatter (for formatting)
     - ~~Simple React Snippets (for additional shortcuts)~~
     - **React Developer Tools (for debugging React components)**

Setting up ReactJS Environment

3. Run the following command to create a new React app:
   - **npx create-react-app my-app**

   (Installing react, react-dom, and react-scripts etc.)
4. Open the Project in VS Code (Navigate to the project)
   - **cd my-app**
5. Start the React Application
   - Start the development server: **npm start**
   - This will start a development server, and you can now view my-app in the default browser.

     Local: **http://localhost:3000**
     On Your Network: http://172.16.105.1:3000

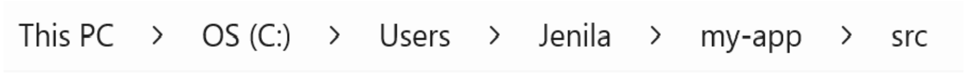


Edit src/App.js and save to reload.

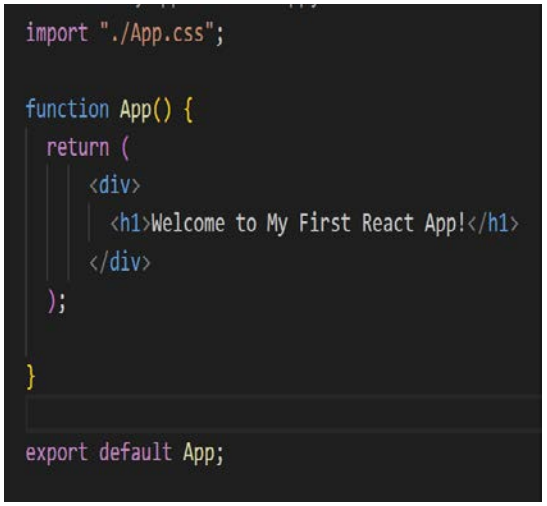
Learn React

Setting up ReactJS Environment

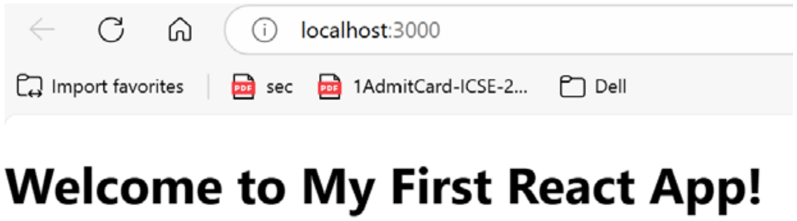
6. Modify the APP and Run the React Code (Edit App.js)
   - Open the file **C:\Users\Jenila\my-app\src\App.js** in VS Code
   - Modify the App.js file and change the <h1> tag inside the return statement:


This PC > OS (C:) > Users > Jenila > my-app > src

```
function App() {
  return (
     <div>
     <h1>Welcome to My First React
App!</h1>
     </div>
  );
}

export default App;
```

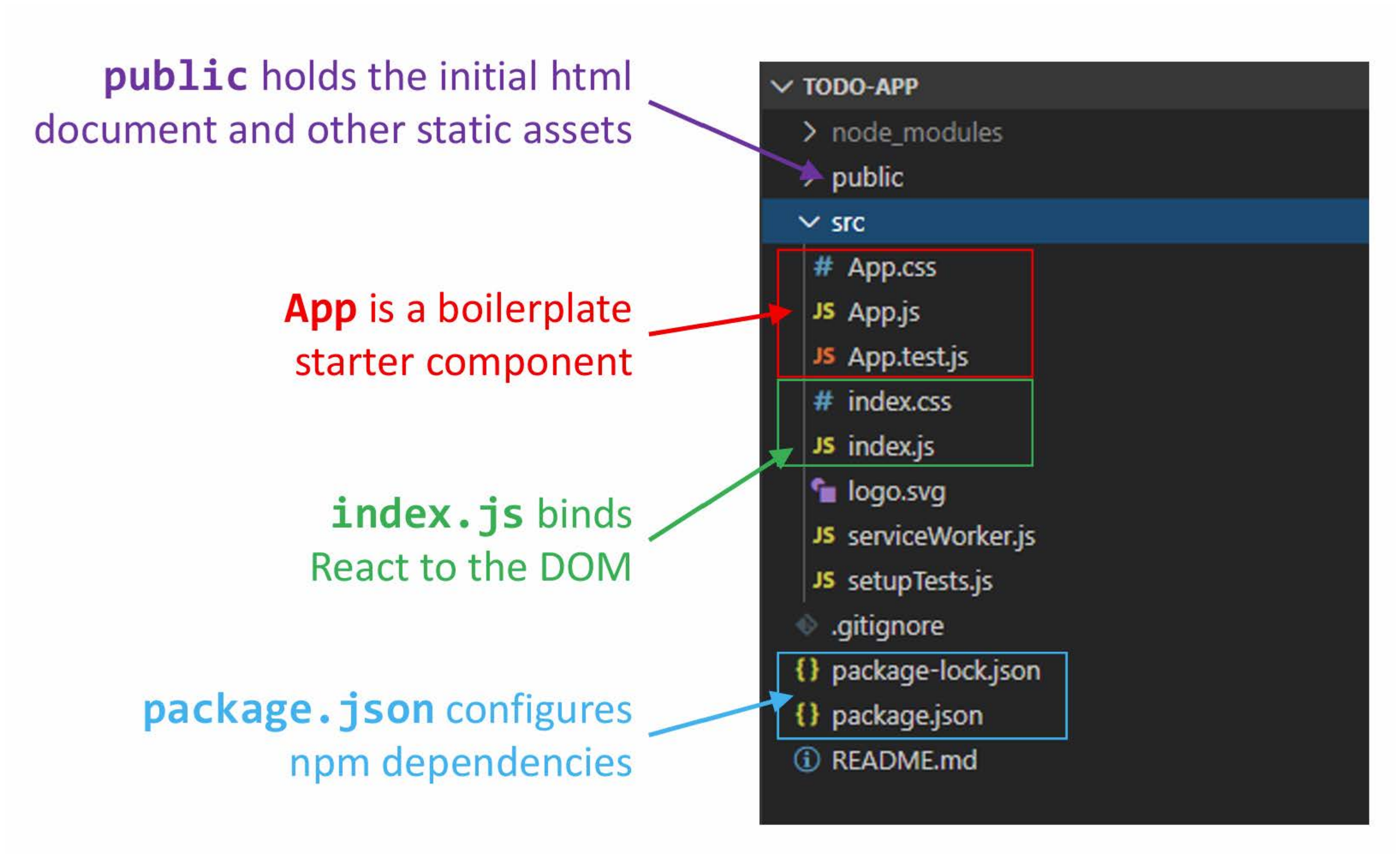


```
import "./App.css";

function App() {
  return (
    <div>
      <h1>Welcome to My First React App!</h1>
    </div>
  );

}

export default App;
```



Welcome to My First React App!

Folder Structure of React App



`public` holds the initial html document and other static assets

App is a boilerplate starter component

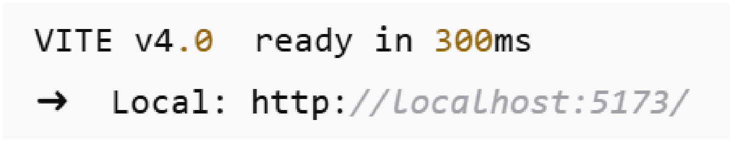
`index.js` binds React to the DOM

`package.json` configures npm dependencies

;.

Vite + React App in VS Code

1. **Open the Terminal in VS Code**
   – **Press Ctrl + ~ (tilde key) to open the terminal**
2. **Create a Vite React App**
   – **npx create-vite@latest my-vite-app --template react**
      • **my-vite-app** is the name of the project folder
   – **cd my-vite-app**
   – **npm install**
   – **npm run dev**
3. Create a Simple Counter Component (Counter.jsx)
4. Render the Counter in App.jsx (Modify src/App.jsx)
5. After running npm run dev, it will show a local development server URL like

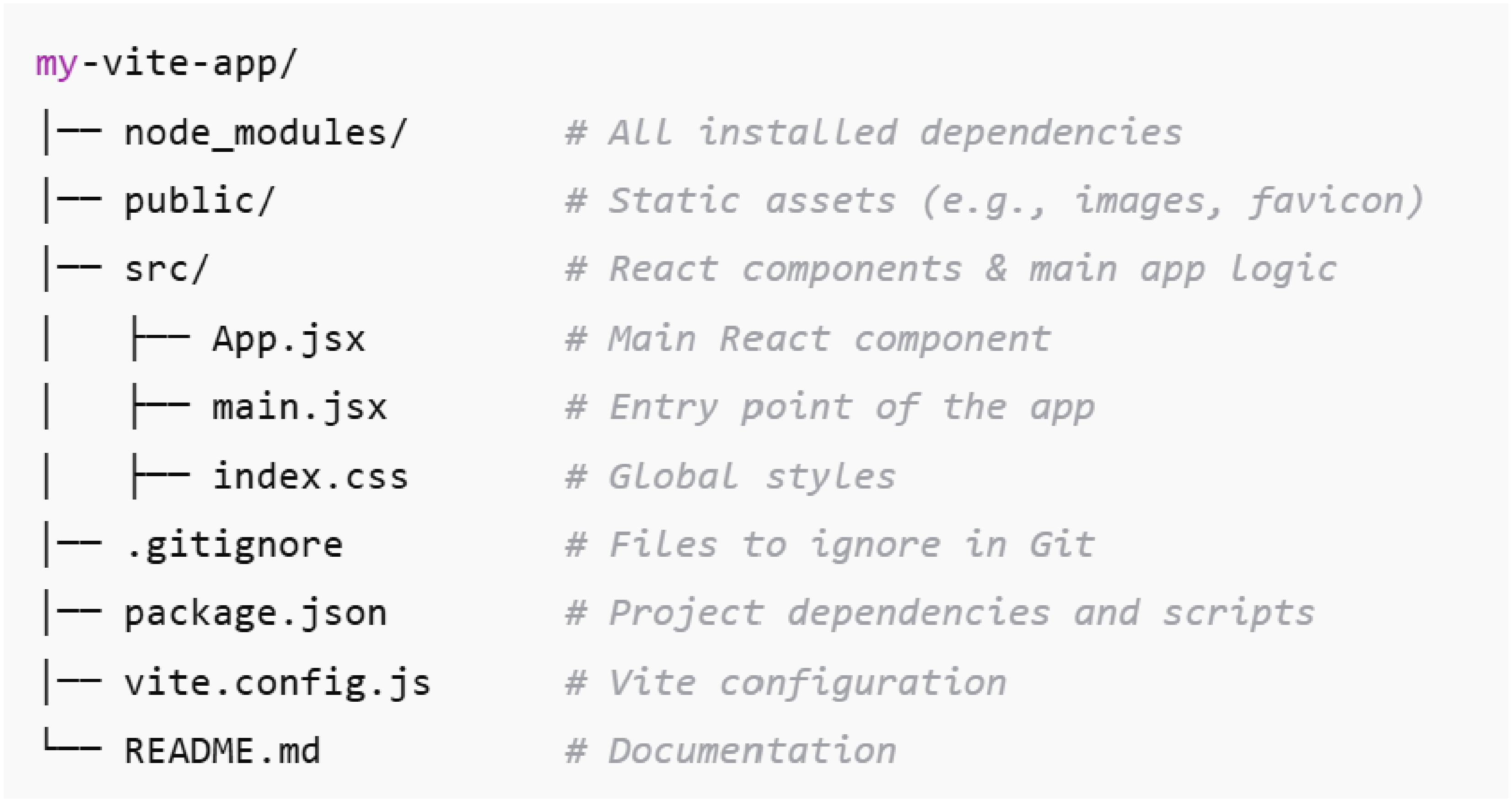


```
VITE v4.0  ready in 300ms

➜  Local: http://localhost:5173/
```

   – Open **http://localhost:5173/** in your browser to see your React app running!

Folder Structure

Vite + React App

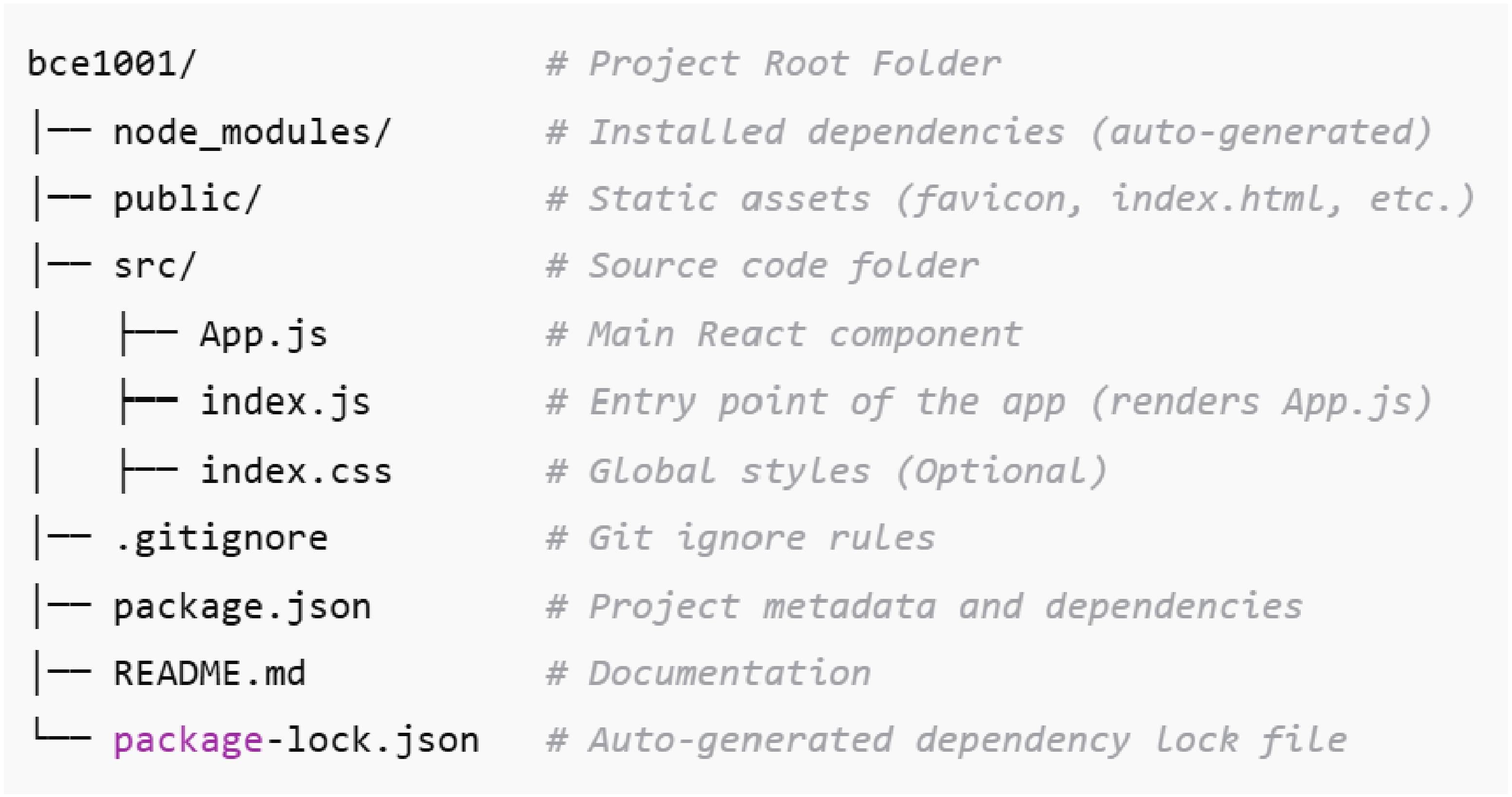


```
my-vite-app/
|── node_modules/      # All installed dependencies
|── public/            # Static assets (e.g., images, favicon)
|── src/               # React components & main app logic
|    ├── App.jsx       # Main React component
|    ├── main.jsx      # Entry point of the app
|    ├── index.css     # Global styles
|── .gitignore         # Files to ignore in Git
|── package.json       # Project dependencies and scripts
|── vite.config.js     # Vite configuration
└── README.md          # Documentation
```

JSX in React (VS Code)

1. **Open the Terminal in VS Code**
   – Press Ctrl + ~ (tilde key) to open the terminal c:\users\student\
2. **Create a React App**
   – npx **create-react-app** bce1001
   – cd bce1001
   – npm start
3. **Write JSX in Your React Component (VS code editor)**
   – Inside src/App.js, modify the file
   – (or) Create a new file, type your coding, save it with .js or .jsx extension
4. **Render the Component in index.js**
5. **Run the project**
   – npm start **(Access at http://localhost:3000)**

Folder Structure



```
bce1001/                  # Project Root Folder
|── node_modules/         # Installed dependencies (auto-generated)
|── public/               # Static assets (favicon, index.html, etc.)
|── src/                  # Source code folder
|   ├── App.js            # Main React component
|   ├── index.js          # Entry point of the app (renders App.js)
|   ├── index.css         # Global styles (Optional)
|── .gitignore            # Git ignore rules
|── package.json          # Project metadata and dependencies
|── README.md             # Documentation
└── package-lock.json     # Auto-generated dependency lock file
```

JSX: Writing HTML in JavaScript

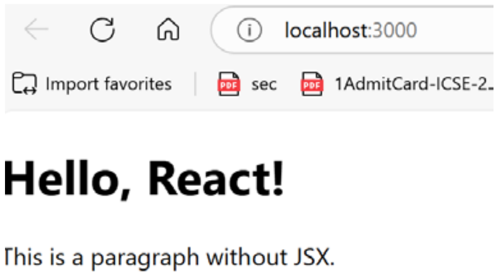
- **J**ava**S**cript **X**ML
- JSX can be used to create React components
- JSX allows mixing HTML (XML) with JavaScript.
- JSX makes React code more readable and easier to write.

  ```
  const element = <h1>Hello, World!</h1>;
  ```

- JSX follows XML rules
  - All elements must be placed inside one parent element
  - All elements must be closed

React component without JSX

```
import React from "react";

function App() {
  return React.createElement(
    "div",
    null,
    React.createElement("h1", null, "Hello, React!"),
    React.createElement("p", null, "This is a paragraph without JSX.")
  );
}

export default App;
```



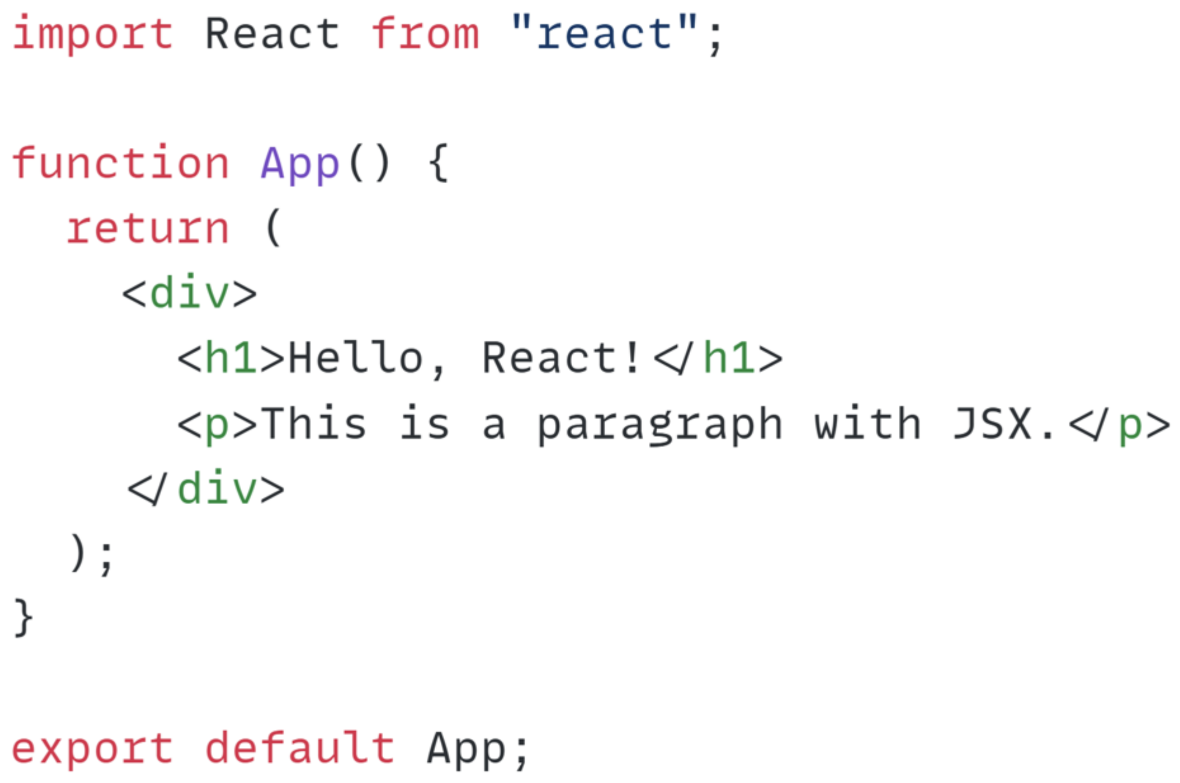
Hello, React!

This is a paragraph without JSX.

React.**createElement** method **manually creates elements** for the **React Virtual DOM**.

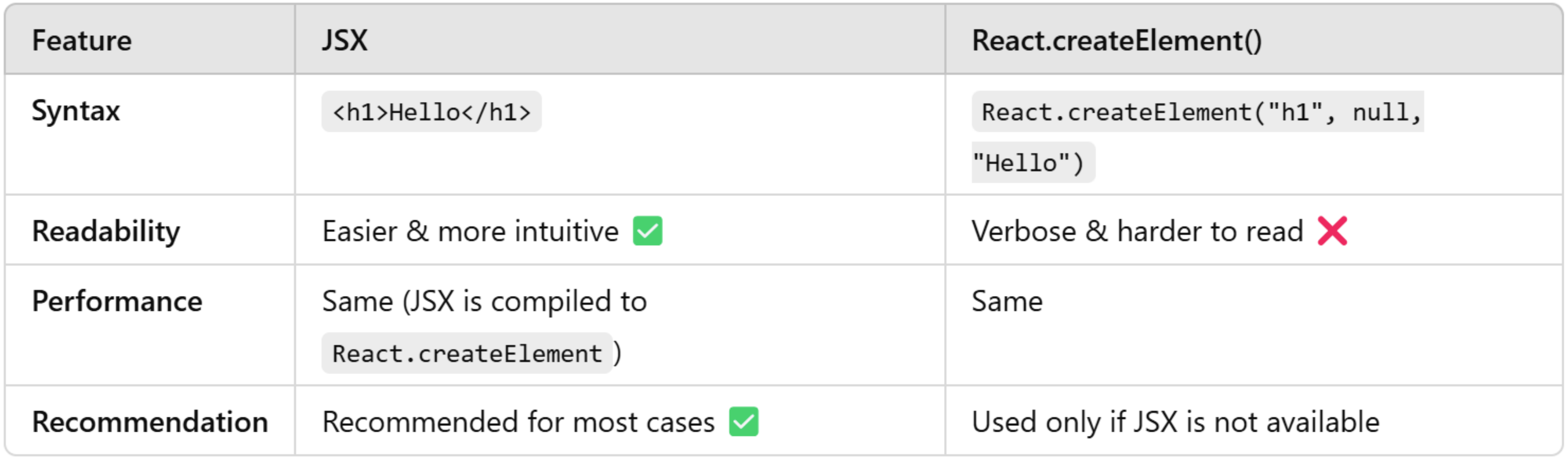
React JSX - Example

- JSX allows you to write HTML-like syntax inside JavaScript.
- Instead of using React.createElement(), JSX lets you write <div>, <h1>, and <p> directly.



```
import React from "react";

function App() {
  return (
    <div>
      <h1>Hello, React!</h1>
      <p>This is a paragraph with JSX.</p>
    </div>
  );
}

export default App;
```

JSX Vs React.createElement()



Feature	JSX	React.createElement()
Syntax	`<h1>Hello</h1>`	`React.createElement("h1", null, "Hello")`
Readability	Easier & more intuitive ✅	Verbose & harder to read ❌
Performance	Same (JSX is compiled to `React.createElement`)	Same
Recommendation	Recommended for most cases ✅	Used only if JSX is not available

React JSX - Steps

Basic Steps for coding React JSX:

1. **Import React**: React must be imported to use JSX. (Not requires for React 17 and above)

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

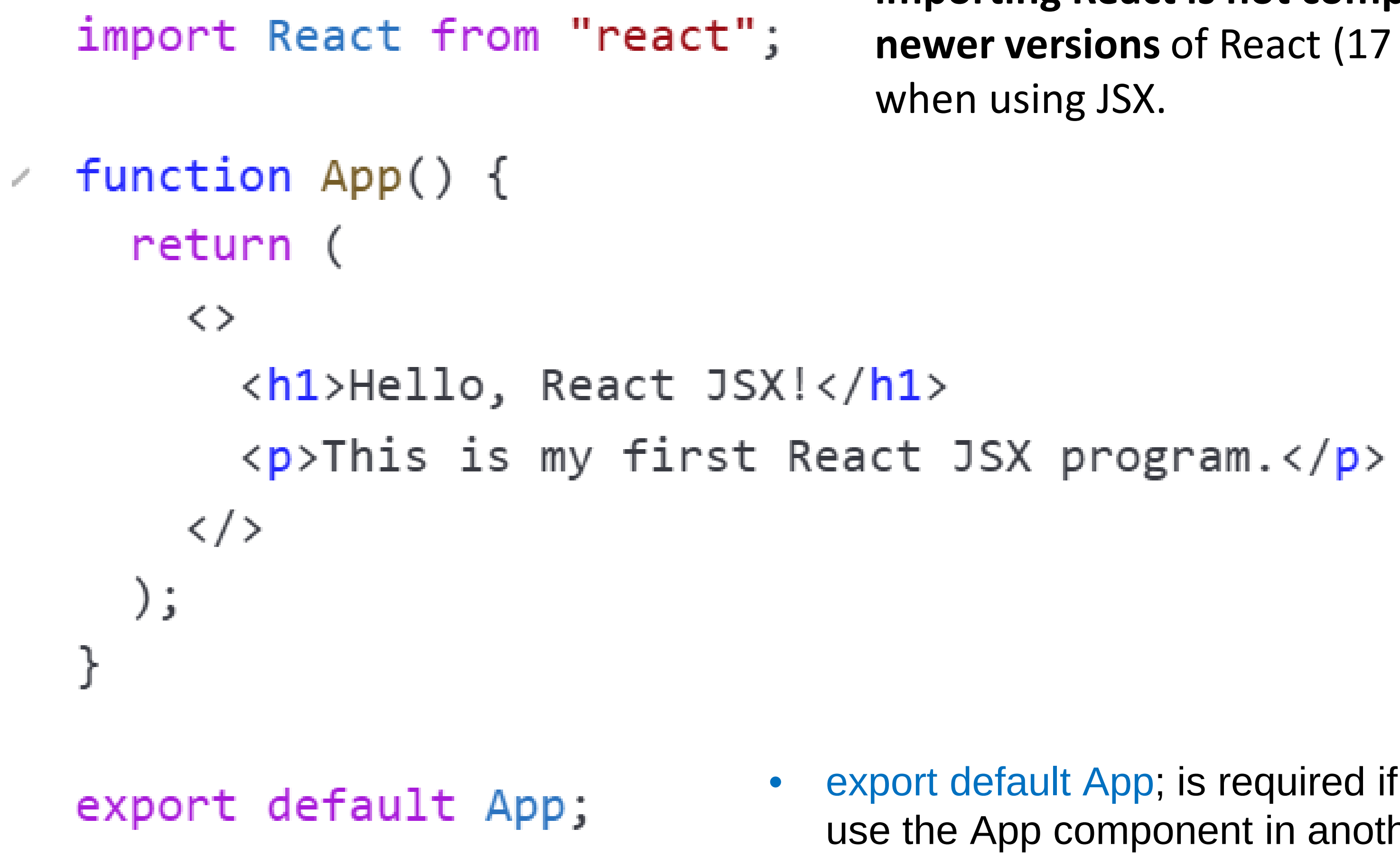
2. **Functional Component**: Write a **function** that returns JSX inside the **return** statement.

```
function App() {
        return (
        ….
        );
}
```

3. **Export JSX**.

```
export default App;
```

First JSX program



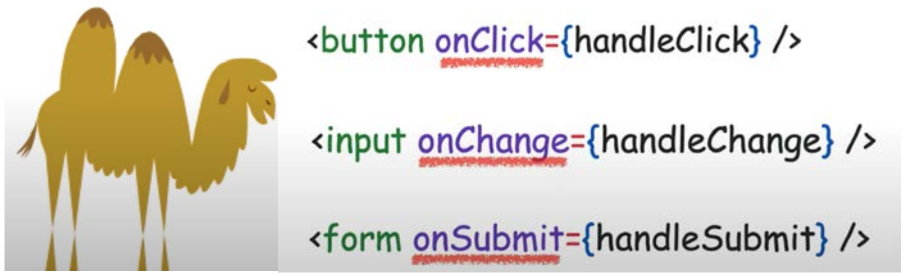
```
import React from "react";

function App() {
  return (
    <>
      <h1>Hello, React JSX!</h1>
      <p>This is my first React JSX program.</p>
    </>
  );
}

export default App;
```

importing React is not compulsory in newer versions of React (17 and above) when using JSX.

- export default App; is required if you want to use the App component in another file

JSX!=HTML
Main Differences

- Dynamic



```
<button onClick={handleClick} />

<input onChange={handleChange} />

<form onSubmit={handleSubmit} />
```

- Written in **camelCase**
  - Use camelCase for CSS properties inside style={{}}.
  - Use camelCase like onClick={handleClick}.

```
const handleClick = () => alert("Button clicked!");
return <button onClick={handleClick}>Click Me</button>;
```

- JSX uses className instead of class to avoid conflicts with JS.

```
return <div className="container">Hello</div>;
```

- JSX uses htmlFor instead of for in labels

```
return <label htmlFor="name">Enter Name</label>;
```

JSX!=HTML

- Use {/* comment */} instead of <!-- HTML comment -->.

```
{/* This is a comment */}
```

- **Use curly braces { } for expression**

```
const name = "Alice";
return <h1>Hello, {name}!</h1>;
```

{ ... }

- Self-Closing Tags Must End With /
  - Example: <img src='image.jpg' alt='React Logo’ />
- Use ? : or && for conditions inside JSX.

```
<p>The number {number} is {number % 2 === 0 ? "Even" : "Odd"}</p>
```

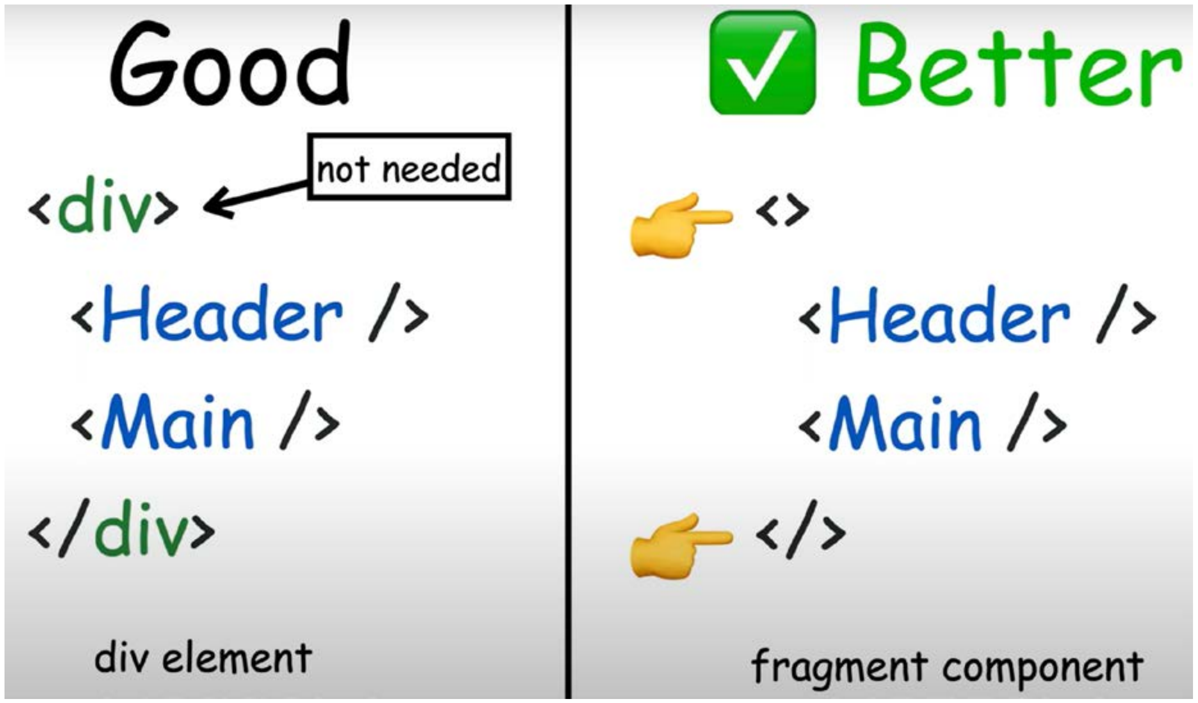
- Functions should be inside {} like {greet()}.

```
const greet = () => "Hello, React!";
return <h1>{greet()}</h1>;
```

- Use <input checked /> instead of checked='true’.
- Lists Must Have a key
  - Example: <li key={index}>{item}</li>.

Root

- Must Return a Single Parent Element
- Wrap elements in a parent tag like <div> or <>...</>.
- Use <>...</> instead of unnecessary <div>.



```
<div>
  <Header />
  <Main />
</div>
```

```
<>
  <Header />
  <Main />
</>
```

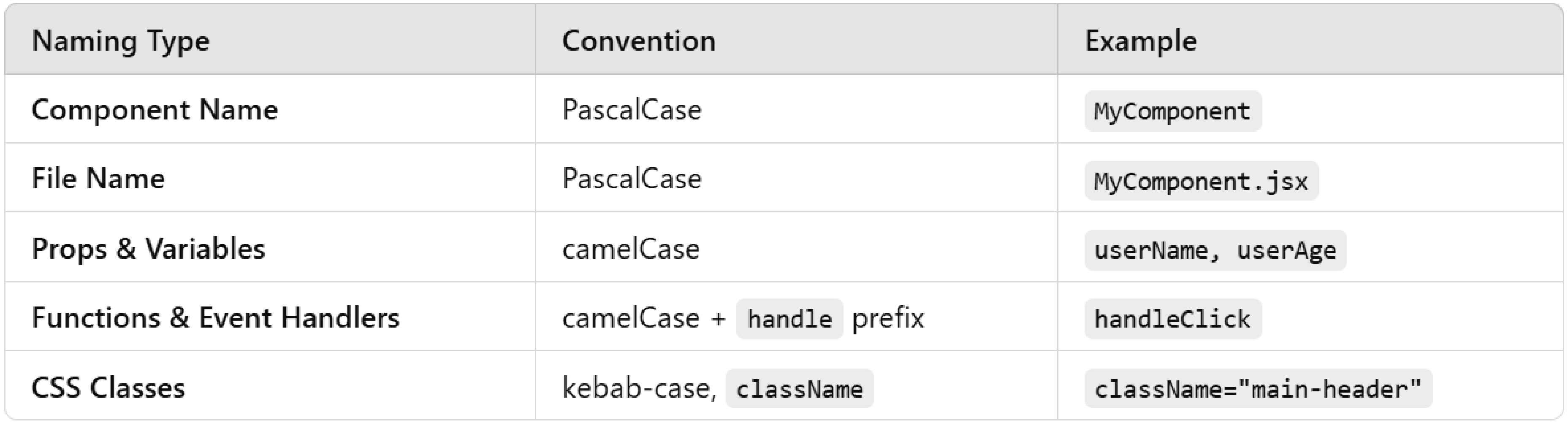
JSX Rules Summary

Rule	Description
1. Must Return a Single Parent Element	Wrap elements in a parent tag like <div> or <>...</>.
2. Use className Instead of class	JSX uses className instead of class to avoid conflicts with JS.
3. Use htmlFor Instead of for	JSX uses htmlFor instead of for in labels.
4. Self-Closing Tags Must End With /	Example: <img src='image.jpg' alt='React Logo' />
5. Expressions Must Be Wrapped in {}	Example: <h1>Hello, {name}!</h1>
6. Conditional Rendering	Use ? : or && for conditions inside JSX.
7. Inline Styles Use Objects	Use camelCase for CSS properties inside style={{}}.
8. Comments in JSX	Use {/* comment */} instead of <!-- HTML comment -->.
9. JavaScript Functions in JSX	Functions should be inside {} like {greet()}.
10. Lists Must Have a key	Example: <li key={index}>{item}</li>.
11. Fragments for Grouping	Use <>...</> instead of unnecessary <div>.
12. Event Handling	Use camelCase like onClick={handleClick}.
13. Boolean Attributes	Use <input checked /> instead of checked='true'.

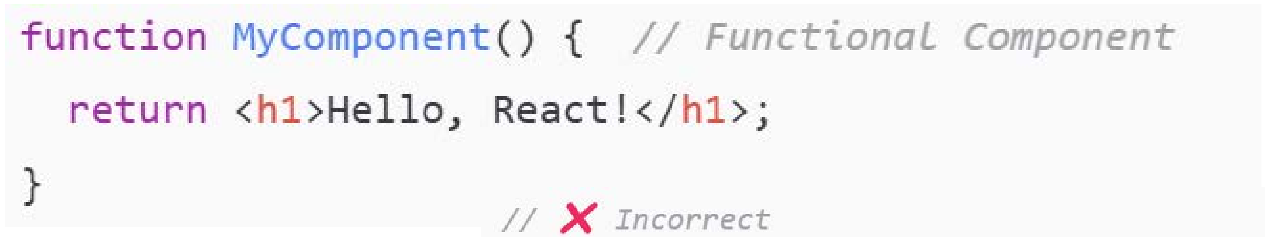
HTML Vs JSX

Feature	**HTML**	**JSX**
Type	Markup Language	JavaScript Syntax Extension
Use Case	Static web pages	Dynamic React components
Rendering	Directly by the browser	Compiled to JavaScript and then rendered by React
Syntax	Standard HTML tags	HTML-like syntax within JavaScript
Interactivity	JavaScript for dynamic behavior	Native support for JavaScript expressions in the markup
Reactivity	Static (unless using JavaScript)	Dynamic (re-renders on state/prop changes)
Expressions	Not supported	Supported inside curly braces `{}`
Attributes	`class`, `for`	`className`, `htmlFor` (for React compatibility)

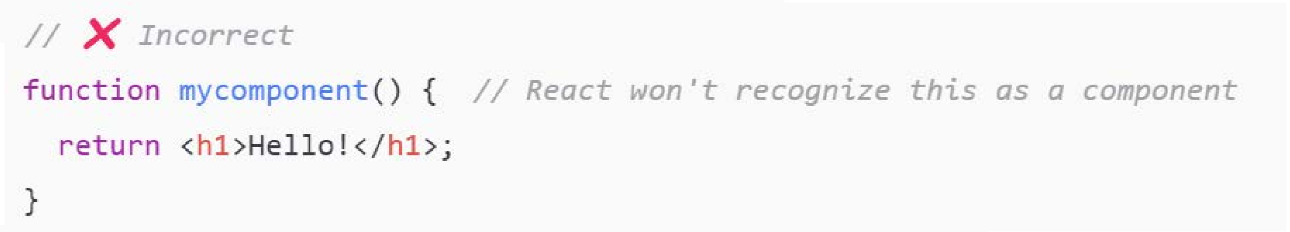
Naming Convention



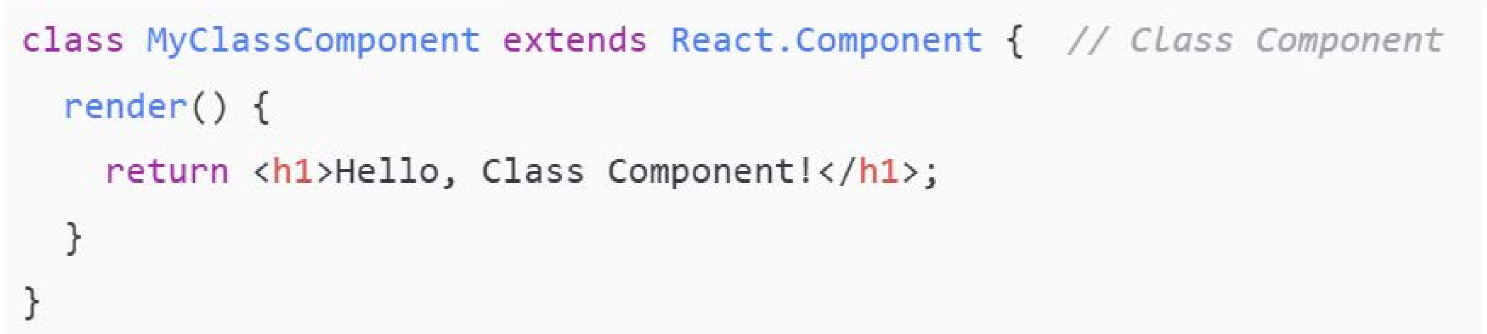
Naming Type	Convention	Example
Component Name	PascalCase	`MyComponent`
File Name	PascalCase	`MyComponent.jsx`
Props & Variables	camelCase	`userName, userAge`
Functions & Event Handlers	camelCase + `handle` prefix	`handleClick`
CSS Classes	kebab-case, `className`	`className="main-header"`



```
function MyComponent() {  // Functional Component
  return <h1>Hello, React!</h1>;
}
```



```
// ❌ Incorrect
function mycomponent() {  // React won't recognize this as a component
  return <h1>Hello!</h1>;
}
```



```
class MyClassComponent extends React.Component {  // Class Component
  render() {
    return <h1>Hello, Class Component!</h1>;
  }
}
```

Rendering React Components

1. **Rendering Basics**
   - **App.js with index.js**
   - **Hello.js with index.js**
2. **Rendering when file name and Function/class name are different**
3. **Rendering multi[le files**
4. **Rendering multiple components**

Rendering React Components

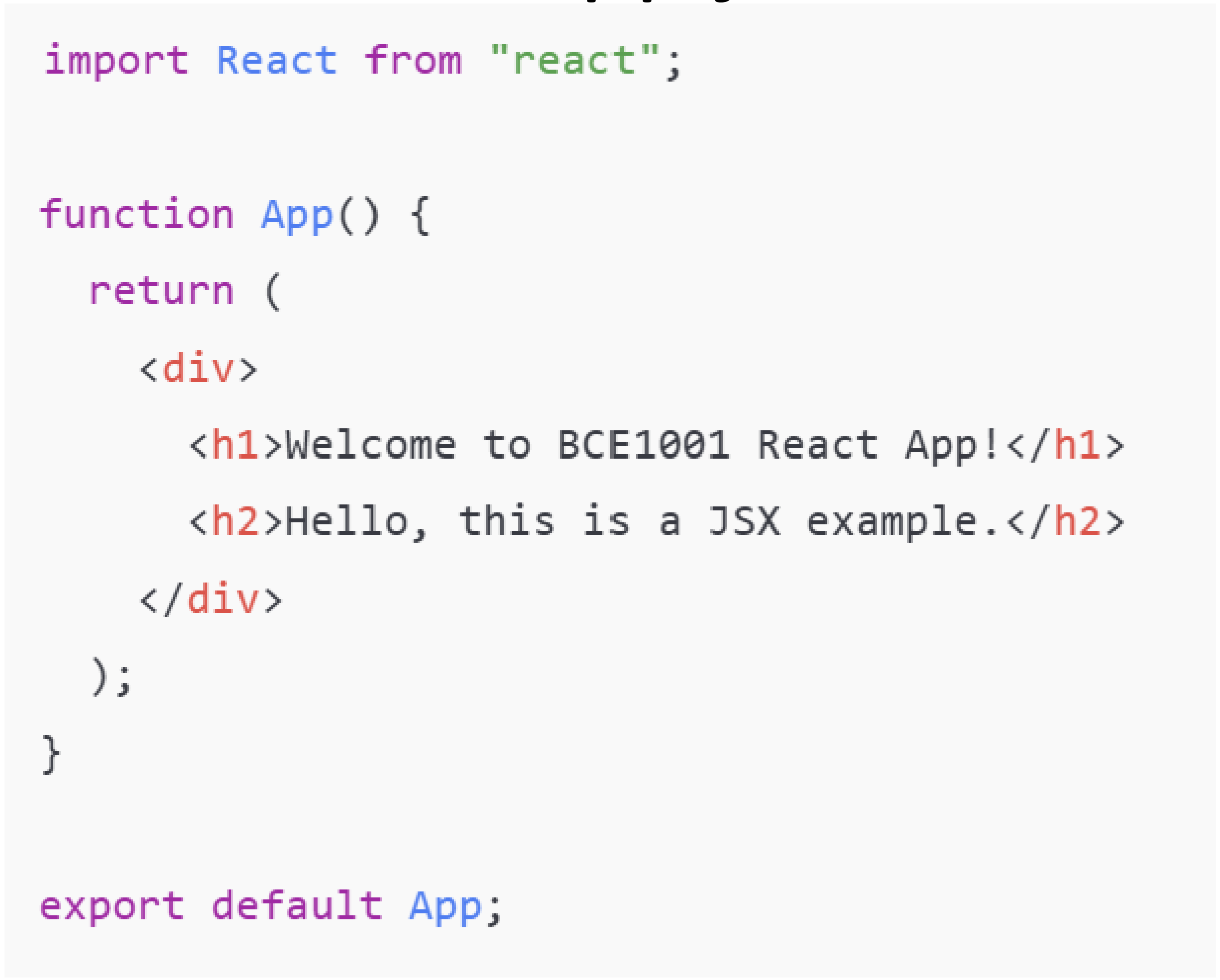
index.js

- import React from 'react’;
- import ReactDOM from 'react-dom’;
- **import App from './App’;**
- **ReactDOM.render**(**<App />**, document.getElementById('root’));

(or)

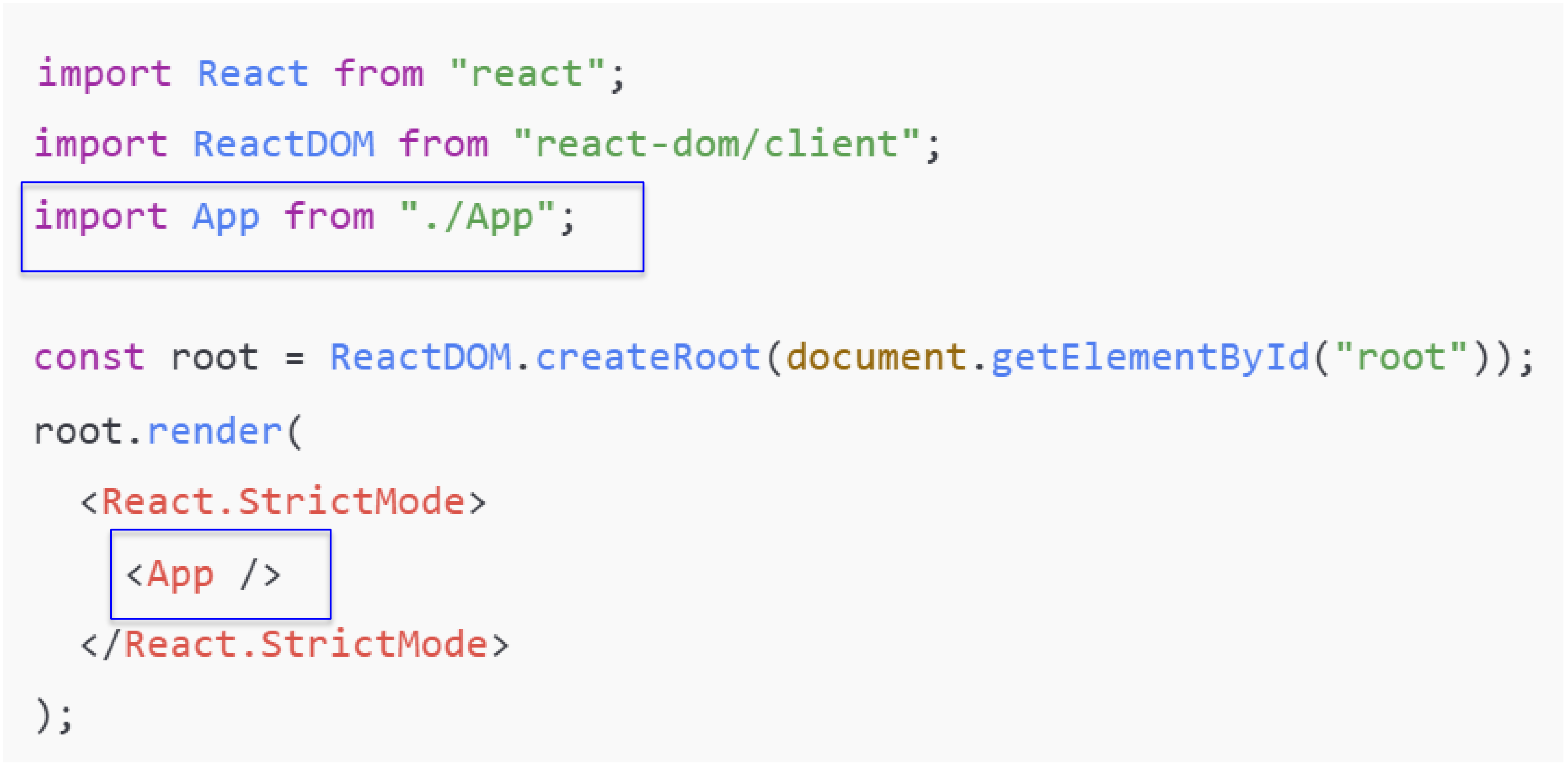
```
const root = ReactDOM.createRoot(document.getElementById('root’))
root.render(<App />);
```

Rendering: Example 1
App.js



```
import React from "react";

function App() {
  return (
    <div>
      <h1>Welcome to BCE1001 React App!</h1>
      <h2>Hello, this is a JSX example.</h2>
    </div>
  );
}

export default App;
```

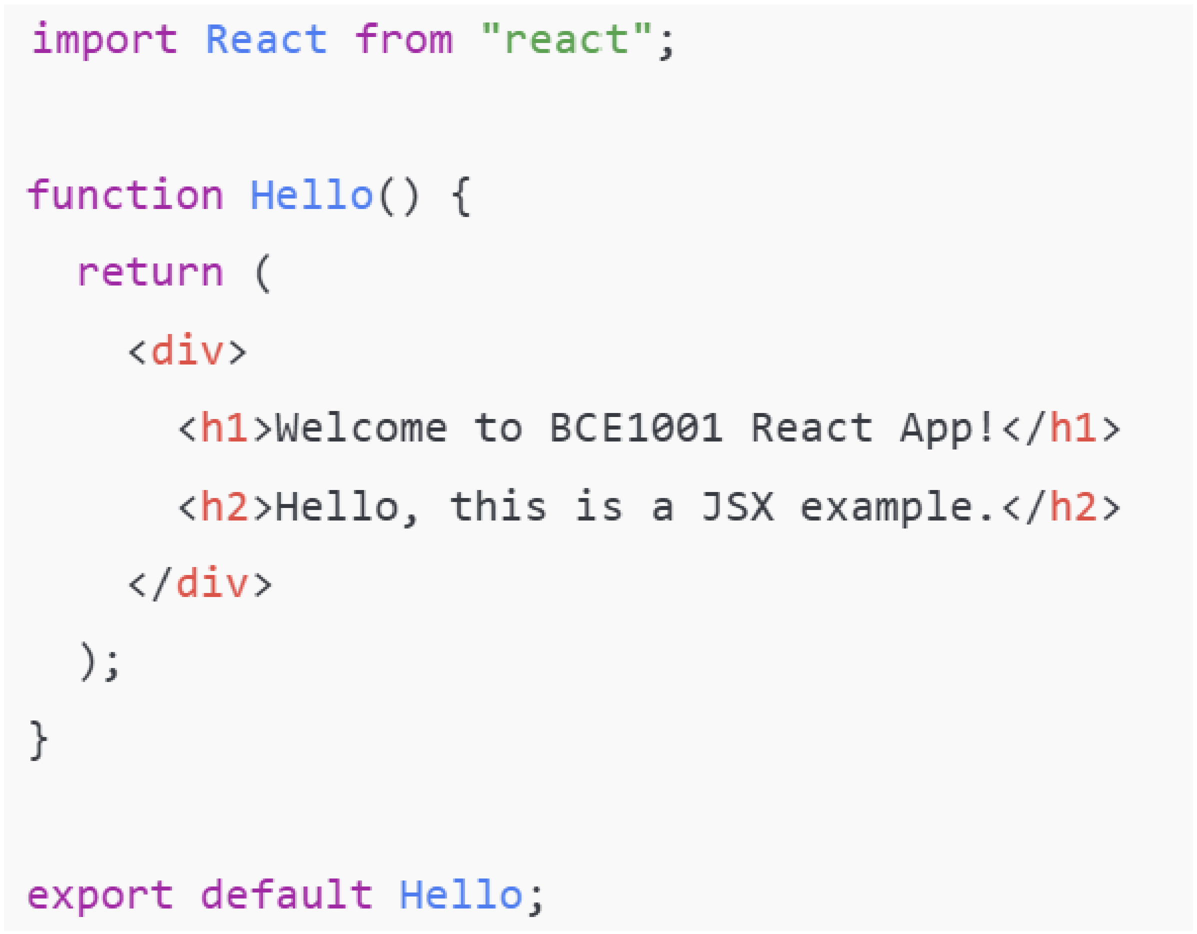
Rendering App.js in src/Index.js



```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

http://localhost:3000)

Rendering – Example 2
Hello.js



```
import React from "react";

function Hello() {
  return (
    <div>
      <h1>Welcome to BCE1001 React App!</h1>
      <h2>Hello, this is a JSX example.</h2>
    </div>
  );
}

export default Hello;
```

Rendering Hello.js in src/Index.js

```
import React from "react";
import ReactDOM from "react-dom/client";
import Hello from "./Hello"; // Importing Hello.js instead of App.js

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <React.StrictMode>
    <Hello /> {/* Rendering Hello instead of App */}
  </React.StrictMode>
);
```

Rendering – Example 3 Welcome.js

```
//Getting values from index.js


import React from "react";
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
export default Welcome;
```

Rendering welcome.js in index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
//import App from './App';
import Welcome from './Welcome';
import reportWebVitals from './reportWebVitals';

    const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Welcome name="Jenila" />
</React.StrictMode>
);
reportWebVitals();
```

Different File name & Function/Class Name

Welcome.js

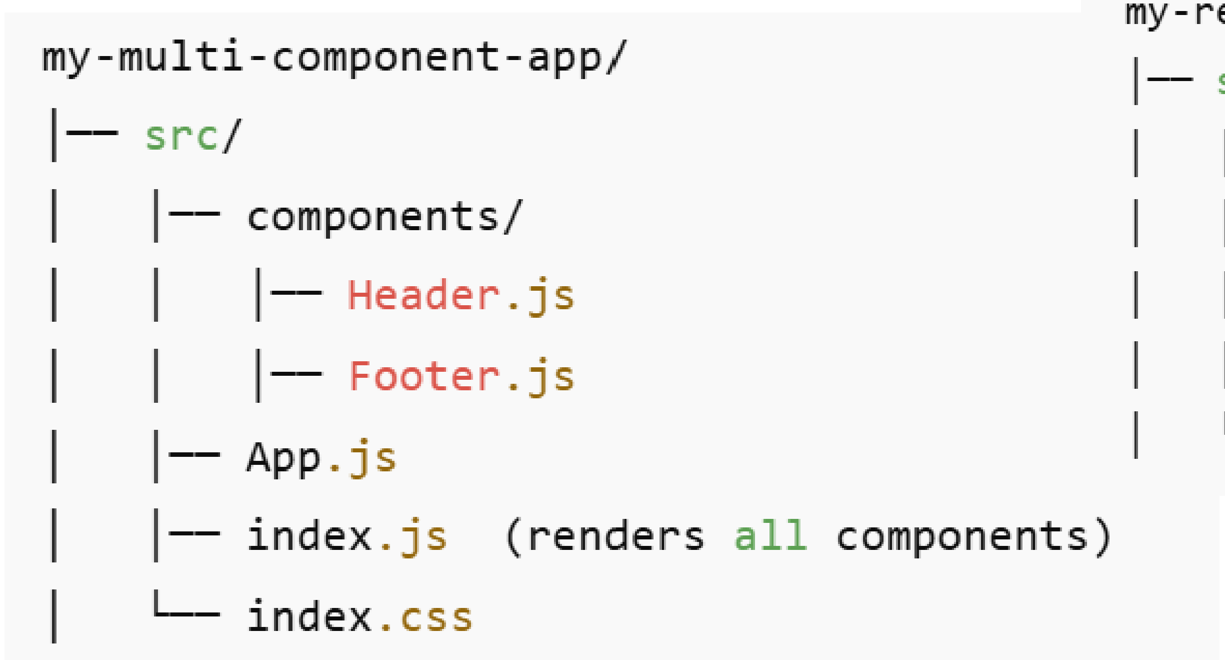
```
import React from "react";
class Wel extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
export default Wel;
```

index.js

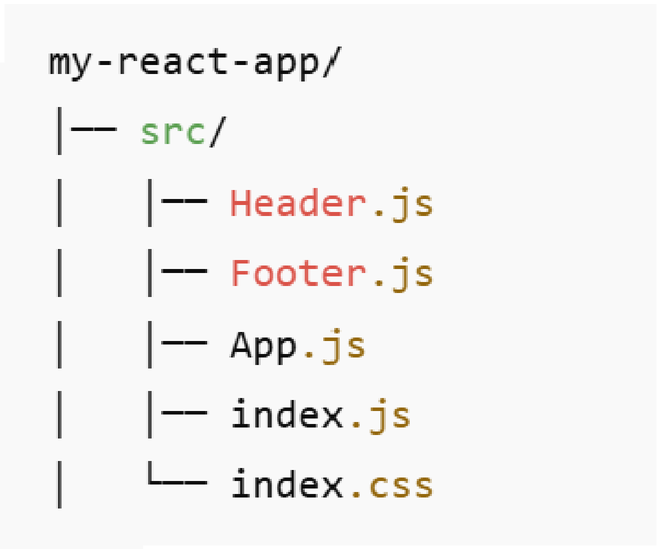
```
import Wel from './Welcome’;
…
…
 <Wel name="Jenila" />
```

Rendering Multiple Files

- Folder Structure:
  - with and without Components Folder

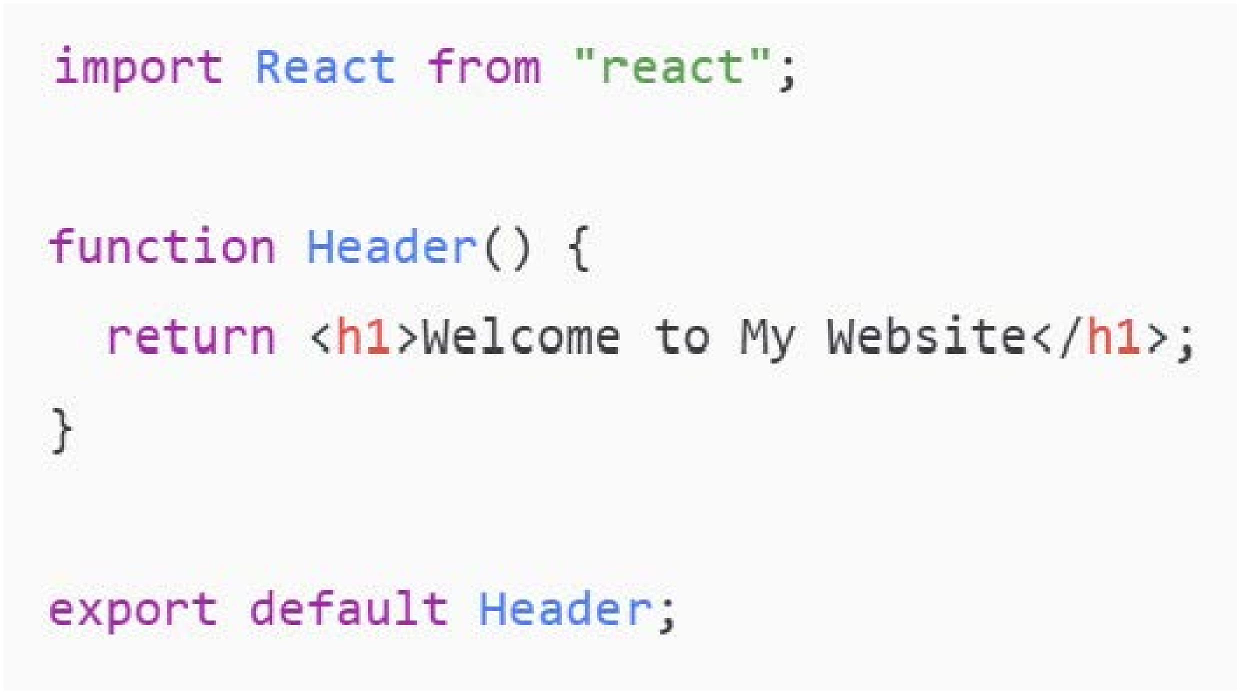


```
my-multi-component-app/
|── src/
|   |── components/
|   |   |── Header.js
|   |   |── Footer.js
|   |── App.js
|   |── index.js  (renders all components)
|   └── index.css
```



```
my-react-app/
|── src/
|   |── Header.js
|   |── Footer.js
|   |── App.js
|   |── index.js
|   └── index.css
```

Header.js (Component 1)



```
import React from "react";

function Header() {
  return <h1>Welcome to My Website</h1>;
}

export default Header;
```

Footer.js (Component 2)

```
import React from "react";

function Footer() {
  return <p>© 2025 My Website. All rights reserved.</p>;
}

export default Footer;
```

Parent Component – App.js (Wrapping header and Footer)

```
import React from "react";
import Header from "./components/Header";
import Footer from "./components/Footer";

function App() {
  return (
    <div>
      <Header />
      <p>This is the main content of the website.</p>
      <Footer />
    </div>
  );
}

export default App;
```

Rendering parent in Index.js

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
import "./index.css"; // Optional styling

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Multiple Components – App.js

```
import React from "react";

function App() {
  return (
    <div style={{ textAlign: "center", fontFamily: "Arial, sans-serif" }}>
      <Header />
      <Content />
      <Footer />
    </div>
  );
}

// Header Component
function Header() {
  return (
    <header style={{ background: "black", color: "white", padding: "10px" }}>
      <h1>My Simple React App</h1>
    </header>
  );
}
```

```
// Content Component
function Content() {
  return (
    <div style={{ margin: "20px" }}>
      <p>This is a simple React app with multiple components.</p>
    </div>
  );
}

// Footer Component
function Footer() {
  return (
    <footer style={{ background: "black", color: "white", padding: "10px" }}>
      <p>© 2025 SimpleApp. All rights reserved.</p>
    </footer>
  );
}

export default App;
```

Render App.js in index.js

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";

ReactDOM.render(<App />, document.getElementById("root"));
```

React apps **automatically use** the <div id="root"></div> that is already present in public/**index.html**.

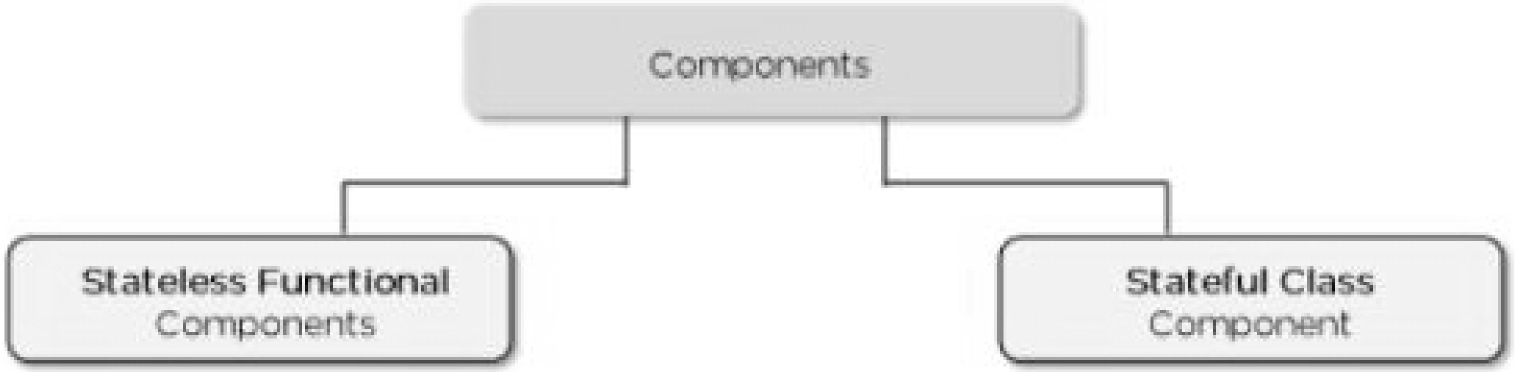
public/index.html already has

```
<body>
    <div id="root"></div>  <!-- React will inject App.js here -->
</body>
```

React Components

- Components are the building blocks of a React application
- Components can be nested, reused, and managed independently.
- **Functional Components:** Simple, stateless, written as **functions**.
- **Class Components:** More powerful, stateful and lifecycle methods, written as **ES6 classes**.

React Components



Functional Components

Stateless, written as **function**

Many React projects start with one core component called **App**

```
function App() {
    return (
        <article>
            <h1>Recipe Manager</h1>
        </article>
    );
}

export default App;
```

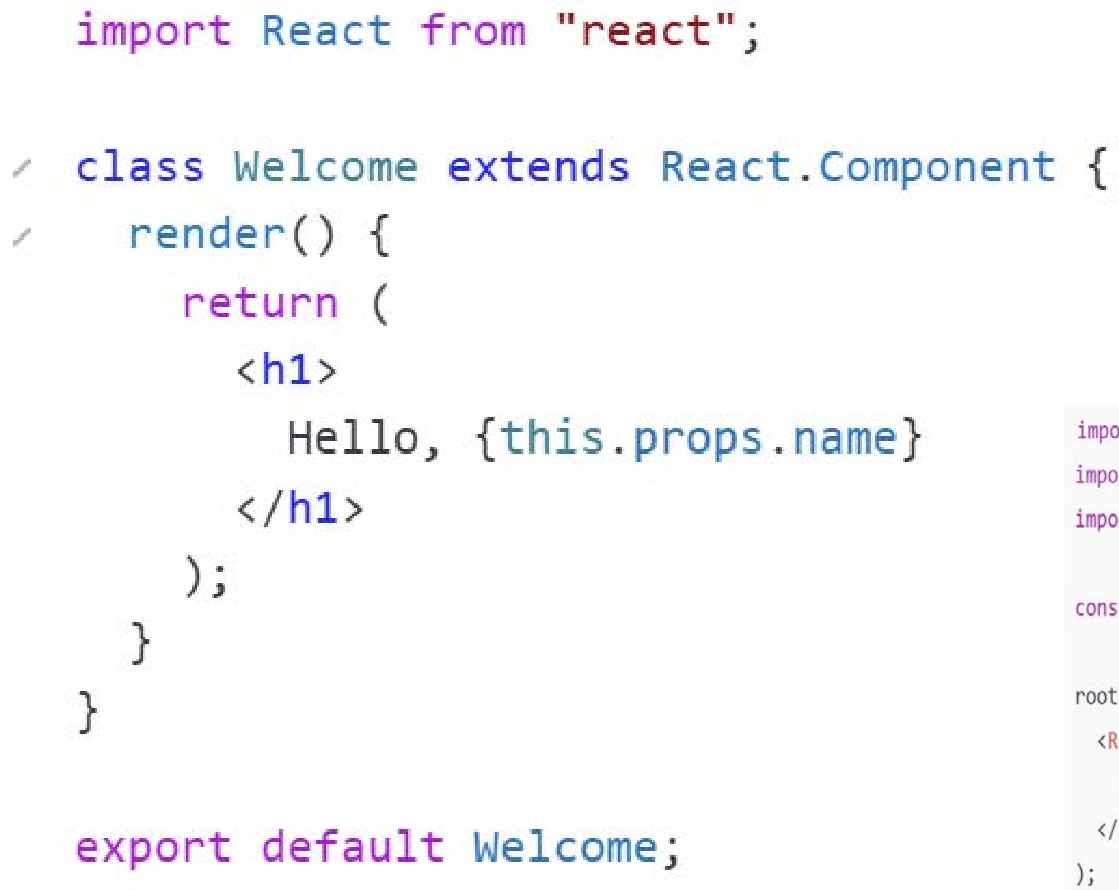
App component using an arrow function

```
const App = () => {
    return (
        <article>
            <h1>Recipe Manager</h1>
        </article>
    );
};

export default App;
```

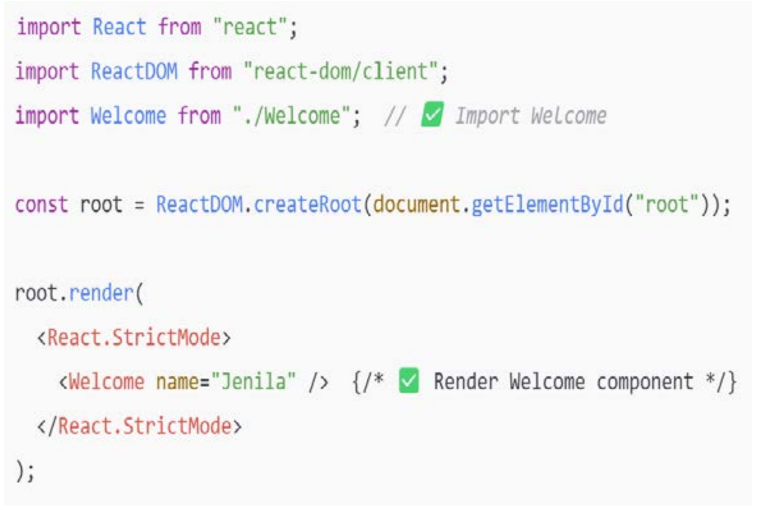
If a function has a **block body** (i.e., {}), it must use an **explicit return** statement to return a value.

Class Components

- Stateful, written as ES6 classes.



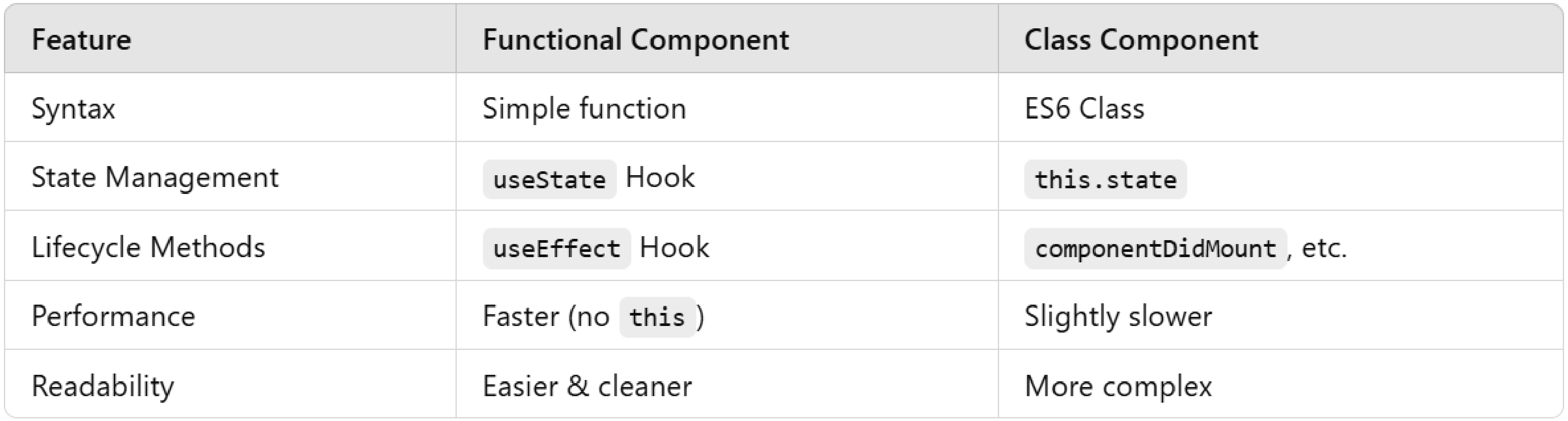
```
import React from "react";

class Welcome extends React.Component {
  render() {
    return (
      <h1>
        Hello, {this.props.name}
      </h1>
    );
  }
}

export default Welcome;
```



```
import React from "react";
import ReactDOM from "react-dom/client";
import Welcome from "./Welcome";  // ✅ Import Welcome

const root = ReactDOM.createRoot(document.getElementById("root"));

root.render(
  <React.StrictMode>
    <Welcome name="Jenila" />  {/* ✅ Render Welcome component */}
  </React.StrictMode>
);
```

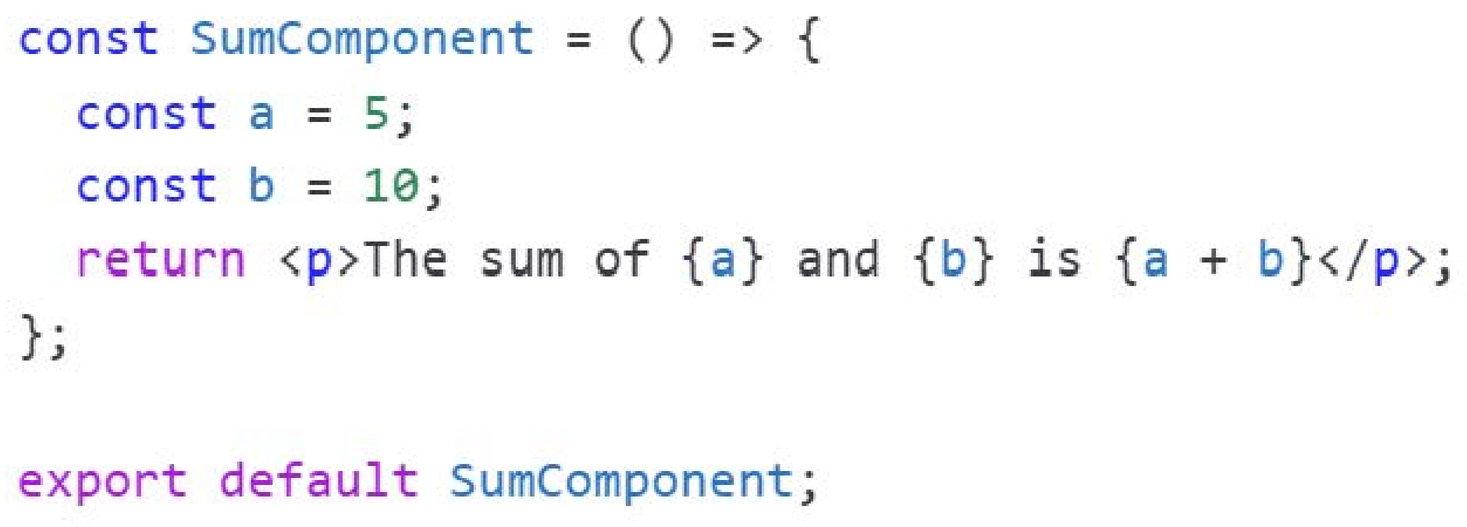
- **extends React.Component is required** for class components.
- Class components must have a render() method to describe what should be displayed in the UI.

Functional Vs Class Component



Feature	Functional Component	Class Component
Syntax	Simple function	ES6 Class
State Management	`useState` Hook	`this.state`
Lifecycle Methods	`useEffect` Hook	`componentDidMount`, etc.
Performance	Faster (no `this`)	Slightly slower
Readability	Easier & cleaner	More complex

Addition of two numbers

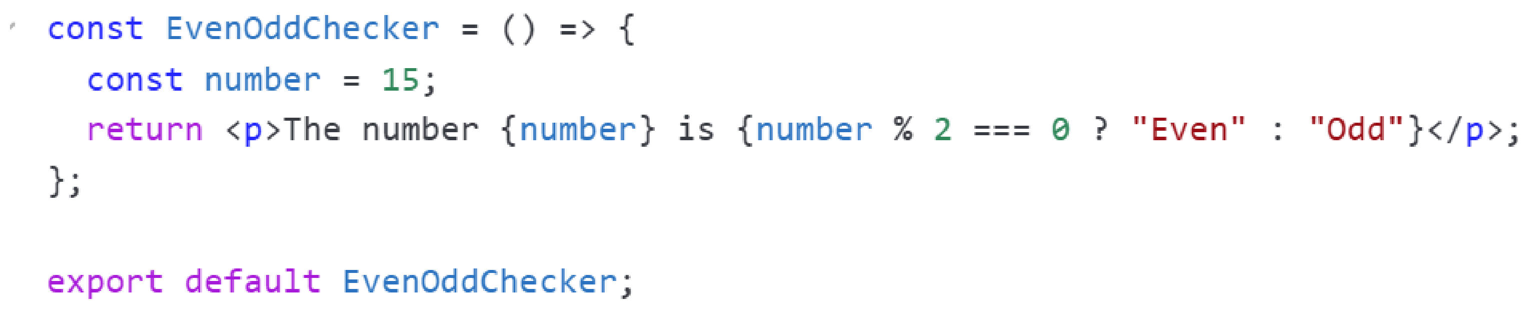


```
const SumComponent = () => {
  const a = 5;
  const b = 10;
  return <p>The sum of {a} and {b} is {a + b}</p>;
};

export default SumComponent;
```

const SumComponet = () => {} defines a functional component in React using an arrow function.

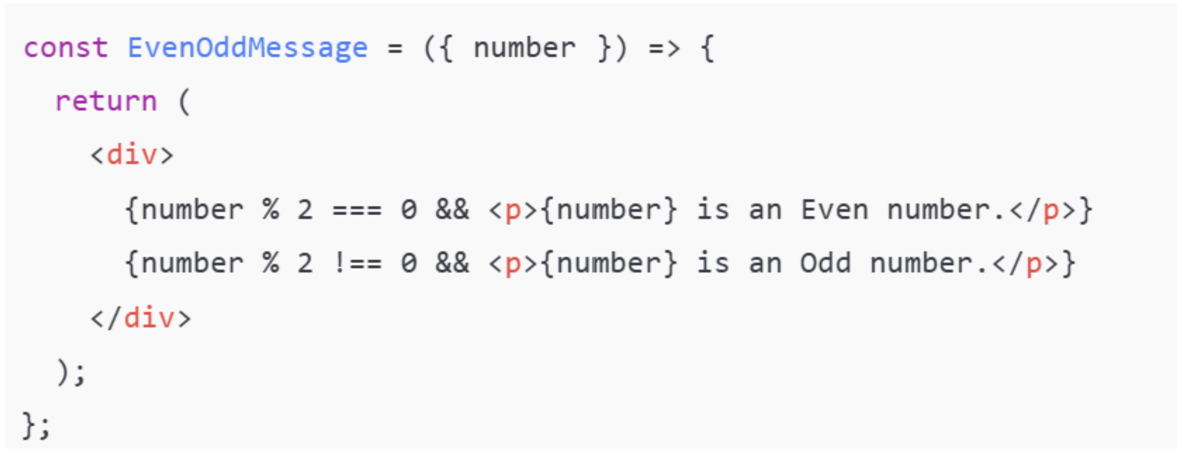
Odd/Even

Using the Ternary Operator (? :)



```
const EvenOddChecker = () => {
  const number = 15;
  return <p>The number {number} is {number % 2 === 0 ? "Even" : "Odd"}</p>;
};

export default EvenOddChecker;
```

Using Logical AND (&&)

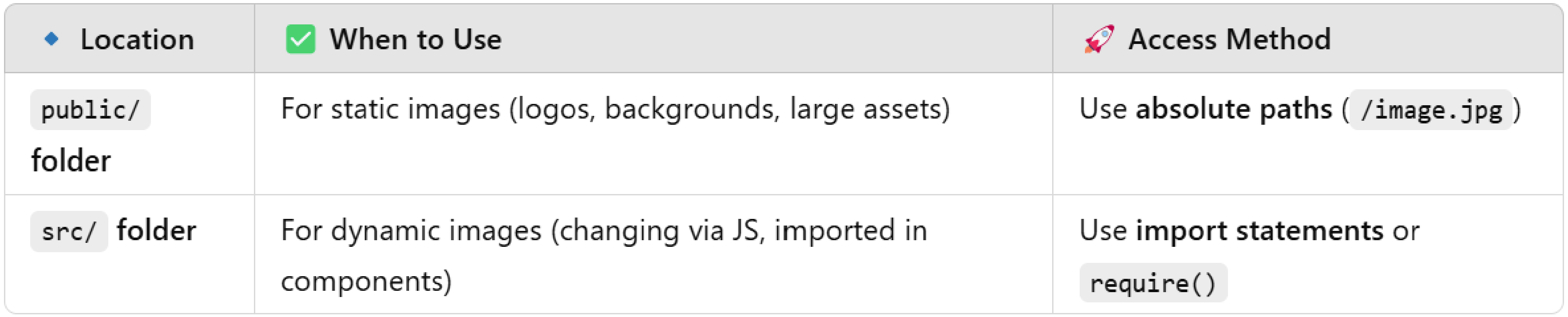


```
const EvenOddMessage = ({ number }) => {
  return (
    <div>
      {number % 2 === 0 && <p>{number} is an Even number.</p>}
      {number % 2 !== 0 && <p>{number} is an Odd number.</p>}
    </div>
  );
};
```

HTML Form

```
import React from "react";

function App() {
  return (
    <div>
      <h2>Contact Us</h2>
      <form>
        {/* Name Input */}
        <label>Name:</label>
        <input type="text" name="name" />

        {/* Email Input */}
        <label>Email:</label>
        <input type="email" name="email" />

        {/* Message Input */}
        <label>Message:</label>
        <textarea name="message"></textarea>

        {/* Submit Button */}
        <button type="submit">Submit</button>
      </form>
    </div>
  );
}

export default App;
```

HTML Images



🔹 Location	✅ When to Use	🚀 Access Method
`public/` folder	For static images (logos, backgrounds, large assets)	Use **absolute paths** (`/image.jpg`)
`src/` folder	For dynamic images (changing via JS, imported in components)	Use **import statements** or `require()`

Using Images from the public/ Folder (Direct URL)

```
<img src="/images/logo.png" alt="Logo" width="200" />
```

Using Images from the src/ Folder

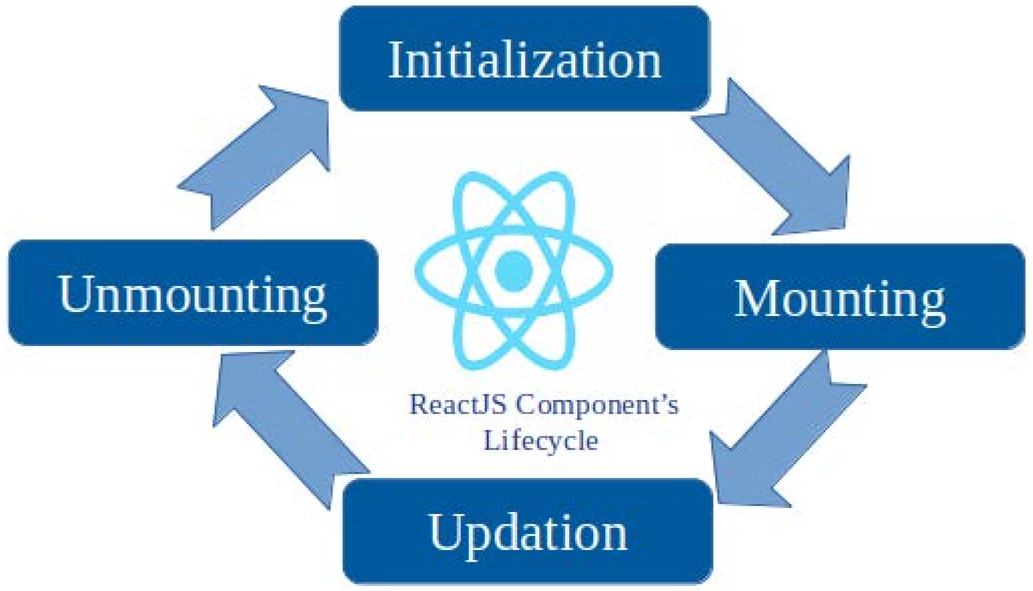
```
<img src={require("./assets/images/logo.png")} alt="Logo" width="200" />
```

(or)

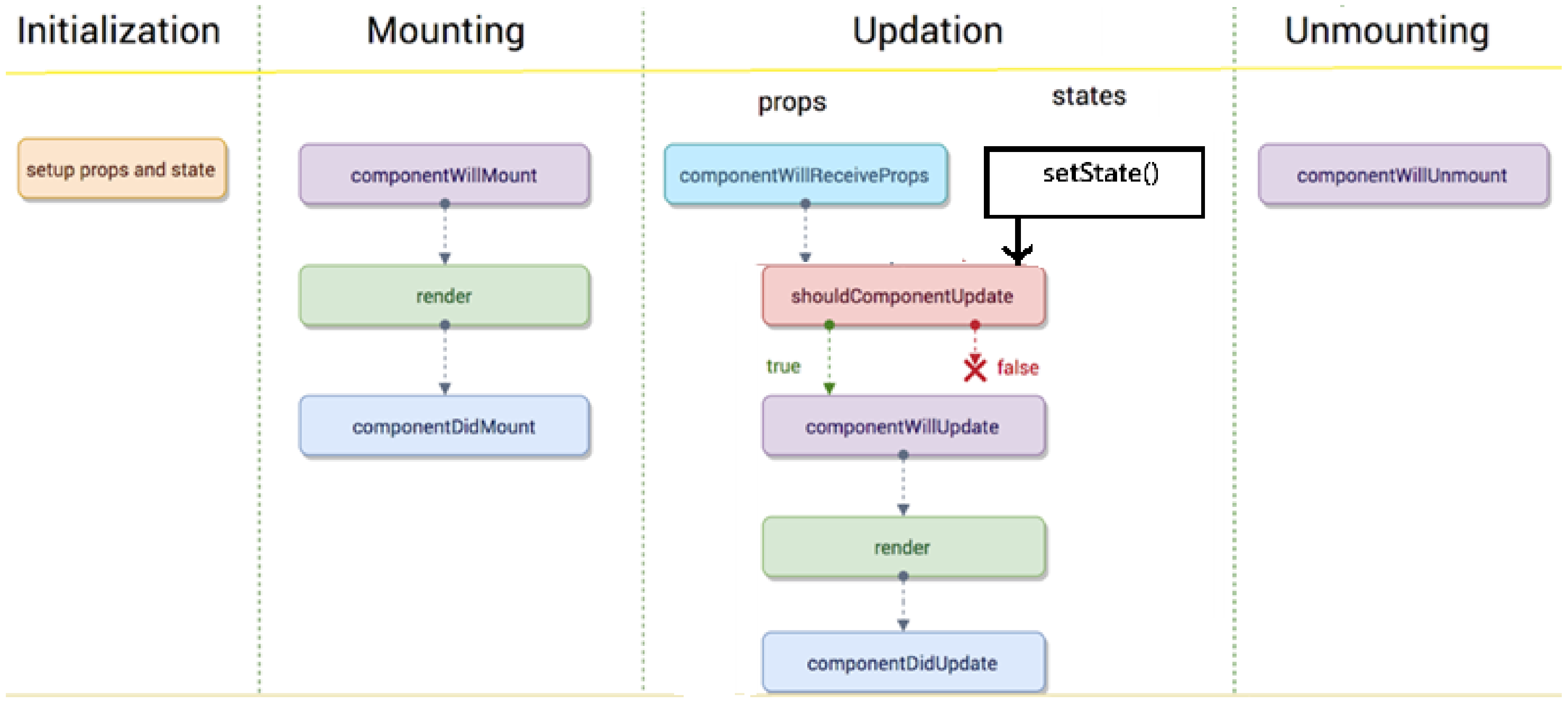
```
import React from "react";
import logo from "./assets/images/logo.png"; // Import the image

function App() {
  return (
    <div>
      <h1>Welcome to My React App</h1>
      <img src={logo} alt="Logo" width="200" /> {/* Use the imported image */}
    </div>
  );
}
```

React Component (class) Life Cycle

- React class components go through **Initialization, Mounting, Updating, and Unmounting** phases.
- Lifecycle methods help in managing component behavior.



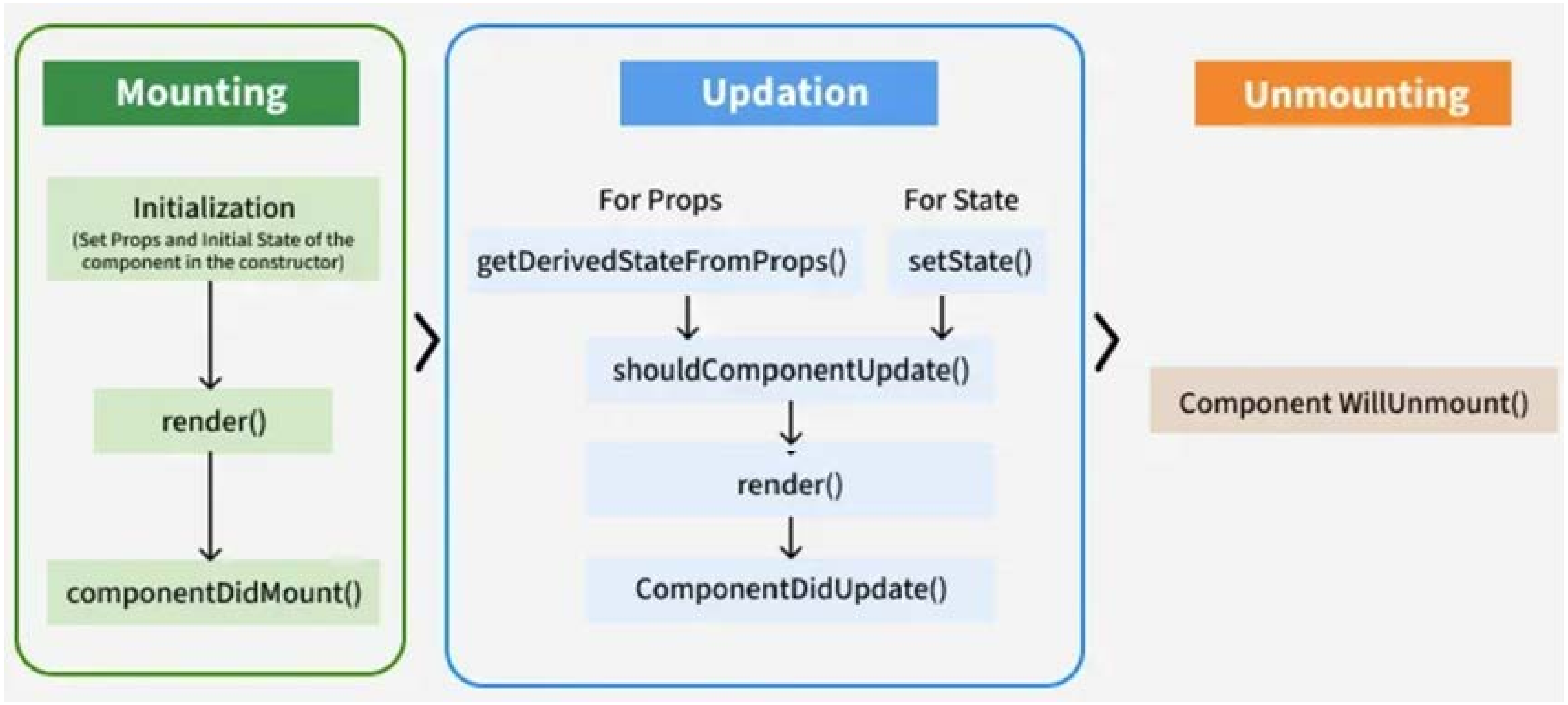
React Component (class) Life Cycle



React Component (class) Life Cycle

- **Initialization:** This phase involves setting props and initializing the state in the constructor.
  - constructor()
- **Mounting:**
  - componentWillMount(),
  - render(),
  - componentDidMount().
- **Updating:**
  - getDerivedStateFromProps()
  - shouldComponentUpdate (nextProps, nextState),
  - componentWillUpdate(nextProps, nextState),
  - render(),
  - componentDidUpdate (prevProps, prevState),
- **Unmounting:** componentWillUnmount().

React Component Life Cycle (Simplified)



.

Mounting (Component Creation)

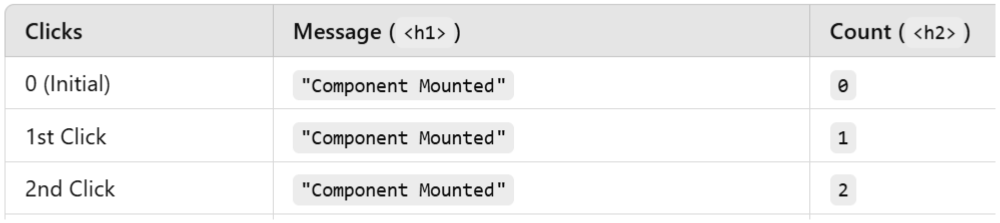
Mounting (Component Creation)
This phase happens **when the component is added to the DOM**.

🚀 Lifecycle methods:

1️⃣ **constructor()**: Initializes state & props.
2️⃣ **render()**: Returns JSX to display.
3️⃣ **componentDidMount()**: Runs after the component is mounted (used for API calls, event listeners).

Example (Mounting)

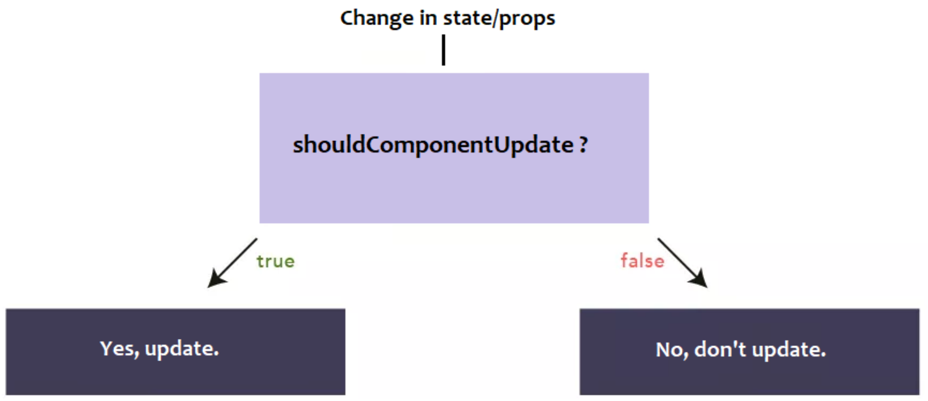
```
import React, { Component } from "react";

class MyComponent extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, message: "Constructor Called" };
  }

  componentDidMount() {
    this.setState({ message: "Component Mounted" });
  }

  render() {
    return (
      <div>
        <h1>{this.state.message}</h1>
        <h2>Count: {this.state.count}</h2>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Increment
        </button>
      </div>
    );
  }
}

export default MyComponent;
```



Clicks	Message (`<h1>`)	Count (`<h2>`)
0 (Initial)	`"Component Mounted"`	`0`
1st Click	`"Component Mounted"`	`1`
2nd Click	`"Component Mounted"`	`2`

Updating (State/Props Change)

This phase occurs **when a component's state or props change**.



🚀 Lifecycle methods:

1️⃣ **shouldComponentUpdate(nextProps, nextState)**: Optimizes re-renders (returns true or false).
2️⃣ **render()**: Re-renders the component.
3️⃣ **componentDidUpdate(prevProps, prevState)**: Runs after re-render (useful for API calls, DOM updates).

Example (Updating)

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

class MyComponent extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, message: "Constructor - Component Created" };
  }

  shouldComponentUpdate(nextProps, nextState) {
    this.setState({ message: "shouldComponentUpdate - Checking Re-render" });
    return true; // Returning false would prevent re-rendering
  }

  componentDidUpdate(prevProps, prevState) {
    this.setState({ message: "componentDidUpdate - Component Updated" });
  }

  render() {
    return (
      <div>
        <h1>{this.state.message}</h1>
        <h2>Count: {this.state.count}</h2>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Increment
        </button>
      </div>
    );
  }
}

export default MyComponent;
```

📝 Expected Output in Browser

Action	Message (`<h1>`)	Count (`<h2>`)
Initial Render	`"Constructor - Component Created"`	`0`
Click "Increment"	`"shouldComponentUpdate - Checking Re-render"`	`1`
After Update	`"componentDidUpdate - Component Updated"`	`1`
Click Again	`"shouldComponentUpdate - Checking Re-render"`	`2`
After Update	`"componentDidUpdate - Component Updated"`	`2`

Unmounting (Component Removal)

This phase happens **when the component is removed from the DOM**.

🚀 Lifecycle method:

- **componentWillUnmount()**: Runs before a component is removed/ destroyed (useful for cleanup).

Example

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

class MyComponent extends Component {
  componentWillUnmount() {
    console.log("🗑️ componentWillUnmount - Component is being removed!");
  }

  render() {
    return <h1>Component is mounted</h1>;
  }
}

export default MyComponent;
```

React Dev Tools

- React DevTools is a **browser extension** for inspecting React component trees.
- Available for **Chrome**, **Firefox**, and as a standalone app.
- Helps developers debug, inspect state/props, and optimize performance.
- **Component Tree View** – Inspect the hierarchy of components.
- **Props & State Inspection** – View and modify state/props in real-time.
- **Highlighting Updates** – See which components re-render during state changes.

Installing React DevTools for Chrome & Firefox:

1. Open the **Chrome Web Store** or **Firefox Add-ons**.
2. Search for **React Developer Tools**.
3. Click **Add to Browser** and enable the extension.

React Dev Tools

1️⃣ **Open DevTools** – Press F12 or Ctrl + Shift + I and go to the **React** tab.

2️⃣ **Select Components** – Hover over a component in the tree.

3️⃣ **Edit Props/State** – Click on a component to modify values dynamically.

4️⃣ **Check Render Performance** – Use the **Profiler** tab to analyze renders.

ReactJS vs React Native

- **ReactJS:** Used for building web applications.
- **React Native:** Used for building mobile applications.
- ReactJS is executed in the browser, while React Native runs on mobile devices.
- React Native does not use HTML but native components.

Thank You!