What is SPA (Single Page Application)?

A single-page application (SPA) is a web application or website that interacts with the user by dynamically rewriting the current web page with new data from the web server, rather than loading entirely new pages, resulting in faster transitions and a more native-app-like feel.

React enables **Single Page Applications (SPA)** by managing routing on the client-side using **React Router** instead of requiring full-page reloads. When a user clicks on different links, React dynamically renders components without reloading the entire page, making the application feel smooth and fast.

Assume a website with two sections:

1. **Home Page** ("/") – Displays general information.
2. **About Page** ("/about") – Displays details about the application.

Instead of making separate HTML files for each page, React dynamically renders components when users navigate.

Must: *npm install react-router-dom*

*[Code written in main.jsx for routing using Vite build tool]*

*Like i did in Workforce-frontend dashboard – where I made multiple tabs like Home, Projects, Todo, and Reminders. Each of them rendered based on the value of a select variable, routing was done in main.jsx.*

**How This Differs from Traditional Websites**

| **Feature** | **Traditional Multi-Page** | **React SPA** |
| --- | --- | --- |
| Page Reload | Yes (every click) | No (only re-renders component) |
| Speed | Slower (server request) | Faster (client-side rendering) |
| User Experience | Flickering & wait time | Smooth transitions |

What is JSX (JavaScript XML)?

JSX stands for JavaScript XML. It allows to write HTML inside JavaScript and place them in the DOM without using functions like appendChild( ) or createElement( ).

As stated in the official docs of React, JSX provides syntactic sugar for React.createElement( ) function.

Note- We can create react applications without using JSX as well.

Let’s understand **how JSX works**:

Without using JSX, we would have to create an element by the following process:

const text = React.createElement('p', {}, 'This is a text');

const container = React.createElement('div','{}',text );

ReactDOM.render(container,rootElement);

**Using JSX**, the above code can be simplified:

const container = (

<div>

<p>This is a text</p>

</div>

);

ReactDOM.render(container,rootElement);

[JSX](https://www.geeksforgeeks.org/jsx-full-form)is basically a syntax extension of regular JavaScript and is used to create React elements. These elements are then rendered to the React DOM. All the React components are written in JSX. To embed any JavaScript expression in a piece of code written in JSX we will have to wrap that expression in curly braces {}.

**Example of JSX:** The name written in curly braces { } signifies JSX

const name = "Learner";  
const element = (  
 <h1>  
 Hello,  
 {name}.Welcome to GeeksforGeeks.  
 </h1>  
);

While it resembles HTML, JSX is not directly interpreted by browsers and requires transpilation into standard JavaScript.

JSX allows embedding JavaScript expressions within curly braces {} directly into the markup. This feature enables dynamic rendering of content, conditional logic, and data manipulation within the UI structure. Under the hood, JSX is syntactic sugar for the React.createElement function, which generates React elements—plain JavaScript objects representing the UI structure. These elements are then used by React to efficiently update the actual DOM.

Although not mandatory, adopting JSX is widely recommended in React development due to its enhanced readability, maintainability, and tooling support. It streamlines UI development by keeping markup and logic closely integrated, leading to more organized and efficient code.

**In general, browsers are not capable of reading JSX and only can read pure JavaScript. The web browsers read JSX with the help of a transpiler. Transpilers are used to convert JSX into JavaScript. The transpiler used is called Babel.**

What is the difference between Functional Components and Class Components?

Before the introduction of Hooks in React, functional components were called stateless components and were behind class components on a feature basis. After the introduction of Hooks, functional components are equivalent to class components.

Although functional components are the new trend, the react team insists on keeping class components in React. Therefore, it is important to know how these components differ.

On the following basis let’s compare functional and class components:

* **Declaration**

Functional components are nothing but JavaScript functions and therefore can be declared using an arrow function or the function keyword:

function card(props){

return(

<div className="main-container">

<h2>Title of the card</h2>

</div>

)

}

const card = (props) =>{

return(

<div className="main-container">

<h2>Title of the card</h2>

</div>

)

}

Class components, on the other hand, are declared using the ES6 class:

class Card extends React.Component{

constructor(props){

super(props);

}

render(){

return(

<div className="main-container">

<h2>Title of the card</h2>

</div>

)

}

}

* **Handling props**

Let’s render the following component with props and analyse how functional and class components handle props:

<Student Info name="Vivek" rollNumber="23" />

In functional components, the handling of props is pretty straightforward. Any prop provided as an argument to a functional component can be directly used inside HTML elements:

function StudentInfo(props){

return(

<div className="main">

<h2>{props.name}</h2>

<h4>{props.rollNumber}</h4>

</div>

)

}

In the case of class components, props are handled in a different way:

class StudentInfo extends React.Component{

constructor(props){

super(props);

}

render(){

return(

<div className="main">

<h2>{this.props.name}</h2>

<h4>{this.props.rollNumber}</h4>

</div>

)

}

}

As we can see in the code above, **this**keyword is used in the case of class components.

* **Handling state**

Functional components use React hooks to handle state. It uses the useState hook to set the state of a variable inside the component:

function ClassRoom(props){

let [studentsCount,setStudentsCount] = useState(0);

const addStudent = () => {

setStudentsCount(++studentsCount);

}

return(

<div>

<p>Number of students in class room: {studentsCount}</p>

<button onClick={addStudent}>Add Student</button>

</div>

)

}

Since useState hook returns an array of two items, the first item contains the current state, and the second item is a function used to update the state.

In the code above, using array destructuring we have set the variable name to studentsCount with a current value of “0” and setStudentsCount is the function that is used to update the state.

For reading the state, we can see from the code above, the variable name can be directly used to read the current state of the variable.

We cannot use React Hooks inside class components, therefore state handling is done very differently in a class component:

Let’s take the same above example and convert it into a class component:

class ClassRoom extends React.Component{

constructor(props){

super(props);

this.state = {studentsCount : 0};

this.addStudent = this.addStudent.bind(this);

}

addStudent(){

this.setState((prevState)=>{

return {studentsCount: prevState.studentsCount++}

});

}

render(){

return(

<div>

<p>Number of students in class room: {this.state.studentsCount}</p>

<button onClick={this.addStudent}>Add Student</button>

</div>

)

}

}

In the code above, we see we are using **this.state** to add the variable studentsCount and setting the value to “0”.

For reading the state, we are using **this.state.studentsCount**.

For updating the state, we need to first bind the addStudent function to **this**. Only then, we will be able to use the **setState** function which is used to update the state.

### ****Difference between Class Components and Functional Components in React****

React provides two ways to create components: **Class Components** and **Functional Components**. Over time, Functional Components with **Hooks** have become the preferred approach due to better performance and cleaner code.

## **1 ⃣ Class Components**

Class components are ES6 classes that extend React.Component and have a **render()** method to return JSX. They can manage **state** and use **lifecycle methods**.

### ****Example of Class Component:****

import React, { Component } from "react";

class Welcome extends Component {

constructor(props) {

super(props);

this.state = { count: 0 };

}

increment = () => {

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

};

render() {

return (

<div>

<h2>Class Component</h2>

<p>Count: {this.state.count}</p>

<button onClick={this.increment}>Increment</button>

</div>

);

}}

export default Welcome;

### ****Key Features of Class Components:****

✅ Uses this.state for managing state.  
✅ Lifecycle methods (componentDidMount, componentDidUpdate, componentWillUnmount).  
✅ Uses this.props to access props.  
✅ More verbose code compared to Functional Components.

## **2 ⃣ Functional Components**

Functional components are JavaScript functions that take props as an argument and return JSX. They **do not have lifecycle methods or state** by default but can manage state using **Hooks**.

### ****Example of Functional Component (with Hooks):****

import React, { useState } from "react";

function Welcome() {

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

return (

<div>

<h2>Functional Component</h2>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);}

export default Welcome;

### ****Key Features of Functional Components:****

✅ Simpler syntax, no need for this.  
✅ Uses **Hooks** (useState, useEffect, etc.) for state and side effects.  
✅ Better performance due to fewer overheads.  
✅ Easier to test and maintain.

|  |  |  |
| --- | --- | --- |
| Scenario | Use Class Components | Use Functional Components |
| Managing state | ✅ (Uses this.state) | ✅ (Uses useState Hook) |
| Using Lifecycle Methods | ✅ (Uses componentDidMount, componentDidUpdate) | ✅ (Replaced with useEffect) |
| Performance & Optimization | ❌ Less optimized | ✅ Better performance |
| Code Complexity | ❌ More complex, uses this | ✅ Simpler, easy to read |
| Hooks Support | ❌ Not available | ✅ Uses useState, useEffect, etc. |
| Legacy Codebase | ✅ Used in older React versions | ❌ Only in newer projects |
| State & Side Effects Handling | ✅ Traditional approach | ✅ Hooks simplify logic |
| Reusability | ❌ Harder to reuse | ✅ Custom Hooks enable reusability |

Controlled vs Uncontrolled Components

Controlled and uncontrolled components are just different approaches to handling input from elements in react.

| **Feature** | **Uncontrolled** | **Controlled** | **Name attrs** |
| --- | --- | --- | --- |
| **One-time value retrieval (e.g. on submit)** | ✔️ | ✔️ | ✔️ |
| **Validating on submit** | ✔️ | ✔️ | ✔️ |
| **Field-level Validation** | ❌ | ✔️ | ✔️ |
| **Conditionally disabling submit button** | ❌ | ✔️ | ✔️ |
| **Enforcing input format** | ❌ | ✔️ | ✔️ |
| **several inputs for one piece of data** | ❌ | ✔️ | ✔️ |
| **dynamic inputs** | ❌ | ✔️ | 🤔 |

* **Controlled component:**In a controlled component, the value of the input element is controlled by React. We store the state of the input element inside the code, and by using event-based callbacks, any changes made to the input element will be reflected in the code as well.

When a user enters data inside the input element of a controlled component, onChange function gets triggered and inside the code, we check whether the value entered is valid or invalid. If the value is valid, we change the state and re-render the input element with the new value.

Example of a controlled component:

function FormValidation(props) {

let [inputValue, setInputValue] = useState("");

let updateInput = e => {

setInputValue(e.target.value);

};

return (

<div>

<form>

<input type="text" value={inputValue} onChange={updateInput} />

</form>

</div>

);

}

As one can see in the code above, the value of the input element is determined by the state of the**inputValue**variable. Any changes made to the input element is handled by the **updateInput** function.

* **Uncontrolled component:** In an uncontrolled component, the value of the input element is handled by the DOM itself. Input elements inside uncontrolled components work just like normal HTML input form elements.

The state of the input element is handled by the DOM. Whenever the value of the input element is changed, event-based callbacks are not called. Basically, react does not perform any action when there are changes made to the input element.

Whenever use enters data inside the input field, the updated data is shown directly. To access the value of the input element, we can use **ref**.

Example of an uncontrolled component:

function FormValidation(props) {

let inputValue = React.createRef();

let handleSubmit = e => {

alert(`Input value: ${inputValue.current.value}`);

e.preventDefault();

};

return (

<div>

<form onSubmit={handleSubmit}>

<input type="text" ref={inputValue} />

<button type="submit">Submit</button>

</form>

</div>

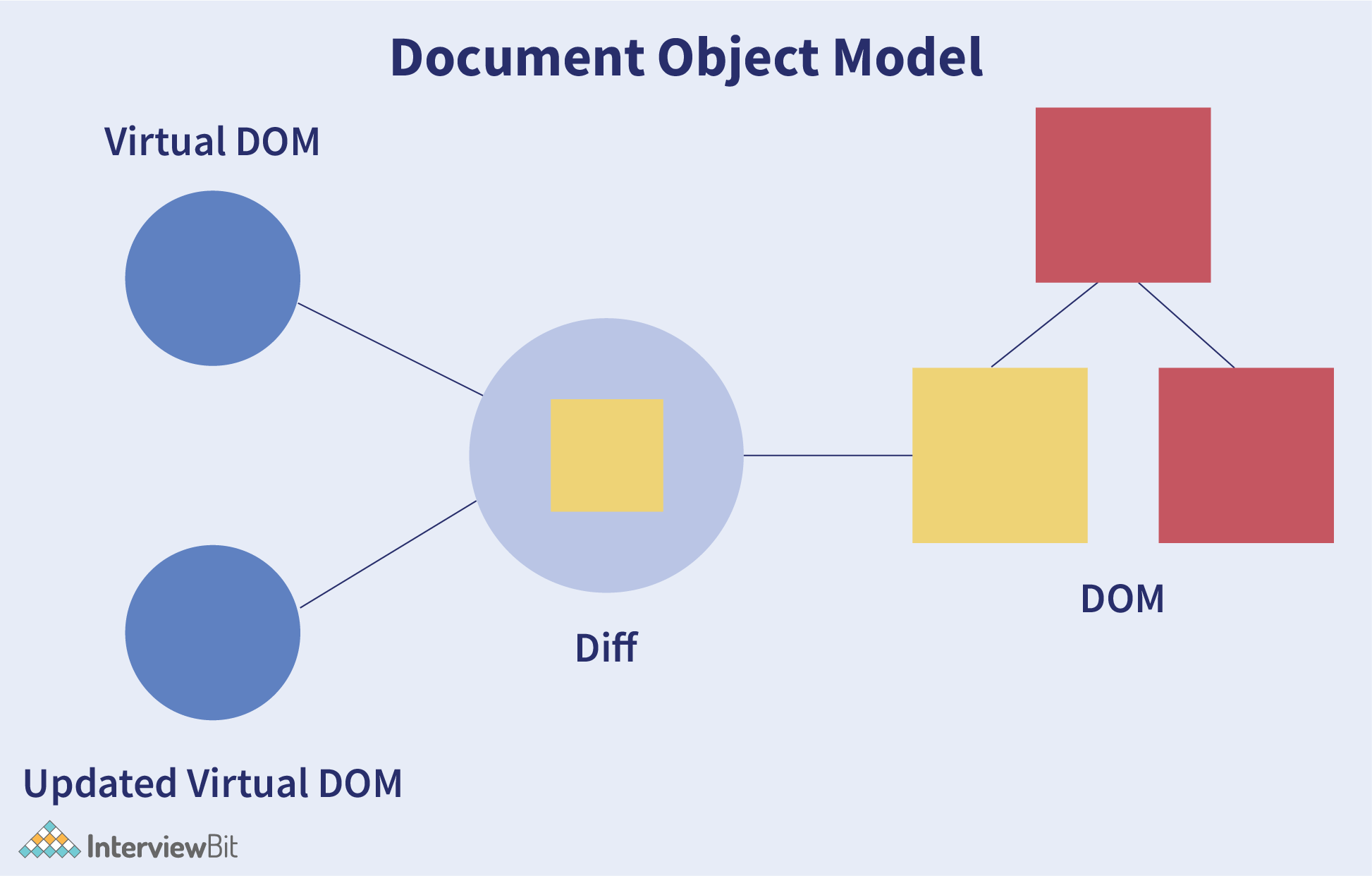
);

}

As one can see in the code above, we are **not** using **onChange** function to govern the changes made to the input element. Instead, we are using **ref** to access the value of the input element.

### What is the virtual DOM? How does react use the virtual DOM to render the UI?

As stated by the react team, virtual DOM is a concept where a virtual representation of the real DOM is kept inside the memory and is synced with the real DOM by a library such as ReactDOM.



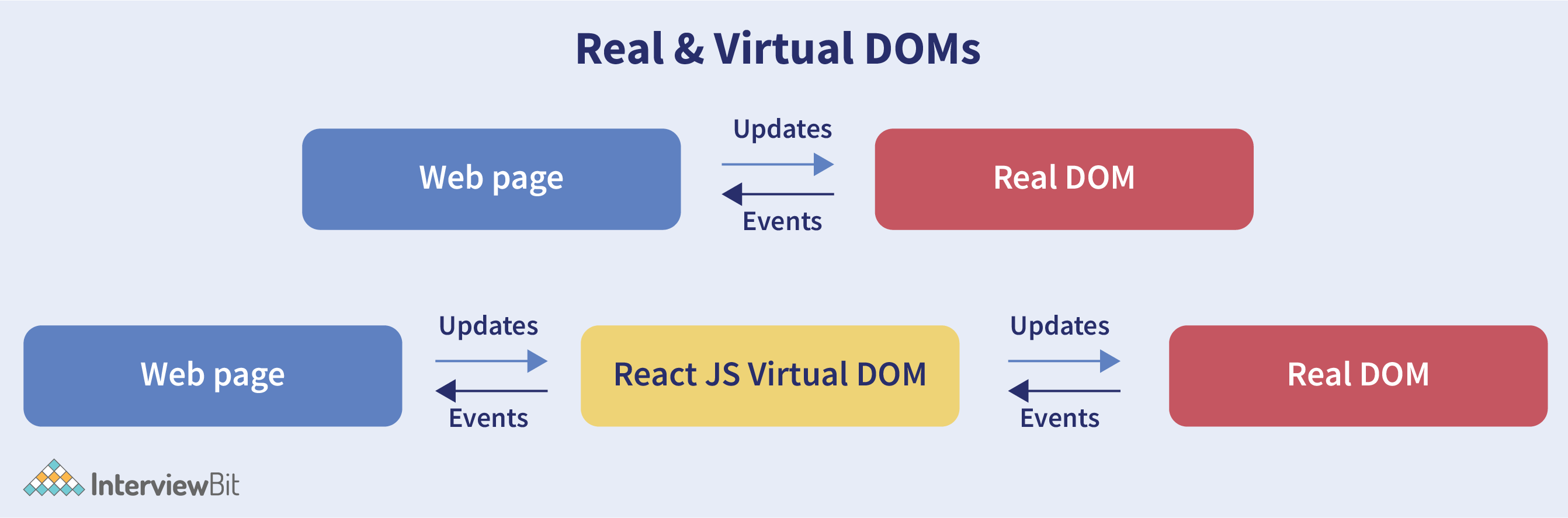
**Why was virtual DOM introduced?**

DOM manipulation is an integral part of any web application, but DOM manipulation is quite slow when compared to other operations in JavaScript. The efficiency of the application gets affected when several DOM manipulations are being done. Most JavaScript frameworks update the entire DOM even when a small part of the DOM changes.

For example, consider a list that is being rendered inside the DOM. If one of the items in the list changes, the entire list gets rendered again instead of just rendering the item that was changed/updated. This is called inefficient updating.

To address the problem of inefficient updating, the react team introduced the concept of virtual DOM.

**How does it work?**



For every DOM object, there is a corresponding virtual DOM object(copy), which has the same properties. The main difference between the real DOM object and the virtual DOM object is that any changes in the virtual DOM object will not reflect on the screen directly. Consider a virtual DOM object as a blueprint of the real DOM object. Whenever a JSX element gets rendered, every virtual DOM object gets updated.

\*\*Note- One may think updating every virtual DOM object might be inefficient, but that’s not the case. Updating the virtual DOM is much faster than updating the real DOM since we are just updating the blueprint of the real DOM.

React uses two virtual DOMs to render the user interface. One of them is used to store the current state of the objects and the other to store the previous state of the objects. Whenever the virtual DOM gets updated, react compares the two virtual DOMs and gets to know about which virtual DOM objects were updated. After knowing which objects were updated, react renders only those objects inside the real DOM instead of rendering the complete real DOM. This way, with the use of virtual DOM, react solves the problem of inefficient updating.

## **Why is Virtual DOM Efficient?**

|  |  |  |
| --- | --- | --- |
| Feature | Traditional DOM | Virtual DOM |
| **Re-renders the whole page?** | ✅ Yes, even for minor changes | ❌ No, only updates changed elements |
| **Performance** | ❌ Slow (Expensive DOM updates) | ✅ Fast (Minimizes updates) |
| **Batch Updates** | ❌ No | ✅ Yes (Combines updates) |
| **Memory Usage** | ❌ Higher | ✅ Lower |
| **Reconciliation** | ❌ Not optimized | ✅ Uses efficient diffing algorithm |

### ****Key Advantages of Virtual DOM:****

✅ **Minimizes direct DOM manipulation** → Faster updates.  
✅ **Efficient Diffing Algorithm** → detects only necessary changes.  
✅ **Batch Updates** → prevents unnecessary re-renders.  
✅ **Improved Performance** → Reduces layout recalculations and reflows.

**What are props in React?**

The props in React are the inputs to a component of React. They can be single-valued or objects having a set of values that will be passed to components of React during creation by using a naming convention that almost looks similar to HTML-tag attributes. We can say that props are the data passed from a parent component into a child component.

The main purpose of props is to provide different component functionalities such as:

* Passing custom data to the React component.
* Using through this.props.reactProp inside render() method of the component.
* Triggering state changes.

For example, consider we are creating an element with reactProp property as given below: <Element reactProp = "1" />  
This reactProp name will be considered as a property attached to the native props object of React which already exists on each component created with the help of React library: props.reactProp;.

**11. Explain React state and props.**

| **Props** | **State** |
| --- | --- |
| Immutable | Owned by its component |
| Has better performance | Locally scoped |
| Can be passed to child components | Writeable/Mutable |
|  | has setState() method to modify properties |
|  | Changes to state can be asynchronous |
|  | can only be passed as props |

* **React State**  
  Every component in react has a built-in state object, which contains all the property values that belong to that component.  
  In other words, the state object controls the behaviour of a component. Any change in the property values of the state object leads to the re-rendering of the component.

Note- State object is not available in functional components but, we can use React Hooks to add state to a functional component.

**How to declare a state object?**

*Example:*

class Car extends React.Component{

constructor(props){

super(props);

this.state = {

brand: "BMW",

color: "black"

}

}

}

**How to use and update the state object?**

class Car extends React.Component {

constructor(props) {

super(props);

this.state = {

brand: "BMW",

color: "Black"

};

}

changeColor() {

this.setState(prevState => {

return { color: "Red" };

});

}

render() {

return (

<div>

<button onClick={() => this.changeColor()}>Change Color</button>

<p>{this.state.color}</p>

</div>

);

}

}

As one can see in the code above, we can use the state by calling **this.state.propertyName** and we can change the state object property using **setState** method.

* **React Props**

Every React component accepts a single object argument called props (which stands for “properties”).  These props can be passed to a component using HTML attributes and the component accepts these props as an argument.

Using props, we can pass data from one component to another.

*Passing props to a component:*

While rendering a component, we can pass the props as an HTML attribute:

<Car brand="Mercedes"/>

The component receives the props:

*In Class component:*

class Car extends React.Component {

constructor(props) {

super(props);

this.state = {

brand: this.props.brand,

color: "Black"

};

}

}

*In Functional component:*

function Car(props) {

let [brand, setBrand] = useState(props.brand);

}

Note- Props are read-only. They cannot be manipulated or changed inside a component.

## **Props and State in React**

In React, **props** and **state** are two essential concepts for handling data within components. They determine how data flows and how the UI updates when data changes.

## **1 ⃣ what are Props in React?**

### ****Definition:****

* **Props (short for "Properties")** are **read-only** data that are passed **from parent to child components**.
* They allow components to be **reusable and dynamic** by receiving external values.
* Props **cannot be modified** inside the child component.

### ****Example of Props:****

function Greeting(props) {

return <h2>Hello, {props.name}!</h2>;

}

function App() {

return <Greeting name="Srajan" />;

}

export default App;

✅ The Greeting component **receives "Srajan" as a prop** and displays it.  
✅ Props **come from the parent component (App)** and cannot be changed inside Greeting.

### ****Key Characteristics of Props:****

✔ **Passed from parent to child**  
✔ **Immutable (Cannot be changed inside the component)**  
✔ **Used for reusability**

## **2 ⃣ What is State in React?**

### ****Definition:****

* **State** is a data container that a component **owns and manages itself**.
* It allows components to **dynamically update the UI** when data changes.
* State is **mutable** (can be changed using setState or useState).

### ****Example of State:****

import React, { useState } from "react";

function Counter() {

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

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

export default Counter;

✅ The Counter component **manages its own count state**.  
✅ When the button is clicked, the state updates, and **React re-renders only the changed part**.

### ****Key Characteristics of State:****

✔ **Managed inside the component**  
✔ **Mutable (can be changed using setState or useState)**  
✔ **Triggers re-rendering**

## **3 ⃣ Difference Between Props and State**

|  |  |  |
| --- | --- | --- |
| Feature | Props | State |
| **Definition** | External data passed from parent to child | Internal data managed by the component |
| **Mutability** | ❌ Immutable (cannot be changed) | ✅ Mutable (can be updated) |
| **Ownership** | Controlled by the **parent** component | Controlled by the **component itself** |
| **Update Effect** | Does **not** trigger re-rendering | **Triggers re-rendering** |
| **Usage** | Used for **passing data** to child components | Used for **dynamic UI updates** |
| **Changes** | Cannot be modified by the component | Modified using setState or useState |

## **4 ⃣ When to Use Props vs. State?**

✅ **Use Props** → when passing **static or external data** (e.g., component settings, API data).  
✅ **Use State** → when a component **needs to update its own data dynamically** (e.g., form inputs, counters).  
✅ **Use Both Together** → a **parent component** can store data in state and pass it to **child components as props**.

## **Conclusion**

* **Props = External data, Read-only, Passed from Parent to Child**
* **State = Internal data, Mutable, Controls Component Behavior**
* React efficiently updates UI using **state changes** and **props flow**.

### ****Types of Side Effects in React Components****

A **side effect** in React refers to **any action performed by a component that affects something outside its scope**, such as updating the DOM, fetching data, or subscribing to events. Since React components should remain **pure functions** (i.e., given the same input, they should produce the same output), **side effects should be handled properly** using the useEffect Hook.

## **1 ⃣ Types of Side Effects in React**

### ****1. Data Fetching Side Effects (API Calls)****

✅ Fetching data from an external API and updating the state when the component mounts.

🔹 **Example (Fetching Data from an API)**

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

function FetchData() {

const [users, setUsers] = useState([]);

useEffect(() => {

fetch("https://jsonplaceholder.typicode.com/users")

.then((response) => response.json())

.then((data) => setUsers(data));

}, []); // Empty dependency array = Runs only on mount

return (

<ul>

{users.map((user) => (

<li key={user.id}>{user.name}</li>

))}

</ul>

);}

export default FetchData;

✅ useEffect runs **once on mount** (empty [] dependency array).  
✅ Prevents unnecessary API calls on every render.

### ****2. Event Listeners Side Effects****

✅ Adding or removing event listeners (e.g., click, keydown events).

🔹 **Example (Adding/Removing a Window Event Listener)**

import React, { useEffect } from "react";

function KeyLogger() {

useEffect(() => {

const logKey = (event) => console.log(`Key Pressed: ${event.key}`);

window.addEventListener("keydown", logKey);

return () => {

window.removeEventListener("keydown", logKey); // Cleanup on unmount

};

}, []);

return <h2>Press any key and check the console!</h2>;

}

export default KeyLogger;

✅ **Cleanup function (return () => {...})** prevents memory leaks.  
✅ window.removeEventListener() runs when the component **unmounts**.

### ****3. Updating the DOM (Manual DOM Manipulation)****

✅ Sometimes, React may not provide built-in ways to manipulate the DOM, requiring direct modifications.

🔹 **Example (Manually Updating Title)**

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

function DynamicTitle() {

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

useEffect(() => {

document.title = `Count: ${count}`;

}, [count]); // Runs when 'count' changes

return (

<button onClick={() => setCount(count + 1)}>Click Me: {count}</button>

);

}

export default DynamicTitle;

✅ React updates the page title dynamically based on count.  
✅ Only runs when count changes to **prevent unnecessary updates**.

### ****4. Subscriptions Side Effects (WebSockets, Intervals)****

✅ Used when a component needs to **subscribe to real-time data** (e.g., WebSockets, Firebase, intervals).

🔹 **Example (Real-time Clock using setInterval)**

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

function Clock() {

const [time, setTime] = useState(new Date().toLocaleTimeString());

useEffect(() => {

const interval = setInterval(() => {

setTime(new Date().toLocaleTimeString());

}, 1000);

return () => clearInterval(interval); // Cleanup on unmount

}, []);

return <h2>Current Time: {time}</h2>;

}

export default Clock;

✅ Uses setInterval to update time every second.  
✅ clearInterval(interval) prevents memory leaks when the component unmounts.

### ****5. Local Storage or Session Storage Side Effects****

✅ Used for **persistent data storage** (e.g., storing user preferences, authentication tokens).

🔹 **Example (Saving Data in Local Storage)**

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

function LocalStorageExample() {

const [name, setName] = useState(() => localStorage.getItem("name") || "");

useEffect(() => {

localStorage.setItem("name", name); // Save to localStorage

}, [name]);

return (

<input

type="text"

value={name}

onChange={(e) => setName(e.target.value)}

placeholder="Enter your name"

/>

);

}

export default LocalStorageExample;

✅ Uses localStorage.getItem() to **persist data across reloads**.  
✅ Updates localStorage whenever name state changes.

## **2 Best Practices for Handling Side Effects in React**

✅ **Always clean up effects** (especially event listeners, intervals, WebSockets).  
✅ **Use dependency arrays correctly** to **avoid infinite loops**.  
✅ **Avoid unnecessary API calls** by using useEffect dependencies.  
✅ **Use useRef for non-reactive values** when needed.

## **3 ⃣ Summary Table: Types of Side Effects in React**

|  |  |  |
| --- | --- | --- |
| **Type** | **Example Use Case** | **Cleanup Needed?** |
| **Data Fetching** | Fetch API data when component mounts | ❌ No (unless polling) |
| **Event Listeners** | Adding/removing click, keydown events | ✅ Yes (removeEventListener) |
| **DOM Updates** | Updating document.title dynamically | ❌ No (React handles updates) |
| **Subscriptions** | WebSockets, Firebase, setInterval | ✅ Yes (clearInterval, unsubscribe) |
| **Local/Session Storage** | Persisting user preferences | ❌ No (unless clearing storage) |

## **Conclusion**

* React **manages side effects with useEffect** to ensure efficiency.
* **Different types of side effects** (API calls, event listeners, storage, etc.) require different handling techniques.
* **Always clean up unnecessary side effects** to **avoid memory leaks and performance issues**.

## **How to Pass Data from Child to Parent?**

We use **a function (callback) defined in the parent** and **pass it as a prop** to the child. The child component **calls this function with data**, and the parent component receives the data.

### ****Example: Sending Data from Child to Parent****

#### **📌 Parent Component (**App.js**)**

import React, { useState } from "react";

import Child from "./Child";

function App() {

const [message, setMessage] = useState("");

// Function to receive data from Child

const handleDataFromChild = (data) => {

setMessage(data);

};

return (

<div>

<h2>Parent Component</h2>

<p>Data from Child: {message}</p>

<Child sendData={handleDataFromChild} />

</div>

);

}

export default App;

#### **📌 Child Component (**Child.js**)**

import React from "react";

function Child({ sendData }) {

const sendMessage = () => {

sendData("Hello from Child!"); // Sending data to Parent

};

return (

<div>

<h3>Child Component</h3>

<button onClick={sendMessage}>Send Data to Parent</button>

</div>

);

}

export default Child;

## **2 Explanation of How It Works**

1 **Parent defines a function** (handleDataFromChild) that updates state when called.  
2 **This function is passed to the Child as a prop (sendData)**.  
3 **Child calls sendData("Hello from Child!") on button click**.  
4 **Parent receives the data and updates the state (message)**.  
5 **Parent UI updates with the new data** received from the Child.

## **3 When to Use This Approach?**

✅ When a child component **needs to send user input** to the parent.  
✅ When a child component triggers an **event that affects the parent state**.  
✅ When you need to **lift state up** for shared state management.

## **Alternative Approach: Using Context API or Redux**

If **multiple components** need to share data, instead of passing callbacks, use:

* **Context API** (For lightweight state management)
* **Redux** (For large-scale apps needing centralized state)

# **Event Handling in React**

Event handling in React is the process of **responding to user interactions**, such as clicks, key presses, form submissions, or mouse movements. React follows a **synthetic event system**, which wraps native browser events to ensure **cross-browser compatibility and performance optimizations**.

## **1 How to Handle Events in React?**

### ****✅ Basic Example: Handling Click Events****

In React, event handlers are written as functions and passed as props to elements.

import React from "react";

function ClickHandler() {

const handleClick = () => {

alert("Button Clicked!");

};

return (

<button onClick={handleClick}>Click Me</button>

);

}

export default ClickHandler;

### ****Key Points****

* The event handler (handleClick) is a function.
* onClick={handleClick} is written in **camelCase** (onClick instead of onclick).
* The function **reference** (not handleClick()) is passed to onClick.

## **2 Event Object in React**

When an event occurs, React provides a **synthetic event object**, which contains information about the event.

### ****✅ Example: Using Event Object****

import React from "react";

function EventExample() {

const handleClick = (event) => {

console.log("Event Type:", event.type);

console.log("Button Text:", event.target.innerText);

};

return (

<button onClick={handleClick}>Click Me</button>

);

}

export default EventExample;

### ****🔹 Key Points****

* event is automatically passed to the event handler.
* event.type gives the event type (e.g., "click").
* event.target refers to the element that triggered the event.

## **3 Handling Different Types of Events**

### ****✅ Click Event****

<button onClick={() => alert("Clicked!")}>Click Me</button>

### ****✅ Mouse Events****

<div onMouseEnter={() => console.log("Mouse Entered")}>

Hover Over Me

</div>

### ****✅ Keyboard Events****

<input type="text" onKeyDown={(e) => console.log("Key Pressed:", e.key)} />

### ****✅ Form Submission Event****

import React, { useState } from "react";

function FormExample() {

const [text, setText] = useState("");

const handleSubmit = (event) => {

event.preventDefault(); // Prevents page reload

alert(`Submitted: ${text}`);

};

return (

<form onSubmit={handleSubmit}>

<input type="text" onChange={(e) => setText(e.target.value)} />

<button type="submit">Submit</button>

</form>

);

}

export default FormExample;

### ****🔹 Key Points****

* event.preventDefault() prevents the default form submission behavior.
* onChange updates the state every time the input changes.

## **4 Passing Arguments to Event Handlers**

Sometimes, you need to pass additional data to an event handler.

### ****✅ Using an Arrow Function****

<button onClick={() => handleClick(5)}>Click Me</button>

### ****✅ Using**** bind()

<button onClick={handleClick.bind(this, 5)}>Click Me</button>

## **5 Event Binding in Class Components**

In **class components**, event binding is required when using this.

### ****✅ Wrong Way (Loses**** this ****Binding)****

class Example extends React.Component {

handleClick() {

console.log(this); // Undefined

}

render() {

return <button onClick={this.handleClick}>Click Me</button>;

}

}

### ****✅ Correct Way (Binding in Constructor)****

class Example extends React.Component {

constructor(props) {

super(props);

this.handleClick = this.handleClick.bind(this);

}

handleClick() {

console.log(this); // Correctly refers to the class instance

}

render() {

return <button onClick={this.handleClick}>Click Me</button>;

}

}

### ****✅ Using Arrow Function****

class Example extends React.Component {

handleClick = () => {

console.log(this);

};

render() {

return <button onClick={this.handleClick}>Click Me</button>;

}

}

🔹 Arrow functions **automatically bind this**, so manual binding isn't required.

## **6 Synthetic Events in React**

React uses **Synthetic Events**, which are lightweight event wrappers over native browser events.

### ****✅ Example: Inspecting Synthetic Events****

import React from "react";

function SyntheticEventExample() {

const handleClick = (event) => {

console.log(event); // Synthetic Event Object

console.log(event.nativeEvent); // Native Browser Event

};

return <button onClick={handleClick}>Click Me</button>;

}

export default SyntheticEventExample;

🔹 **Why Synthetic Events?** ✔ Provides **cross-browser compatibility**.  
✔ Improves **performance** by reusing event objects.  
✔ Reduces **memory usage** by event pooling.

## **7 Preventing Event Bubbling & Capturing**

### ****✅ Stopping Event Bubbling****

<div onClick={() => console.log("Parent Clicked")}>

<button onClick={(e) => e.stopPropagation()}>

Click Me (Doesn't Bubble)

</button>

</div>

🔹 e.stopPropagation() prevents the event from bubbling to the parent.

### ****✅ Event Capturing (Use**** onClickCapture****)****

<div onClickCapture={() => console.log("Captured at Parent")}>

<button onClick={() => console.log("Button Clicked")}>Click Me</button>

</div>

🔹 onClickCapture triggers **before** onClick.

## **8 Handling Multiple Events**

Sometimes, you need to handle multiple events inside a component.

### ****✅ Example: Handling Mouse & Keyboard Events Together****

function MultiEventHandler() {

const handleEvent = (event) => {

if (event.type === "click") {

console.log("Mouse Clicked");

} else if (event.type === "keydown") {

console.log("Key Pressed:", event.key);

}

};

return (

<div>

<button onClick={handleEvent}>Click Me</button>

<input type="text" onKeyDown={handleEvent} />

</div>

);

}

export default MultiEventHandler;

## **9 Summary Table: Event Handling in React**

|  |  |  |
| --- | --- | --- |
| **Event Type** | **Example Handler** | **Use Case** |
| Click Event | onClick={handleClick} | Buttons, divs |
| Mouse Events | onMouseEnter={handleMouseEnter} | Hover effects |
| Keyboard Events | onKeyDown={handleKeyPress} | Forms, shortcuts |
| Form Events | onSubmit={handleSubmit} | Form submission |
| Event Bubbling | e.stopPropagation() | Prevents event propagation |
| Prevent Default | e.preventDefault() | Stops default behavior |

## **Conclusion**

✔ **React uses Synthetic Events** for consistency & performance.  
✔ **Event handlers are functions** that update state or trigger actions.  
✔ **Use useState & useEffect to manage event-driven state changes.**  
✔ **Prevent bubbling, use stopPropagation() when necessary.**  
✔ **Use arrow functions to avoid this binding issues in class components.**

**Best articles(Must go through):**

<https://react.dev/learn/responding-to-events\>

<https://www.freecodecamp.org/news/how-to-handle-events-in-react-19/>