

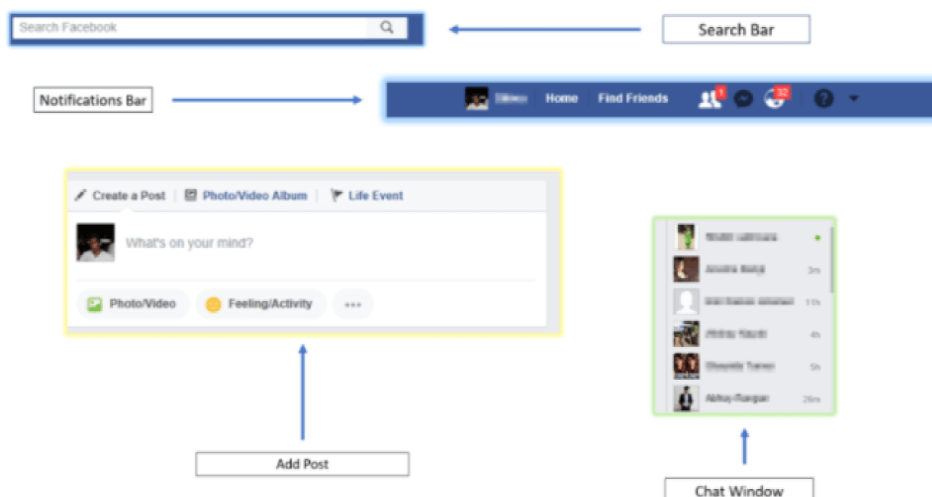
Unit-2: Fundamentals of React.js

Overview of React

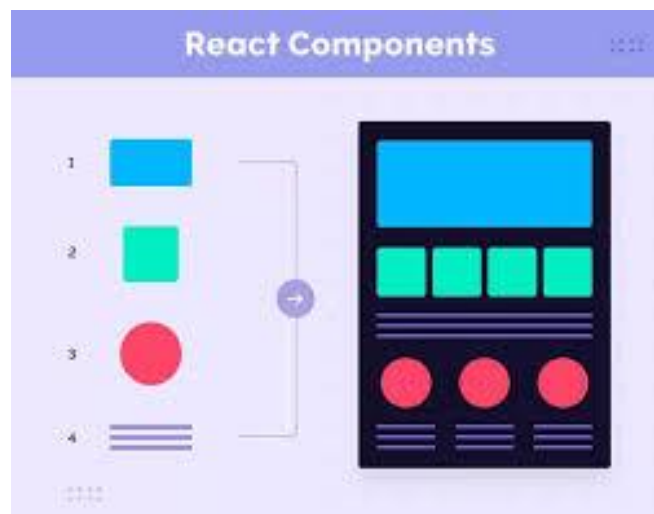
What is React?

React.js is a **front-end JavaScript framework** developed **by Facebook**. To build composable user interfaces predictably and efficiently using declarative code, we use React. It's an open-source and component-based framework responsible for creating the application's view layer.

- React.js follows the Model View Controller (MVC) architecture, and the view layer is accountable for handling mobile and web apps.
- React is famous for building single-page applications and mobile apps.
- As the figure shows, React.js divides the UI into multiple components, making the code easier to debug.
- This way, each function is assigned to a specific component, and it produces some HTML which is rendered as output by the DOM.

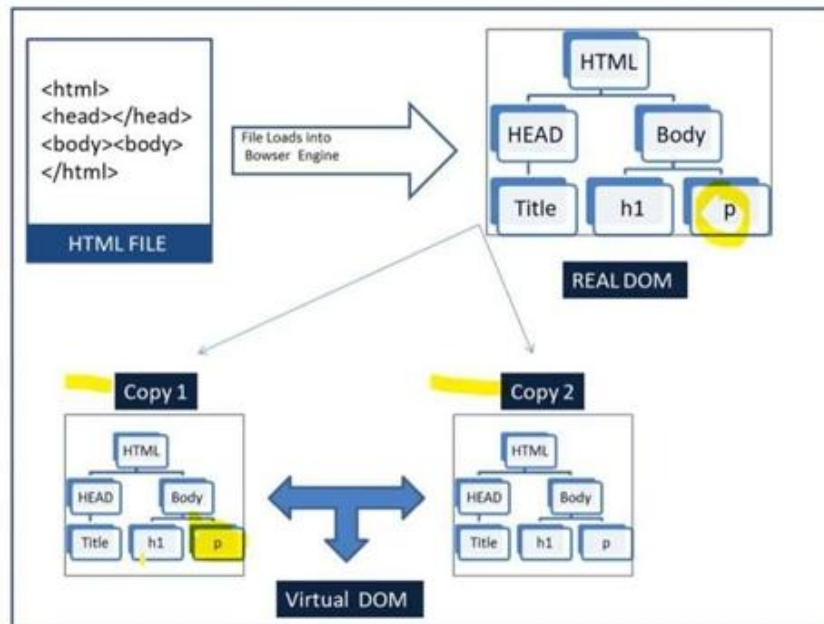


Component:



How does React Work?

- React creates a VIRTUAL DOM in memory.



- Instead of manipulating the browser's DOM directly, React creates a virtual DOM in memory, where it does all the necessary manipulating, before making the changes in the browser DOM.
- React only changes what needs to be changed!
- React finds out what changes have been made, and changes only what needs to be changed.
- You will learn the various aspects of how React does this in the rest of this tutorial.

ReactJS History

- Facebook **Software Engineer, Jordan Walke**, created it.
- React.JS was **first used in 2011 for Facebook's Newsfeed feature**.
- Initial **Release to the Public (V0.3.0) was in July 2013**.
- Current version of create-react-app is v18.2.0 (14 june 2022).
- create-react-app includes built tools such as webpack, Babel, and ESLint.

Why React?

- Easy creation of dynamic applications.
- Improved performance.
- Reusable components.
- Unidirectional data flow.
- Small learning curve.
- It can be used for the development of both web and mobile apps.
- Dedicated tools for easy debugging.

Features:

1. JSX(JavaScript Syntax Extension):

JSX is a combination of HTML and JavaScript. You can embed JavaScript objects inside the HTML elements. JSX is not supported by the browsers, as a result, Babel compiler transcompile the code into JavaScript code. JSX makes codes easy and understandable. It is easy to learn if you know HTML and JavaScript.

```
const name="React World";  
const element = <h1>Welcome to {name}</h1>;
```

2. Virtual DOM:

DOM stands for Document Object Model. It is the most important part of the web as it divides into modules and executes the code. Usually, JavaScript Frameworks updates the whole DOM at once, which makes the web application slow. But react uses virtual DOM which is an exact copy of real DOM. Whenever there is a modification in the web application, the whole virtual DOM is updated first and finds the difference between real DOM and Virtual DOM.



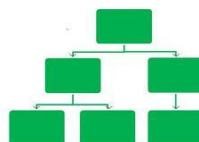
In the above-shown figure, when the whole virtual DOM has updated there is a change in the child components. So, now DOM finds the difference and updates only the changed part.

3. Performance:

As we discussed earlier, react uses virtual DOM and updates only the modified parts. So, this makes the DOM to run faster. DOM executes in memory so we can create separate components which makes the DOM run faster.

4. One-way Data Binding:

One-way data binding, the name itself says that it is a one-direction flow. The data in react flows only in one direction i.e. the data is transferred from top to bottom i.e. from parent components to child components. The properties (props) in the child component cannot return the data to its parent component but it can have communication with the parent components to modify the states according to the provided inputs. Below diagram, data can flow only from top to bottom.



One-way Data Binding

5. Extension:

React has many extensions that we can use to create full-fledged UI applications. It supports mobile app development and provides server-side rendering. React is extended with Flux, Redux, React Native, etc. which helps us to create good-looking UI.

6. Conditional Statements:

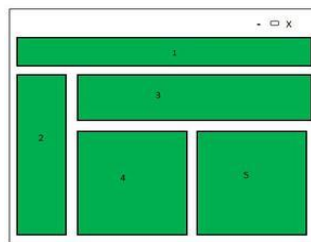
JSX allows us to write conditional statements. The data in the browser is displayed according to the conditions provided inside the JSX.

Syntax:

```
const age = 12;
if (age >= 10)
{
    <p> Greater than { age } </p>;
}
else
{
    <p> { age } </p>;
}
```

7. Components:

React.js divides the web page into multiple components as it is component-based. Each component is a part of the UI design which has its own logic and design as shown in the below image. So the component logic which is written in JavaScript makes it easy and run faster and can be reusable.



Multiple components

8. Simplicity:

React.js is a component-based which makes the code reusable and React.js uses JSX which is a combination of HTML and JavaScript. This makes code easy to understand and easy to debug and has less code.

Benefits:

- Easy to learn
- Easy to adept in modern as well as legacy application
- Faster way to code a functionality
- Availability of large number of ready-made component
- Large and active community

Applications:

Few popular websites powered by React library are listed below –

- Facebook
- Instagram
- Netflix
- Code Academy
- Reddit
- Airbnb

Advantages of React:**1. Easy to Learn and Use**

- React has a simple API and component-based architecture.
- Developers with JavaScript knowledge can quickly grasp React concepts.

2. Component-Based Architecture

- Applications are built using reusable components.
- Each component manages its own logic and rendering, promoting modularity and maintainability.

3. Reusable Components

- Components can be reused across different parts of the application.
- This reduces code duplication and speeds up development.

4. Virtual DOM for High Performance

- React uses a virtual DOM to optimize rendering.
- Only the components that change are updated in the real DOM, improving performance.

5. One-Way Data Binding

- Data flows in a single direction (from parent to child).
- This makes the application more predictable and easier to debug.

6. JSX – JavaScript XML

- JSX allows writing HTML-like syntax directly in JavaScript.
- It improves readability and makes the code more expressive.

7. Strong Community Support

- Large community of developers and contributors.
- Rich ecosystem of libraries, tools, and resources.

8. SEO-Friendly

- React can render on the server side using frameworks like Next.js.
- This improves SEO for web applications compared to traditional client-side rendering.

9. Handy Developer Tools

- React Developer Tools for Chrome and Firefox help inspect component hierarchies.
- Developers can easily debug and analyze props and state.

10. Rich JavaScript Library

- Offers flexibility in choosing tools and libraries.
- Can be integrated with other frameworks or libraries like Redux, React Router, etc.

11. Easy Testing

- React applications are easy to test using tools like Jest and React Testing Library.
- Components can be tested in isolation.

Disadvantages of React:

1. High Pace of Development

- The React ecosystem evolves rapidly.
- Developers may find it challenging to keep up with frequent updates and changes.
- Constant relearning can be uncomfortable for some developers.

2. Poor Documentation

- Due to the fast-paced updates, official documentation often lags behind.
- Developers may need to rely on community-written guides or create their own documentation.

3. Covers Only the View Layer

- ReactJS focuses solely on the UI (View) part of the application.
- Developers must integrate other libraries (like Redux or Flux) for state management and routing to build a full-fledged application.

4. JSX as a Barrier

- JSX mixes HTML with JavaScript, which can be confusing for beginners.
- Some developers find JSX complex and difficult to learn initially.

What is render() in React?

- The render() function is a core method in React used to display content on the web page.
- It plays a crucial role in both initial rendering and updating the UI when data changes.

Key Characteristics

- Used in Class Components: It's a required lifecycle method in class-based components.
- Returns JSX: The function returns JSX (HTML-like syntax) that defines what should appear in the UI.
- Triggers UI Updates: When state or props change, React calls render() to update the component.

Syntax Example

```
import ReactDOM from "react-dom";
ReactDOM.render(<h2>Hello</h2>, document.getElementById('root'));
```

In class components:

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

How It Works

1. Initial Render:

- ReactDOM targets a DOM node (usually <div id="root">) and renders the root component.

2. Re-render on State Change:

- When `setState()` is called, React detects the change and re-invokes `render()` to update the UI.

3. Virtual DOM Optimization:

- React compares the new virtual DOM with the previous one.
- Only the changed elements are updated in the real DOM, improving performance.

ReactJS – Installation:

1. Install vs code or any editor

2. Install Node.js

- Visit: <https://nodejs.org>
- Download and install the **LTS version** (Long Term Support)

3. After install, check version in VS Code terminal: We can check it using the below command

node -version / node -v

4. You could see the version of Node.js you might have installed.

V20.9.0

5. Check npm install or not

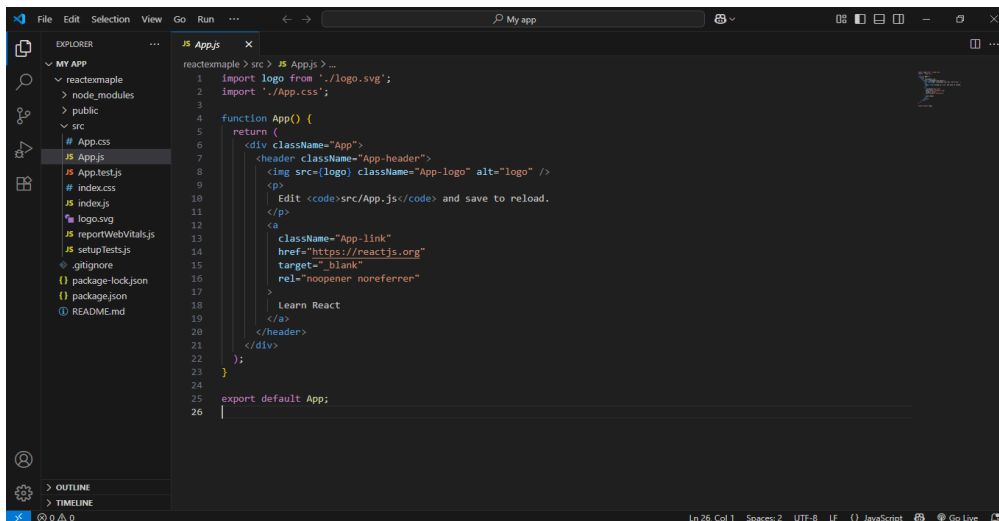
npm -version

6. Now to install react.

npm init react-app reactexample

(Note: always use small letter for name of project)

7. Now this will install react in your Computer



8. How to Run

- 1) Open Terminal in Project Folder
- 2) Then navigate into the folder: **cd reactexample**
- 3) Start Development Server

Run this command to start the React app: **npm start**

- 4) View in Browser

After a few seconds, it will automatically open in your browser at: <http://localhost:3000>

If not, open your browser and go to that address manually.

Create basic program to display hello world App.js

```
import React from 'react';

function App() {
  return (
    <div>
      <h1>Hello World</h1>
    </div>
  );
}
export default App;
```

Using react with HTML:

- React JSX is an extension to JavaScript.
- JSX allows us to write HTML in React.
- JSX makes it easier to write and add HTML in React.
- It enables developer to create virtual DOM using XML syntax.
- It compiles down to pure JavaScript (*React.createElement function calls*).
- JSX converts HTML tags into react elements.
- JSX provides you to write HTML/XML-like structures (e.g., DOM-like tree structures) in the same file where you write JavaScript code, and then pre-processor will transform these expressions into actual JavaScript code. Just like XML/HTML, JSX tags have a tag name, attributes, and children. Components are like functions that return HTML elements. Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML.
- React renders HTML to the web page by using a function called **ReactDOM.render()**.
- Since it compiles to JavaScript, it can be used inside any valid JavaScript code. For example, below codes are perfectly valid.
- Assign to a variable.

```
var greeting = <h1>Hello React!</h1>
```

- Assign to a variable based on a condition.

```
var canGreet = true;
if(canGreet) {
    greeting = <h1>Hello React!</h1>
}
```

- Can be used as return value of a function.

```
function Greeting() {
    return <h1>Hello React!</h1>
}
```

React Interactive components: Components within components and Files

React Components

In React, React components are independent, reusable building blocks in a React application that define what gets displayed on the UI. They accept inputs called props and return React elements describing the UI.

What Are React Components?

A component is a building block that encapsulates a piece of the user interface (UI). It defines the structure and behavior of the UI, either by managing its own state or by receiving data through props and rendering content accordingly. Components are reusable and can be composed together to build complex UIs in a modular way.

- Components can be reused across different parts of the application to maintain consistency and reduce code duplication.
- Components manage how their output is rendered in the DOM based on their state and props.
- React loads only the necessary components, ensuring optimized performance.
- Only the specific Component updates instead of the whole page.

React interactive components, especially components within components (also called **nested** components) and how to organize them in files. In React:

- Component = Reusable UI piece (like button, card, form).
- Interactive = Responds to user actions (click, input).
- Components within Components = One component *calls or includes* another.
- Files = Each component is usually in its own file.

Types of React Components

There are two primary types of React components:

1. Functional Components

Functional components are simpler and preferred for most use cases. They are JavaScript functions that return React elements. With the introduction of React_Hooks, functional components can also manage state and lifecycle events.

- **Stateless or Stateful:** Can manage state using React Hooks.
- **Simpler Syntax:** Ideal for small and reusable components.
- **Performance:** Generally faster since they don't require a 'this' keyword.

```
function Greet(props) {  
    return <h1>Hello, {props.name}!</h1>;  
}
```

2. Class Components

Class components are ES6 classes that extend React. Component. They include additional features like state management and lifecycle methods.

- **State Management:** State is managed using the this.state property.
- **Lifecycle Methods:** Includes methods like componentDidMount, componentDidUpdate, etc.

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

Props in React Components

- Props (short for properties) are read-only inputs passed from a parent component to a child component.
- They enable dynamic data flow and reusability.
- Props are immutable (unchanging over time or unable to be changed.).
- They enable communication between components.

```
function Greet(props) {  
    return <h2>Welcome, {props.username}!</h2>;  
}  
<Greet username="Pooja" />;
```

State in React Components

The state is a JavaScript object managed within a component, allowing it to maintain and update its own data over time. Unlike props, state is mutable and controlled entirely by the component.

- State updates trigger re-renders.
- Functional components use the `useState` hook to manage state.

```
function Counter() {  
  const [count, setCount] = React.useState(0);  
  return (  
    <div>  
      <p>Count: {count}</p>  
      <button onClick={() =>  
        setCount(count + 1)}>Increment</button>  
    </div>  
  );  
}
```

Rendering a Component

Rendering a component refers to displaying it on the browser. React components can be rendered using the `ReactDOM.render()` method or by embedding them inside other components.

- Ensure the component is imported before rendering.
- The `ReactDOM.render` method is generally used in the root file.

```
ReactDOM.render(<Greeting name="Pooja" />, document.getElementById('root'));
```

Components in Components

In React, you can nest components inside other components to build a modular and hierarchical structure.

- Components can be reused multiple times within the same or different components.
- Props can be passed to nested components for dynamic content.

```
function Header() {  
  return <h1>Welcome to My Site</h1>;  
}  
function Footer() {  
  return <p>© 2024 My Company</p>;  
}  
function App() {  
  return (  
    <div>  
      <Header />  
      <p>This is the main content.</p>  
      <Footer />  
    </div>  
  );  
}  
export default App;
```

Example: 1. App.js (Main App Component)

```
import React from 'react';
import Header from './components/Header';
import UserCard from './components/UserCard';

function App() {
  return (
    <div>
      <Header />
      <UserCard name="Pooja" age={26} />
      <UserCard name="Nisha" age={30} />
    </div>
  );
}
export default App;
```

2. components/Header.js

```
import React from 'react';
function Header() {
  return <h1>Welcome to the React App</h1>;
}
export default Header;
```

3. components/UserCard.js

```
import React from 'react';
import Button from './Button';
function UserCard({ name, age }) {
  return (
    <div style={{ border: '1px solid gray', padding: '10px', margin: '10px' }}>
      <h3>Name: {name}</h3>
      <p>Age: {age}</p>
      <Button label="Say Hello" />
    </div>
  );
}
export default UserCard;
```

4. components/Button.js

```
import React from 'react';
function Button({ label }) {
  function handleClick() {
    alert('Hello from Button!');
  }
  return <button onClick={handleClick}>{label}</button>;
}
export default Button;
```

Functional Components vs Class Components

Feature	Functional Components	Class Components
State Management	It can use Hooks like <code>useState</code> , <code>useReducer</code>	It uses <code>this.state</code> and <code>this.setState()</code>
Lifecycle Methods	It uses <code>useEffect</code> Hook for lifecycle methods	It uses traditional lifecycle methods like <code>componentDidMount</code> , <code>componentWillUnmount</code>
Rendering	Returns JSX directly inside the function	Uses a <code>render()</code> method to return <u>JSX</u>
Performance	Faster and more lightweight due to simpler structure	Slightly heavier due to the overhead of class instances
Hooks	Can use React hooks (<code>useState</code> , <code>useEffect</code> , etc.)	Cannot use hooks; relies on lifecycle methods and state
This Keyword	Does not use this keyword	Uses this to access props and state
Code Complexity	Less boilerplate code, easier to write and understand	More boilerplate code, especially for state and methods
Event Handling	Simple and direct event handling	Requires method binding for event handling

Passing data through Props

What are props?

- **Props** (short for "properties") are a way to pass data from one component to another in React.
- Everything in ReactJS is a component and to pass in data to these components, **props** are used. Whenever we call child components from parents we can pass data as props.
- It used to **pass data from one component to another**, usually **from a parent to a child**.
- This helps the parent component communicate with the child.
- Read-only
- Immutable (cannot be changed by the receiving component)
- Similar to function arguments

Why Use Props?

Props allow components to be **dynamic and reusable**. Instead of hard coding values, you can pass them as props and render different outputs based on the data.

Basic Example of Passing Data through Props

Pass Props from Parent Component

1. App.js (Parent Component)

```
import React from 'react';
import Greeting from './Greeting';

function App() {
  return (
    <div>
      <Greeting name="Pooja" />
      <Greeting name="Nisha" />
    </div>
  );
}
export default App;
```

Receive Props in Child Component

2. Greeting.js (Child Component)

```
import React from 'react';
function Greeting(props) {
  return <h2>Hello, {props.name}!</h2>;
}
export default Greeting;
```

Props in Functional vs Class Components

Example: Functional Component with Props

```
function Student(props) {
  return (
    <div>
      <p>Roll No: {props.rno}</p>
      <p>Name: {props.name}</p>
      <p>Gender: {props.gender}</p>
    </div>
  );
}

function App() {
  return (
    <div>
      <Student rno={1} name="Hetvi" gender="Male" />
      <Student rno={2} name="Riya" gender="Female" />
    </div>
  );
}
```

Example: Class Component with Props

```
class Employee extends React.Component {
  render() {
    return (
      <div>
        <p>Employee No: {this.props.eno}</p>
        <p>Name: {this.props.name}</p>
      </div>
    );
  }
}

function App() {
  return (
    <div>
      <Employee eno="E01" name="Rajesh Patel" />
      <Employee eno="E02" name="Ronit Vaghela" />
    </div>
  );
}
```

What is Class components?

React Class

- This term is often used informally to refer to a **class-based component** in React.
- It's not a separate concept—just shorthand for a component defined using ES6 class syntax.

React Class Components

Class components are ES6 classes that extend `React.Component`. They allow state management and lifecycle methods for complex UI logic.

- Used for stateful components before Hooks.
- Support lifecycle methods for mounting, updating, and unmounting.

The `render()` method in React class components returns JSX elements describing the UI of the Application.

Example: App.js

```
import React from "react";

class App extends React.Component {
  render() {
    return <h1>AWD</h1>;
  }
}

export default App;
```

Structure of React Class Components

A typical React class component consists of the following parts:

- **Class Declaration:** The component is declared as a class that extends `React.Component`. This inheritance gives the class access to React's methods and properties.

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

- **Constructor:** The `constructor()` method is used to initialize the component's state and bind event handlers. It is optional, but if used, it must call `super(props)` to initialize the parent Component class.

```
constructor(props) {  
  super(props);  
  this.state = { count: 0 }; // Initialize state  
}
```

- **Render Method:** The `render()` method is the only required method in a class component. It returns `JSX`, which represents the structure of the UI. Whenever the state or props change, the `render()` method is re-invoked.

```
render() {  
  return <div>Current count: {this.state.count}</div>;  
}
```

- **State:** Class components can manage their own internal state using the `this.state` property. The state is mutable and is used to store values that change over time, such as user input or API responses.

```
this.state = {  
  count: 0  
};
```

Event Handlers: Event handlers are typically methods in the class. They are used to handle user interactions like clicks, form submissions, etc. These methods often use `this.setState()` to update the component's state.

```
increment = () => {  
  this.setState({ count: this.state.count + 1 });  
};
```

- **Props in Class Components:** Props allow data to flow from parent components to child components. They are accessible via `this.props`. Props cannot be changed by the component itself; they are controlled by the parent component.

```
class MyComponent extends React.Component {  
  render() {  
    return <div>{this.props.name}</div>; // Access props  
  }  
}
```

Conditional statements, Operators, Lists in react with example

Conditional statements

Conditional rendering in React works similarly to conditions in JavaScript. It enables components to display different outputs depending on states or props. This ensures that the UI updates dynamically based on logic instead of manually manipulating the DOM.

1. Using If/Else Statements

If/else statements allow rendering different components based on conditions. This approach is useful for complex conditions.

```
function Item({ name, isPacked }) {  
  if (isPacked) {  
    return <li className="item">{name}</li>;  
  }  
  return <li className="item">{name}</li>;  
}
```

- If isPacked is true, it displays: Space Suit
- If isPacked is false, it displays: Space Suit

2. Using Ternary Operator

The ternary operator (condition ? expr1: expr2) is a concise way to conditionally render JSX elements. It's often used when the logic is simple and there are only two options to render.

```
function Greeting({ isLoggedIn }) {  
  return <h1>{isLoggedIn ? "Welcome Back!" : "Please Sign In"}</h1>;  
}
```

- If isLoggedIn is true: Welcome Back!
- If isLoggedIn is false: Please Sign In

3. Using Logical AND (&&) Operator

The && operator returns the second operand if the first is true, and nothing if the first is false. This can be useful when you only want to render something when a condition is true.

```
function Notification({ hasNotifications }) {  
  return <div>{hasNotifications && <p>You have new notifications!</p>}</div>;  
}
```

- If has Notifications is true: You have new notifications!
- If has Notifications is false: Nothing is rendered.

4. Using Switch Case Statements

Switch case statements are useful when you need to handle multiple conditions, which would otherwise require multiple if conditions. This approach can be more readable if there are many conditions to check.

```
function StatusMessage({ status }) {  
  switch (status) {  
    case 'loading':  
      return <p>Loading...</p>;  
    case 'success':  
      return <p>Data loaded successfully!</p>;  
    case 'error':  
      return <p>Error loading data.</p>;  
    default:  
      return <p>Unknown status</p>;  
  }  
}
```

- If status is 'loading': Loading...
- If status is 'success': Data loaded successfully!
- If status is 'error': Error loading data.

5. Conditional Rendering in Lists (Using .map())

Conditional rendering can be helpful when rendering lists of items conditionally. You can filter or map over an array to selectively render components based on a condition.

```
const items = ["Apple", "Banana", "Cherry"];  
const fruitList = items.map((item, index) =>  
  item.includes("a") ? <p key={index}>{item}</p> : null  
);
```

If the item contains letter "a", it will be displayed.

6. Conditional Rendering with Component State

You can also manage conditional rendering based on the component's state. For example, you can show a loading spinner until some data is fetched, and then display the actual content once the data is ready.

Syntax:

```
if (loading) {  
  // Render Loading Component  
} else {  
  // Render Data Component  
}
```

React Lists

React Lists are used to **display** a collection of similar **data** items like an array of objects and menu items. It allows us to dynamically render the array elements and display **repetitive data**.

Rendering List in React

To render a list in React, we will use the JavaScript array `map()` function. We will iterate the array using `map()` and return the required element in the form of JSX to render the repetitive elements.

Example:

```
import React from 'react';
function ItemList() {
  const items = ['Apple', 'Banana', 'Mango', 'Orange'];
  return (
    <div>
      <h2>Fruit List</h2>
      <ul>
        {items.map((fruit, index) => (
          <li key={index}>{fruit}</li>
        ))}
      </ul>
    </div>
  );
}
export default ItemList;
```

Why 'key' Prop is Important in React Lists

The key prop is essential in React when rendering lists because it helps to identify which items have changed, been added, or removed. This allows React to efficiently update and re-render only the changed items in the DOM, rather than re-rendering the entire list.

List with Objects

Lists can also be created using objects where each item contains multiple properties.

```
const users = [
  { id: 1, name: 'Pooja', age: 30 },
  { id: 2, name: 'Nisha', age: 25 },
  { id: 3, name: 'Aarana', age: 20 },
];
const App = () => {
  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>
          {user.name} is {user.age} years old.
        </li>
      ))}
    </ul>
  );
}
export default App;
```

List with a Click Event

A "List with a Click Event" in React typically refers to displaying a list of items (such as an array) on the screen, where each item is associated with an event handler (such as onClick) that executes a function when the item is clicked.

```
const App = () => {
  const COMPANY = ["POOJA", "AARANA", "DRASHTI"];
  const handleClick = (COMPANY) => {
    alert(`You clicked on ${COMPANY}`);
  };
  return (
```

```

    <ul>
      {COMPANY.map((COMPANY, index) => (
        <button key={index} onClick={() => handleClick(COMPANY)}>
          {COMPANY}
        </button>
      ))}
    </ul>
  );
};
export default App;

```

React Events

In React, events are actions that occur within an application, such as **clicking a button**, **typing in a text field**, or **moving the mouse**. React provides an efficient way to handle these actions using its event system. Event handlers like `onClick`, `onChange`, and `onSubmit` are used to capture and respond to these events.

An event is an action that could be triggered as a result of the user action or system generated event. For example, a mouse click, loading of a web page, pressing a key, window resizes, and other interactions are called events.

React has its own event handling system which is very similar to handling events on DOM elements. The react event handling system is known as Synthetic Events. The synthetic event is a cross-browser wrapper of the browser's native event.

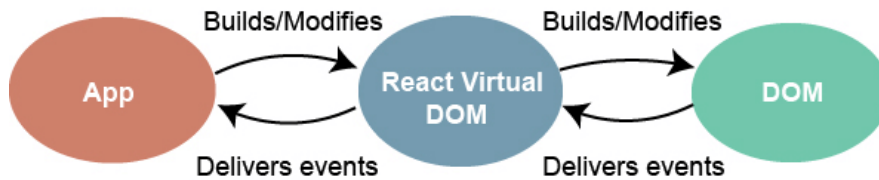
Syntax

```
<element onEvent={handlerFunction} />
```

- **element:** The JSX element where the event is triggered (e.g., `<button>`, `<input>`, etc.).
- **onEvent:** The event name in camelCase (e.g., `onClick`, `onChange`).
- **handlerFunction:** The function that handles the event when it occurs.

Commonly Used React Event Handlers

React Event	Description
<u>onClick</u>	This event is used to detect mouse clicks in the user interface.
<u>onChange</u>	This event is used to detect a change in the input field in the user interface.
<u>onSubmit</u>	This event fires on the submission of a form in the user interface and is also used to prevent the default behavior of the form.
<u>onKeyDown</u>	This event occurs when the user press any key from the keyboard.
<u>onKeyUp</u>	This event occurs when the user releases any key from the keyboard.
<u>onMouseEnter</u>	This event occurs when the ouse enters the boundary of the element

Handling Events in React**Events Handler****1. Adding Event Handlers**

In React, event handlers are added directly to elements via JSX attributes. React uses the camelCase convention for event names, which differs from the standard HTML event names (e.g., onClick instead of onclick).

Example : ClickEventExample.js

```

import React from 'react';
const ClickEventExample = () => {
  const handleClick = () => {
    alert('Button clicked!');
  };
  return (
    <div>
      <h2>Click Event</h2>
      <button onClick={handleClick}>Click Me</button>
    </div>
  );
};
export default ClickEventExample;

```

2. Reading Props in Event Handlers

In React, event handlers often need access to the props passed from parent components. This is useful when the event handler needs to perform actions based on the data provided via props.

Example: EventObjectExample.js

```

import React from 'react';
const EventObjectExample = () => {
  const handleChange = (event) => {
    console.log('Input value:', event.target.value);
  };
  return (
    <div>
      <h2>Event Object Example</h2>
      <input type="text" onChange={handleChange} placeholder="Type something..." />
    </div>
  );
};

export default EventObjectExample;

```

3. Passing Event Handlers as Props

Event handlers can be passed down to child components as props. This allows child components to communicate back to the parent and trigger actions in the parent when events occur.

Example: EventWithArgument.js

```
import React from 'react';
const EventWithArgument = () => {
  const greetUser = (name) => {
    alert(`Hello, ${name}!`);
  };
  return (
    <div>
      <h2>Passing Arguments</h2>
      <button onClick={() => greetUser('Pooja')}>Greet</button>
    </div>
  );
};
export default EventWithArgument;
```

Forms: (Adding forms, Handling forms, Submitting forms)

Forms are an essential part of any application used for collecting user data, processing payments, or handling authentication. **React Forms** are the components **used to collect and manage the user inputs**. These components include the input elements like text field, check box, date input, dropdowns etc.

React Forms :

React forms are the way to collect user data in a React application. React typically uses controlled components to manage form state and handle user input changes efficiently. It provides additional functionality such as preventing the default behavior of the form which refreshes the browser after the form is submitted.

1. Adding Forms

This means **creating input fields** like text boxes, dropdowns, text areas, radio buttons, etc., and displaying them in your React component.

Example:

```
<form>
  <label>Name:</label><br />
  <input type="text" /><br />

  <label>Message:</label><br />
  <textarea rows="4" cols="30" /><br />

  <label>City:</label><br />
  <select>
    <option>Ahmedabad</option>
    <option>Surat</option>
```

```
    </select><br />
    <button type="submit">Submit</button>
  </form>
```

2. Handling Forms

React handles form input using **state (useState)**. When the user types or selects something, we update the state using the onChange event.

Example:

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

function HandlingForm() {
  const [name, setName] = useState('');
  const [city, setCity] = useState('');

  return (
    <form>
      <label>Name:</label><br />
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
      /><br />

      <label>City:</label><br />
      <select
        value={city}
        onChange={(e) => setCity(e.target.value)}
      >
        <option value="">--Select City--</option>
        <option value="Ahmedabad">Ahmedabad</option>
        <option value="Surat">Surat</option>
      </select><br />

      <p>You typed: {name}</p>
      <p>Selected city: {city}</p>
    </form>
  );
}
```

3. Submitting Forms

React uses the onSubmit event to handle form submission. We usually prevent the default browser behavior using e.preventDefault() and do something like show an alert, log the values, or send them to a server.

Example:

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

function SubmittingForm() {
  const [formData, setFormData] = useState({
```

```

    name: '',
    city: '',
    message: ''
  });

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData((prev) => ({
      ...prev,
      [name]: value
    }));
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    alert(`Submitted:\nName: ${formData.name}\nCity: ${formData.city}\nMessage:
    ${formData.message}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>Name:</label><br />
      <input name="name" value={formData.name} onChange={handleChange} /><br />

      <label>City:</label><br />
      <select name="city" value={formData.city} onChange={handleChange}>
        <option value="">--Select City--</option>
        <option value="Ahmedabad">Ahmedabad</option>
        <option value="Surat">Surat</option>
      </select><br />

      <label>Message:</label><br />
      <textarea
        name="message"
        value={formData.message}
        onChange={handleChange}
        rows="4"
        cols="30"
      /><br />
      <button type="submit">Submit</button>
    </form>
  );
}

```

event.target.name and event.Target.event, React Memo

In React, when handling events like onChange for input elements, the event object (e or event) passed to the event handler provides information about the event.

- **event.target:**

This property refers to the DOM element that triggered the event. For example, if a user types into an <input> field, event.target would be that specific <input> element.

1. event.target.name

- event → the event object passed to the event handler.
- event.target → the HTML element that triggered the event.
- event.target.name → gives the value of the element's **name attribute**.
- Commonly used in forms to identify which input field triggered the change without writing separate handlers for each input.

Example: Myform.js

```
function MyForm() {
  const handleChange = (event) => {
    console.log("Name:", event.target.name);
    console.log("Value:", event.target.value);
  };
  return (
    <>
      <input name="username" onChange={handleChange} placeholder="Username" />
      <input name="email" onChange={handleChange} placeholder="Email" />
    </>
  );
}
```

2. event.target.event

- This is **not valid** in JavaScript or React.
- There is **no event property** inside event.target.
- Accessing it will return undefined.

Example:

```
console.log(event.target.event);
```

3. React.memo

- A **higher-order component** in React.
- Wraps a functional component to **prevent unnecessary re-renders**.
- Works by doing a **shallow comparison** of props — if props are the same as the previous render, React skips rendering that component again.
- Useful for **performance optimization**, especially in large applications where components receive the same props repeatedly.
- Similar in concept to PureComponent for class components.

Example:

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

const Child = React.memo(({ name }) => {
  console.log("Child rendered!");
  return <h2>Hello {name}</h2>;
});

export default function App() {
```

```

const [count, setCount] = useState(0);
return (
  <div>
    <Child name="Pooja" />
    <p>Count: {count}</p>
    <button onClick={() => setCount(count + 1)}>Increase</button>
  </div>
);
}

```

Components (TextArea, Drop down list (SELECT))

Multiple Input Fields

React Forms allow to handle multiple inputs in a single form. Other types of input fields present in Forms are

- Textarea tag
- Select tag

1. Textarea

The textarea tag defines the element by its children i.e., enclosed in the tags. In React we use the value prop instead.

Syntax:

```
<textarea value={text} onChange={handleChange} />
```

Here,

- **text:** refers to the state variable in which the value is stored
- **handleChange:** is the function to be executed to update the state when the onChange event triggers.

Select:

- The select tag defines the element by its children i.e., enclosed in the tags. In React similar to text.area we use the value prop instead.

Syntax:

```

<select value={this.state.value} onChange={this.handleChange}>
  <option value="HTML">HTML</option>
  <option value="CSS">CSS</option>
  <option value="JavaScript">JavaScript</option>
  <option value="React">React</option>
</select>

```

Here,

- **this.state.value:** refers to the state variable in which the value is stored
- **this.handleChange:** is the function to be executed to update the state when the onChange event triggers.

Hooks: Concepts and Advantages

React Hooks

ReactJS Hooks, introduced in React 16.8, are among the most impactful updates to the library; with over 80% of modern React **projects** adopting them for state and lifecycle management. They let developers use state, side effects, and other React features without writing class components. Hooks **streamline code, improve readability, and promote a functional programming style**, making functional components as powerful—if not more—than their class-based counterparts.



Hooks are features that React provides us if we want to make functional components while creating a React web app. These features are alternatives to a few lifecycle methods like `componentDidMount()`, `componentDidUpdate()`, apart from this it gives us more flexibility to write maintainable code.

Advantages of Using Hooks:

- **Simplicity and Readability:** Hooks simplify the syntax and structure of functional components, making them more concise and easier to read. This is especially beneficial for developers who are new to React.
- **Code Reusability:** Hooks encourage the creation of reusable logic through custom hooks. This modularity allows users to share and reuse functionalities across different components.
- **Improved State Management:** Hooks, especially `useState`, provide a more easy and declarative way to manage state in functional components compared to the class component's `this.setState` method.
- **Better Handling of Side Effects:** `useEffect` simplifies the management of side effects, such as data fetching, subscriptions, or manual DOM manipulations. It combines lifecycle methods into a single, more flexible function.
- **No Need for Classes:** Hooks eliminate the need for class components, allowing users to use the functional component syntax exclusively. This aligns with the trend of writing more functional and less class-based code.
- **Easier to Extract Logic:** Hooks make it easier to extract and reuse logic from components. Logic can be organized into custom hooks, promoting cleaner and more maintainable code.

Disadvantages of Using Hooks:

- **Learning Curve:** Users familiar with class components may face a learning curve when transitioning to hooks. Understanding the various hooks and their use cases requires some time and practice.
- **Compatibility Issues:** Hooks were introduced in React 16.8, and some existing codebases may still heavily rely on class components. Migrating older projects to hooks might require effort and careful consideration.
- **Backward Compatibility:** Hooks are not backward compatible with class components. This means that existing class-based code cannot seamlessly use hooks without a significant refactor.
- **Lack of Implicit Binding:** Unlike class components, hooks do not have implicit binding of methods, which might be advantageous in certain scenarios. Users need to be explicit about function binding, especially when passing functions as props.

Types of React Hooks

React offers various hooks to handle state, side effects, and other functionalities in functional components.

- **State management:** Hooks like `useState` and `useReducer` let functional components store and update values (e.g., counters, form inputs).
- **Side effects:** `useEffect` allows components to perform tasks like fetching data, updating the DOM, or subscribing to services.
- **Context access:** `useContext` lets components read values directly from React's Context API without prop drilling.
- **Refs:** `useRef` stores mutable values (like DOM nodes or timers) that don't trigger re-renders.
- **Performance optimization:** Hooks like `useMemo` and `useCallback` help avoid unnecessary recalculations and re-renders.

Commonly Used Hooks

Hook	Purpose	Example Use Case
<code>useState</code>	Manage local state	Counter, form input
<code>useEffect</code>	Handle side effects	API calls, DOM updates
<code>useContext</code>	Access global values	Theme, authentication
<code>useRef</code>	Store mutable values	Focus input, timer ID
<code>useReducer</code>	Complex state logic	Shopping cart, multi-step forms
<code>useMemo</code>	Memoize calculations	Expensive math operations
<code>useCallback</code>	Memoize functions	Prevent re-render of child components

Why Hooks Matter

- **Simplified logic:** No need for verbose class components.
- **Reusability:** Extract logic into **custom hooks** for reuse.
- **Cleaner code:** Encourages functional programming style.
- **Better testing:** Functional components with hooks are easier to test.

Rules of Hooks

- **Call hooks only at the top level** (not inside loops or conditions).
- **Call hooks only in React functions** (functional components or custom hooks).
- **Follow the order consistently** so React can track state correctly.

Below are some of the most commonly used types of React hooks:

1. useState

- Manages state in a function component. Returns a **state variable** and a **function** to update it.

Syntax:

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

- **state:** The current value of the state.
- **setState:** A function used to update the state.
- **initialState:** The initial value of the state, which can be a primitive type or an object/array

Example:

```
import React, { useState } from "react";
function Counter() {
  const [count, setCount] = useState(0); // initial value = 0
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}
```

2. useReducer

- Alternative to useState for **complex state logic**. Uses a reducer function like in Redux.

Syntax:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

- **state:** The current state value.
- **dispatch:** A function used to dispatch actions that will update the state.
- **reducer:** A function that defines how the state should change based on the dispatched action.
- **initialState:** The initial state value.

Example:

```
import React, { useReducer } from "react";
function reducer(state, action) {
  switch (action.type) {
    case "inc": return { count: state.count + 1 };
    case "dec": return { count: state.count - 1 };
    default: return state;
  }
}
```

```

    }
  }
  function Counter() {
    const [state, dispatch] = useReducer(reducer, { count: 0 });
    return (
      <>
        <p>{state.count}</p>
        <button onClick={() => dispatch({ type: "inc" })}>+</button>
        <button onClick={() => dispatch({ type: "dec" })}>-</button>
      </>
    );
  }
}

```

3. useEffect

- Runs **side effects** (data fetching, timers, DOM updates).
 - Runs after render by default.
 - Can have dependencies to control when it runs.

Example:

```

import React, { useState, useEffect } from "react";
function App() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `Clicked ${count} times`;
  }, [count]); // runs when count changes
  return (
    <button onClick={() => setCount(count + 1)}>Click</button>
  );
}

```

4. useContext

- Lets you use **React Context** without prop drilling (passing props manually down many components).

Example:

```

import React, { useContext } from "react";
const ThemeContext = React.createContext("light");
function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button>{theme} mode</button>;
}

```

5. useRef

- Stores a **mutable value** that does **not** trigger re-render when changed. Often used to access DOM elements directly.

Example:

```

import React, { useRef } from "react";

```

```
function InputFocus() {
  const inputRef = useRef(null);
  const focusInput = () => {
    inputRef.current.focus();
  };
  return (
    <>
      <input ref={inputRef} type="text" />
      <button onClick={focusInput}>Focus</button>
    </>
  );
}
```

6. useMemo

- Memoizes (caches) computed values to avoid re-calculation unless dependencies change. Improves performance in expensive calculations.

Example:

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

function ExpensiveCalc({ num }) {
  const result = useMemo(() => {
    console.log("Calculating...");
    return num * 2;
  }, [num]);

  return <p>Result: {result}</p>;
}
```

7. useCallback

- Memoizes **functions** so they are not recreated on every render. Useful for preventing unnecessary re-renders in child components.

Example:

```
import React, { useState, useCallback } from "react";
function App() {
  const [count, setCount] = useState(0);

  const increment = useCallback(() => setCount(c => c + 1), []);
  return <button onClick={increment}>{count}</button>;
}
```

8. useEffect

- Same as useEffect, but runs **synchronously** after DOM updates (before browser paints). Useful for layout measurements.

Example:

```
import React, { useEffect, useRef } from "react";
function Box() {
  const boxRef = useRef(null);

  useEffect(() => {
    console.log(boxRef.current.getBoundingClientRect());
  }, []);

  return <div ref={boxRef} style={{ width: 100, height: 100, background: "red" }} />;
}
```

Hook: Building custom hook, advantages and use

What is a Custom Hook?

- A **custom hook** is simply a **JavaScript function** whose name starts with "use" and that can use other React hooks (useState, useEffect, etc.) inside it.
- It's a way to **extract and reuse logic** between components.

"If you have some repeated state + effect logic in multiple components, move it into a custom hook."

When to Use a Custom Hook

- When **two or more components** share the same logic (like fetching data, handling forms, timers, and authentication).
- When you want **cleaner, more readable code** by separating UI from logic.

Advantages

1. **Reusability** – Write once, use in multiple components.
2. **Cleaner Components** – UI code and logic stay separate.
3. **Maintainability** – Logic changes only in one place.
4. **Testability** – Easier to test logic separately.
5. **Performance** – Reduces code duplication and possible mistakes.

Example: Custom Hook for Window Width

We'll create a hook called useWindowWidth that returns the current browser window width.

Custom Hook (useWindowWidth.js)

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

function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    const handleResize = () => setWidth(window.innerWidth);

    window.addEventListener("resize", handleResize);
    return () => window.removeEventListener("resize", handleResize);
  });
}
```

```
    }, []));

    return width; // return state value
}

export default useWindowWidth;
```

Using the Custom Hook in a Component

```
import React from "react";
import useWindowWidth from "../useWindowWidth";

function App() {
  const width = useWindowWidth();

  return (
    <div>
      <h1>Window Width: {width}px</h1>
    </div>
  );
}
export default App;
```

Example: 2 useToggle.js

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

// Custom Hook with name: useToggle
function useToggle(initialValue = false) {
  const [value, setValue] = useState(initialValue);
  const toggle = () => setValue(!value);
  return [value, toggle];
}
export default useToggle;
```

TogglePage.js

```
import React from "react";
import useToggle from "../useToggle"; // import the custom hook
function TogglePage() {
  const [isOn, toggleIsOn] = useToggle(false);
  return (
    <div>
      <h2>Toggle Example Page</h2>
      <p>Status: {isOn ? "ON" : "OFF"}</p>
      <button onClick={toggleIsOn}>Toggle</button>
    </div>
  );
}
export default TogglePage;
```

App.js

```
import React from "react";
import TogglePage from "./TogglePage";
function App() {
  return (
    <div>
      <h1>My React App</h1>
      <TogglePage />
    </div>
  );
}
export default App;
```