React

1. React is a JavaScript library for building user interfaces.
2. React is used to build single-page applications.
3. React allows us to create reusable UI components.
4. React is a component base architecture

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Hello(props) {
      return <h1> Hello World! </h1>;
}
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render (<Hello />);
```

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 finds out what changes have been made, and changes **only** what needs to be changed.

Install React Project:

```
npx create-react-app my-react-app
```

React Features

1. JSX
2. Components
3. One-way Data Binding
4. Virtual DOM
5. Simplicity
6. Performance

React Directly in HTML

```
<!DOCTYPE html>
<html>
 <head>
        <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script>
        <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>
        <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
 </head>
 <body>
        <div id="mydiv"> </div>
        <script type="text/babel">
                function Hello() {
```

```
                return <h1> Hello World! </h1>;
            }
             ReactDOM.render(<Hello />, document.getElementById('mydiv'))
        </script>
 </body>
</html>
```

Introducing JSX

1. JSX stands for JavaScript XML.
2. JSX allows us to write HTML in React.
3. JSX makes it easier to write and add HTML in React.
4. With JSX you can write expressions inside curly braces { }.

```
const myElement =  <h1> React is {5 + 5} times better with JSX </h1>;
```

5. inserting a Large Block of HTML

```
const myElement = (
    <ul>
        <li>Apples</li>
        <li>Bananas</li>
        <li>Cherries</li>
    </ul>
);
```

6. Use ternary expressions instead:

```
const myElement = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1>;
```

React Render

1. React's goal is in many ways to render HTML in a web page.
2. React renders HTML to the web page by using a function called **ReactDOM.render()**

```
<body>
    <div id="root"></div>
</body>
ReactDOM.render( <p>Hello</p> ,  document.getElementById('root') );
```

React Components

Earlier, the developers write more than thousands of lines of code for developing a single page application. These applications follow the traditional DOM structure, and making changes in them was a very challenging task. If any mistake found, it manually searches the entire application and update accordingly. The component-based approach was introduced to overcome an issue. In this approach, the entire application is divided into a small logical group of code, which is known as components. Components are like functions that return HTML elements.

A React component represents a small chunk of user interface in a webpage

Components come in two types,

1. Class components
2. Function components

Functional Components

function components are a way to write components that only contain a render method and don't have their own state. They are simply JavaScript functions that may or may not receive data as parameters. The functional component is also known as a stateless component because they do not hold or manage state.

```
function WelcomeMessage(props) {
 return <h1>Welcome to the , {props.name}</h1>;
}
```

Class Components

- When we used **class** as a component it called class component.
- The component must include the extends **React.Component** statement, this statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.
- The component also requires a **render()** method, this method returns HTML.
- The class component is also known as a stateful component because they can hold or manage local state.

```
class MyComponent extends React.Component {
        render() {
                return (
                  <div> This is main component. </div>
                );
        }
}
```

Class Component State

- React Class components have a built-in **state** object. The state object is where you store property values that belongs to the component.

```
class Car extends React.Component {
        state = {
                        brand: "Ford",
                        model : 123,
```

```
        };

    render () {
        return (
          <h1> My Car</h1>
        );
    }
}
```

- Refer to the state object anywhere in the component by using the **this.state.propertyname**

```
class Car extends React.Component {
    state = {
        brand: "Ford",
        model: "Mustang",
    };
    render () {
        return (
            <h1> { this.state.brand } </h1>
            <p> { this.state.model } </p>
        );
    }
}
```

- **Changing the state Object**

1. To change a value in the state object, use the **this.setState(**) method.
2. When a value in the state object changes, the component will re-render, meaning that the output will change according to the new value(s).

```
class Car extends React.Component {
    state = {
        brand: "Ford",
        model: "Mustang",
        color: "red",
        year: 1964
    };
    changeColor = () => {
        this.setState( {color: "blue"} );
    }
    render() {
        return (
          <div>
            <h1> My { this.state.brand } </h1>
            <button   onClick={ this.changeColor } > Change color </button>
          </div>
        );
```

```
        }
    }
```

Syntex:

```
setState(updater , [callback])

this.setState((state, props) => {
        return {counter: state.counter + props.step};
});
```

If want pass a function in setState method, this function takes two arguments as parameter.

1. state: this take previous state
2. props: previous props

Class Constructor

- The constructor is a method used to initialize an object's state in a class. It automatically called during the creation of an object in a class.
- The constructor in a React component is called before the component is mounted.
- When you implement the constructor for a React component, you need to call **super(props)** method before any other statement.
- If you do not call super(props) method, this.props will be undefined in the constructor and can lead to bugs.

Syntax:

```
Constructor(props) {
        super(props);
}
```

Constructor and state:

- You cannot call setState() method directly in the **constructor().**
- If the component needs to use local state, you need directly to use **'this.state'** to assign the initial state in the constructor.
- The constructor only uses this.state to assign initial state, and all other methods need to use set.state() method.

```
class App extends Component {
    constructor(props){
        super(props);
        this.state = {
            data: 'www.javatpoint.com'
        }
```

```
            this.handleEvent = this.handleEvent.bind(this);
        }
        handle(){
            console.log( this.props );
        }
        render() {
            return (
                <div className="App">
                <input type ="text" value={this.state.data} />
                <button onClick={ this.handle }> Please Click </button>
                </div>
            );
        }
}
export default App;
```

React Component Life-Cycle

Each component has several "lifecycle methods" that you can override to run code at times in the process

1. Mounting Phase
2. Updating Phase
3. Unmounting Phase

Mounting Phase

constructor():

1. The constructor for a React component is called before it is mounted.
2. If you don't initialize state and you don't bind methods, you don't need to implement a constructor for your React component.

Syntax:

```
constructor(props)
```

```
constructor(props) {
    super(props);
    // Don't call this.setState() here!
    this.state = { counter: 0 };
    this.handleClick = this.handleClick.bind(this);
}
```

componentWillMount():

1. This is invoked immediately before a component gets rendered into the DOM.
2. In the case, when you call setState() inside this method, the component will not re-render.

```
class App extends React.Component {
        constructor(props) {
                super(props);
                this.state = { name: "shuvo"};
        }
        componentWillMount() {
                console.log('Component Will MOUNT!')
        }
        render() {
                return (
                        <h1> { this.state.name} </h1>
                    );
            }
}
```

componentDidMount()

1. The componentDidMount() method is called after the component is rendered.
2. This is where you run statements that requires that the component is already placed in the DOM.
3. If you need to load data from a remote endpoint, this is a good place to instantiate the network request.
4. This method is a good place to set up any subscriptions. If you do that, don't forget to unsubscribe in `componentWillUnmount()`

```
class App extends React.Component {
        constructor(props) {
                super(props);
                this.state = { name: "shuvo"};
        }
        componentDidMount() {
                console.log('Component Did MOUNT!')
        }
        render() {
                return (
                        <h1> { this.state.name} </h1>
                    );
            }
}
```

render():

1. The `render` method will be called each time an update happens
2. If you don't want to render anything, you can return a null or false value.

```
class App extends React.Component {
        render() {
                return (
                        <h1> I am render method  </h1>
                    );
            }
```

}

Updating Phase

1. This phase also allows to handle user interaction and provide communication with the component's hierarchy.
2. A component is updated whenever there is a change in the component's state or props.
3. The main aim of this phase is to ensure that the component is displaying the latest version of itself.

componentWillRecieveProps()

It is invoked when a component receives new props

shouldComponentUpdate()

It is invoked when a component decides any changes/updation to the DOM.

componentWillUpdate()

1. it is invoked just before rendering when new props or state are being received.
2. Use this as an opportunity to perform preparation before an update occurs.
3. This method is not called for the initial render.

render()

It is invoked to examine this.props and this.state and return one of the following types: React elements, Arrays and fragments, Booleans or null, String and Number. If shouldComponentUpdate () returns false, the code inside render () will be invoked again to ensure that the component displays itself properly.

componentDidUpdate()

1. componentDidUpdate() is invoked immediately after updating occurs.
2. This method is not called for the initial render.

Syntex:

```
componentDidUpdate(prevProps, prevState, snapshot)
```

preprops: it takes previous props of a component
prestate : take previous state of component .

```
class App extends React.Component {
        constructor(props) {
                super(props);
                this.state = { name: "shuvo"};
        }
        componentDidUpdate(prevProps, prevState) {
                console.log('Component Did UPDATE!')
        }
        render() {
                return (
```

```
                    <h1> { this.state.name} </h1>
                );
            }
        }
```

Unmounting Phase

componentWillUnmount()

1. ComponentWillUnmount() is invoked immediately before a component is unmounted and destroyed.
2. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in componentDidMount()

```
class App extends React.Component {
        constructor(props) {
                super(props);
                this.state = {name: "shuvo"};
        }
        componentWillUnmount() {
                console.log('Component Will UNMOUNT!')
        }

        render() {
                return (
                  <div>
                    <h1>ReactJS component's Lifecycle</h1>
                    <h3>Hello {this.state.hello}</h3>
                    <button onClick = {this.changeState}>Click Here!</button>
                  </div>
                );
        }
}
export default App;
```

React Props

1. **Props** stand for "Properties." They are read-only components.
2. It is an object which stores the value of attributes of a tag and work like the HTML attributes.
3. it gives a way to pass data from one component to other components.
4. It is like function arguments.
5. Props are passed to the component in the same way as arguments passed in a function.
6. Props are immutable so we cannot modify the props from inside the component.

Props in function:

```
function Car(props) {
        return <h2> I am a { props.brand } </h2>;
}

const myElement = <Car brand="Ford" />;
```

Props in class:

In class, Props use by **this.props.property**

```
class App extends React.Component {
    render() {
        return (
            <h1> Welcome to { this.props.name } </h1>
        );
    }
}
export default App;

const main = <App  name="shuvo"  />
```

React 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 clicks, 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 even

1. React events are named as camel Case instead of lowercase.
2. With JSX, a function is passed as the event handler instead of a string.
3. React event handlers are written inside curly braces:

```
function Football() {
    const shoot = () => {
        alert("Great Shot!");
    }

    return (
        <button onClick={shoot}>Take the shot!</button>
    );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
```

```
root.render(<Football />);
```

4. In react, we cannot return false to prevent the default behavior. We must call **preventDefault** event explicitly to prevent the default behavior. For example:

```
function ActionLink() {
        function handleClick(e) {
                e.preventDefault();
                console.log('You had clicked a Link.');
        }
        return (
                <a href="#" onClick={handleClick}> Click_Me    </a>
        );
}
```

5. To pass an **argument** to an event handler, use an arrow function.

```
function Football() {
        const shoot = (a) => {
                alert(a);
        }

        return (
                <button onClick={ () => shoot("Goal!")}> shot </button>
        );
}

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

6. Event handlers have access to the React **event** that triggered the function.

```
function Football() {
        const shoot = (a, b) => {
                alert(b.type);
        }

        return (
          <button onClick={(event) => shoot("Goal!", event)}> Take the shot </button>
        );
}
```

This problem in event handle function:

when we want to set a value in setState() method in event function, react face a problem for **reference** value of this.

```
class App extends React.Component {
   constructor(props) {
```

```
            super(props);
            this.state = {
                count : 1
            };
    }
    handle(event) {
            this.setState({
              count : count +1
            });
    }
    render () {
      return (
            <button type="text"  onClick ={ this.handle } >   click </button>

      );
    }
}
export default App;
```

The way to solve this problem –

1. arrow function:

If we convert the function in **arrow function**, then reference value this not create a problem.

```
class App extends React.Component {
    constructor(props) {
            super(props);
            this.state = {
                count : 1
            };
    }
    handle = () => {
            this.setState({
              count : count +1
            });
    }
    render () {
      return (
            <button type="text"  onClick ={ this.handle }>   click </button>
      );
    }
}
export default App;
```

2. Bind the function in construction function. Then this not create problem.

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

```
constructor(props) {
        super(props);
        this.state = {
            count : 1,
            this.handle = this.handle.bind(this)
        };
}
handle =  {
        this.setState({
           count : count +1
        });
}
render () {
        return (
             <button type="text"  onClick ={ this.handle } >  click <button>
        );
}
}
export default App;
```

3. bind in inline event:

```
handle = (param) {
    this.setState({
        console.log(param)
    });
}
<button type="text" onClick = { this.handle.bind( this ,param) } >  click </button>
```

Here bind function in inline function with **this.fucntion_name.bind(this, param).** We can pass argument as second parameter.

```
class App extends React.Component {
  constructor(props) {
        super(props);
        this.state = {
            count : 1,
        };
  }
  handle =( num )  {
        this.setState({
           count : count + num
        });
  }
  render () {
    return (
        <button type="text"  onClick = { this.handle.bind( this , 2 ) } >  click </button>
    );
```

```
  }
}
export default App;
```

React Conditional Rendering

In React, we can create multiple components which encapsulate behavior that we need. After that, we can render them depending on some conditions or the state of our application. In other words, based on one or several conditions, a component decides which elements it will return.
There is more than one way to do conditional rendering in React. They are given below.

1. if
2. ternary operator
3. logical && operator
4. switch case operator

if Statement

```
function Goal(props) {
  const isGoal = props.isGoal;
  if (isGoal) {
            return <MadeGoal/>;
  }
            return <MissedGoal/>;
}
```

Logical && operator

This operator is used for checking the condition. If the condition is true, it will return the element right after &&, and if it is false, React will ignore and skip it. function Garage(props) {

```
  const cars = props.cars;
        return (
          <>
            <h1>Garage</h1>
            { cars.length > 0    &&   <h2>  shuvo </h2>   }
          </>
        );
}

const cars = ['Ford', 'BMW', 'Audi'];
<Garage  cars={cars}   />
```

Ternary Operator

The ternary operator is used in cases where two blocks alternate given a certain condition. This operator makes your if-else statement more concise. It takes three operands and used as a shortcut for the if statement.

Syntax:

```
condition ?  true : false
```

```
function Goal(props) {
        const isGoal = props.isGoal ;
        return (
          <>
                { isGoal ?  <MadeGoal/>  :  <MissedGoal  /> }
          </>
        );
}
root.render(  <Goal  isGoal= { false }  />  );
```

Switch case operator

```
function NotificationMsg({ text}) {
    switch(text) {
        case 'Hi All':
            return <Message: text={text} />;
        case 'Hello JavaTpoint':
            return <Message text={text} />;
        default:
            return null;
    }
}
```

Preventing Component form Rendering

Sometimes it might happen that a component hides itself even though another component rendered it. To do this (prevent a component from rendering), we will have to return null instead of its render output. It can be understood in the below example:

```
function Show(props)
{
        If ( !props.displayMessage )
            return null;
        else
            return <h3>Component is rendered</h3>;
}
<Show displayMessage = {true}  />
```

React Lists

Lists are used to display data in an ordered format and mainly used to display menus on websites. In React, Lists can be created in a similar way as we create lists in JavaScript.

The map() function is used for traversing the lists.

```
var numbers = [1, 2, 3, 4, 5];
const multiplyNums = numbers.map(   (number)   =>  {
            return (number * 5);
});
console.log(multiplyNums);
```

map() in react :

```
const myList = ['Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa'];
const listItems = myList.map(    (myList) => {
            return <li> { myList } </li>;
});

 <ul> { listItems } </ul>,
```

React Keys

A key is a unique identifier. In React, it is used to identify which items have changed, updated, or deleted from the Lists. It is useful when we dynamically created components or when the users alter the lists. Keys should be given inside the array to give the elements a stable identity.

```
const stringLists = [ 'Peter', 'Sachin', 'Kevin', 'Dhoni', 'Alisa' ];
const updatedLists = stringLists.map(  (strList)  =>  {
            <li  key = { strList.id } > { strList } </li>;
});
function Car(props) {
          return <li>  I am a { props.brand }    </li>;
}

function Garage() {
     const cars = [
            {id: 1, brand: 'Ford'},
            {id: 2, brand: 'BMW'},
            {id: 3, brand: 'Audi'}
        ];
```

```
        return (
            <ul>  { cars.map(   (car) => <Car key={car.id} brand = {car.brand}  /> )  } </ul>
             );
}

< Garage   />
```

Form

In HTML, form elements such as `<input>`, `<textarea>`, and `<select>` typically maintain their own state and update it based on user input. But in react , form element also maintains by react. So that when both browser and react want to maintain the form element, there will be conflict. So that there are two way maintain the form element.

1. Controlled component
2. Uncontrolled component

Controlled Component

When form element whose value is controlled by React in this way is called a “controlled component”. When the data is handled by the components, all the data is stored in the component state.

Value:

When we use value attribute in form element react know that this is controlled component. It we controlled by react.

```
<input type="text"  value={ this.state.value }  onChange={this.handleChange} />
```

Change value:

In react, all DOM event handle by it. So, if we do not do any event, it takes form element as read only mode. In react all value are change by state. if we want to change form element value, we want to change the state value.

Form in react:

```
export default class From extends Component {
        state = {
                input: "hello",
                text: "text",
                select: "",
                checkbox: true,
        };

        handle = (e) => {
```

```
                if (e.target.type === "text") {
                        this.setState({
                          input: e.target.value,
                        });
                } else if (e.target.type === "textarea") {
                        this.setState({
                          text: e.target.value,
                        });
                } else if (e.target.type === "select-one") {
                        this.setState({
                          select: e.target.value,
                        });
                } else if (e.target.type === "checkbox") {
                        this.setState({
                          checkbox: e.target.value,
                        });
                } else {
                        console.log("lfk");
                }
        };

        render() {
          const { input, text, select, checkbox } = this.state;
          return (
            <form>
              <input type="text “  value={input}  onChange={this.handle}   />

              <h1> TEXT AREA </h1>
              <textarea name="text" value={text} onChange={this.handle} />

              <h1> Select BOX </h1>

              <select value={select} onChange={this.handle}>
                        <option value="1">name</option>
                        <option value="2">roll</option>
              </select>

              <h1> check BOX </h1>
              <input type="checkbox" value={checkbox} onChange={this.handle} />
            </form>
          );
        }
}
```

Submitting Forms

You can control the submit action by adding an event handler in the **onSubmit** attribute for the <form>:

```
function MyForm() {
        const [name, setName] = useState("");

        const handleSubmit = (event) => {
                event.preventDefault();
                alert(`The name you entered was: ${name}`)
        }

        return (
         <form onSubmit={handleSubmit}>
                 <label>Enter your name:
                  <input
                    type="text"
                    value={name}
                    onChange={(e) => setName(e.target.value)}
                   />
                 </label>
                 <input type="submit" />
         </form>
        )
}
```

Handling Multiple Inputs

1. When you need to handle multiple controlled input elements, you can add a name attribute to each element and let the handler function choose what to do based on the value of event.target.name.
2. Without <select> and <checkbox> other form element handles by **event.target.name** and **event.target.value**

```
export default class From extends Component {
        state = {
                input: "hello",
                text: "text",
                select: "",
                checkbox: true,
        };

        handle = (e) => {
                setState( { e.target.name : e.target.value } )
        };

        render() {
                const { input, text, select, checkbox } = this.state;
                return (
                  <form>
                    <input type="text " value={input}  onChange={this.handle}   />
                    <textarea name="text" value={text} onChange={this.handle} />
                  </form>
                );
```

```
        }
    }
```

The file input Tag

Input file tag should be controlled by uncontrolled component. because in file there are no value attribute.

```
<input type="file" />
```

Uncontrolled component

The uncontrolled input is similar to the traditional HTML form inputs. The DOM itself handles the form data. Here, the HTML elements maintain their own state that will be updated when the input value changes. To write an uncontrolled component, you need to use a ref to get form values from the DOM. In other words, there is no need to write an event handler for every state update. You can use a ref to access the input field value of the form from the DOM.

```
class App extends React.Component {
    constructor(props) {
        super(props);
        this.updateSubmit = this.updateSubmit.bind(this);
        this.input = React.createRef();
    }
    updateSubmit(event) {
        alert('You have entered the UserName and CompanyName successfully.');
        event.preventDefault();
    }
    render() {
        return (
          <form onSubmit={this.updateSubmit}>
            <h1>Uncontrolled Form Example</h1>
            <label>Name:
              <input type="text" ref={this.input} />
            </label>
            <label>
              CompanyName:
              <input type="text" ref={this.input} />
            </label>
            <input type="submit" value="Submit" />
          </form>
        );
    }
}
```

export default App;

React Memo

1. memo use in component.
2. memo use for skip rendering a component if its props have not changed.

Example :

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

const Todos = ({todos}) => {
        console.log ("child render");

        return (
                <>
                 <h2>My Todos</h2>
                 { todos.map((todo, index) => {
                   return <p key={index}>{todo}</p>;
                 }) }
                </>
        );
    };

export default memo(Todos);
```

ref problem in memo:

In react function are reference type data. When we called a function, the function reference is change. So that react new render function is new function. We solve by **usecallback**() function .

Ref

1. Refs are a function provided by React to access the DOM element and the React element that you might have created on your own.
2. They are used in cases where we want to change the value of a child component, without making use of props and all.
3. They also provide us with good functionality as we can use callbacks with them.
4. Refs are created using React.createRef() and attached to React elements via the ref attribute.

```
const refContainer = React.createRef();
```

5. When a ref is passed to an element in render, a reference to the node becomes accessible at the current attribute of the ref.

```
const node = this.myRef.current
```

6. You may not use the ref attribute on function components because they don't have instances.

Example:

```
class CustomTextInput extends React.Component {
    constructor(props) {
        super(props);
        this.textInput = React.createRef();
        this.focusTextInput = this.focusTextInput.bind(this);
    }
    handle() {
        this.textInput.current.focus();
    }
    render() {
        return (
        <input  type="text" ref={this.textInput} />
        <input  type="button"  value="Focus" onClick={this.handle} />
        );
    }
}
```

ForwardRef

Ref forwarding is an opt-in feature that lets some components take a ref they receive, and pass it further down (in other words, "forward" it) to a child.

FancyBUtton.js:

```
const FancyButton = (props, ref) => (
 <button ref={ref} className="FancyButton">    {props.children}  </button>
));

Export default React.forwardRef(FancyButton);
```

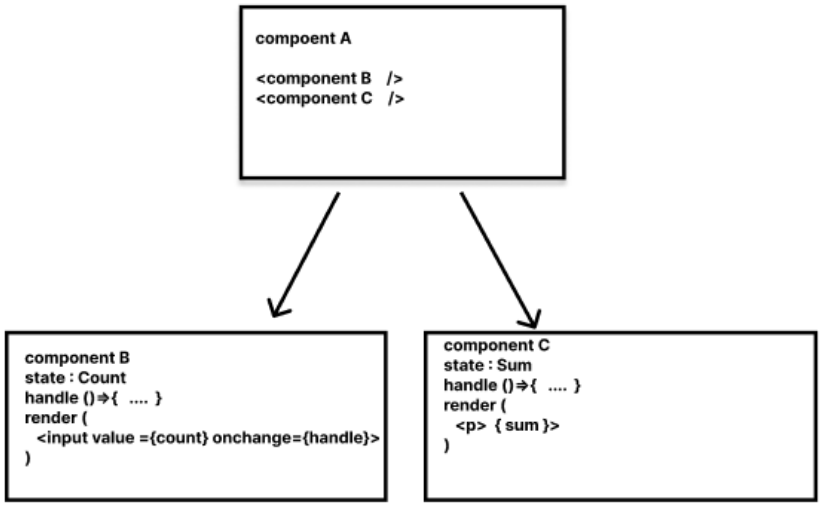
Main.js

```
const ref = React.createRef();
<FancyButton ref={ref} > Click me!  </ FancyButton>;
```

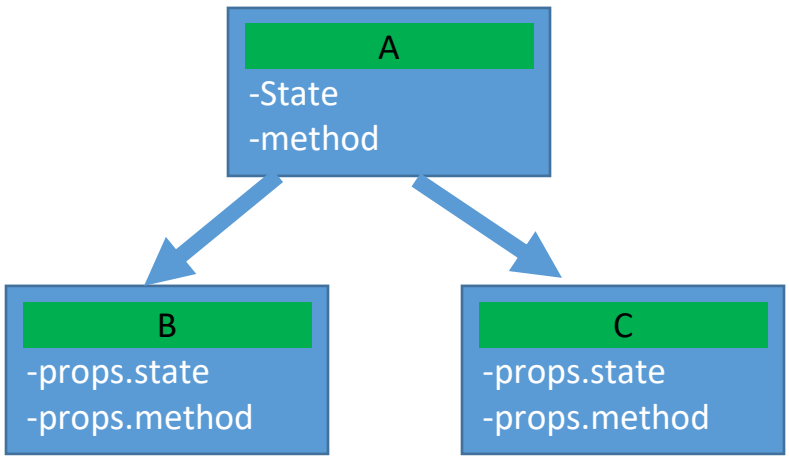
Lifting state:

In lifting solution, all state handle by top level component. Then with props we send handler and state to bottom component.

In view:



Here component A export component B, C. here component B take input but cannot send to component C. So, it is a problem to send data to other component. to solve this, we use lifting state up



Here all state and function are in top lavel component A. then with props component sent to B and C. it is called lifting state up.

Composition vs Inheritance

Inheritance:

Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object.

```
class Car {
        constructor(brand) {
                this.carname = brand;
        }
        present() {
                return 'I have a ' + this.carname;
        }
}
class Model extends Car {
        constructor(brand, mod) {
                super(brand);
                this.model = mod;
        }
        show() {
                return this.present() + ', it is a ' + this.model;
        }
}
```

Problem of inheritance:

1. In inheritance, react child component extends all parent component method. If child want to use single method of parent component it not possible.
2. component is tightly coupled.
3. from child it not clear what parents does.

Composition:

composition is the name for passing components as props to other components, thus creating new components with other component. Props.children are used for composition value.

```
function App () {
        return (
                <div>
                  <Hello>
                        <h2> name </h2>
                  </Hello>
                </div>
```

```
        )
}

export default function Hello(props) {
        return (
          <div>
                { props.children }
          </div>
        )
}
```

Pattern of share State of react

To share same functionality in all component there are two pattern.

1. higher order component
2. props rendering

Higher-Order Components

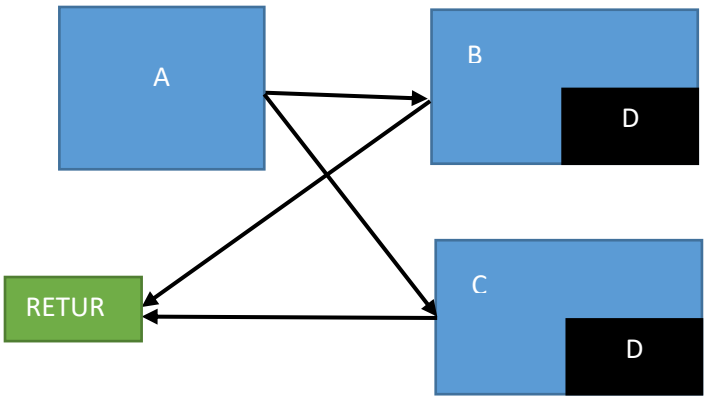
- higher-order component is a function that takes a component and returns a new component. Where as a component transforms props into UI, a higher-order component transforms a component into another component.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

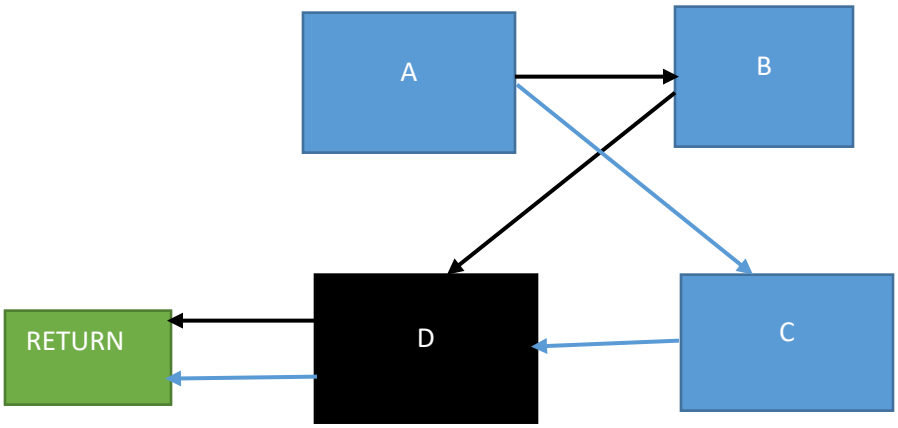
- when we use similar function in various component, we use higher-order component. we can solve it by lifting state up. In lifting state up, we send component in props. If want to send very deep component it is very bad. So, in react some time lifting state up is not good for component. so we use HOC in react.
- If we want to more add component/event in component it is very difficult to add component in every other component. With higher order component is very easy.

```
function add (a, b) {
        return a + b
}
function higherOrder(a, addReference) {
        return addReference(a, 20)
}
//Function call
higherOrder(30, add) // 50
```

Component without HOC:



Component with HOC:



Hoc function:

```
const WithCounter = (OriginalCom) => {
        class NewCom extends React.Component {
                state = {
                        count: 0,
                };
                handle = () => {
                        this.setState((m) => ({
                                count: m.count + 1,
                        }));
                };
                render() {
                         const {count} = this.state;
                         return <OriginalCom count={count} handle={this.handle} />
                }
```

```
        }
        return NewCom;
};
export default WithCounter;
```

other function.

```
import WithCounter from "./WithCounter";
const ClickCounter = (props) => {
        const {count , handle} = props;
        return (
                <div>
                 <button type="button" onClick={handle}>
                  click {count} times
                 </button>
                </div>
         );
};
export default WithCounter(ClickCounter);
```

main function

```
function App() {
        return (
                <ClickCounter />
        )
}
```

We create new component with a state, that use in various component. Then we pass all the component in new state component and become other component.

Props Rendering

- "render prop" refers to a technique for sharing code between React components using a prop whose value is a function. In simple words, render props are simply props of a component where you can pass functions. These functions need to return elements, which will be used in rendering the components.
- In component we send data with help of props. And we can return it as function return

```
function Name ({value}) {
    return value;
}

export default function App () {
    const [value, setValue] = useState("");
```

```
        return (
          <div className="App">
                <Name value={ value } />
          </div>
        );
    }
```

- In render props pattern we pass function as a props.

```
export default class Counter extends React.Component{
        state={
                count : 0
        };
        render(){
                return this.render(count);
        }
}

export default class Click extends React.Component{
        render(){
                const {counter, handle } = this.props;
                return (
                      <div>
                        <button>  click { counter}   </button>
                       </div>
                )
        }
}

function App() {
        return (
                <div>
                  <Counter render={
                        (counter) =>{ <Click counter={counter} />   }
                       }
                  />
                </div>
        );
}
```

React Context

1. Context allows passing data through the component tree without passing props down manually at every level.
2. Context is primarily used when some data needs to be accessible by *many* components at different nesting levels
3. In React application, we passed data in a top-down approach via props. Sometimes it is inconvenient for certain types of props that are required by many components in the React application. Context provides a way to pass values between components without explicitly passing a prop through every level of the component tree.

React Context API

1. React.createContext
2. Context.provider
3. Context.Consumer
4. Class.contextType

React.createContext : we are creating context with React.createContext()

```
Const Context = React.createContext( defaultValue );
```

Context.Provider : Wrap child components in the Context Provider and supply the state value.

```
<Context.Provider  value={/* some value */}>
        <Component user={user} />
</  Context.Provider >
```

Context.Consumer:

1. wherever we want to consume (or use) what was provided on our context, we use the consumer component.
2. Consumer component take a function only

```
< Context.Consumer >
    { value =>  context value }
</ Context.Consumer >
```

Class.contextType:

The contextType property on a class used to assign a Context object which is created by React.createContext(). It allows you to consume the closest current value of that Context type using this.context. We can reference this in any of the component life-cycle methods, including the render function.

Example:

```
import React from 'react';
export const UserContext = React.createContext();
export default function App() {
        return (
          <UserContext.Provider value="Reed">
                <User />
          </UserContext.Provider>
        )
}

function User() {
        return (
          <UserContext.Consumer>
                {value => <h1>{value}</h1>}
          </UserContext.Consumer>
        )
}
```

HOOK

1. Hooks allow function components to have access to state and other React features
2. hook use in top level of code.
3. hook use in react function only.
4. Hooks cannot be conditional

Basic Hooks

1. useState
2. useEffect
3. useContext

Additional Hooks

1. useReducer
2. useCallback
3. useMemo
4. useRef
5. useImperativeHandle
6. useLayoutEffect
7. useDebugValue

Use state

1. The React useState Hook allows us to track state in a function component.
2. State generally refers to data or properties that need to be tracking in an application.
3. To use the `useState` Hook, we first need to `import` it into our component.
4. useState Hook is a special function which takes the initial state as the arguments and returns it as an input array.

Syntax:

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

state = initial state
setState = is a function that change the state
initialState = initial value of state

Example:

```
import React, { useState } from 'react';
function Example() {
    const [count, setCount] = useState(0);
    return (
        <div>
          <p>You clicked {count} times</p>
          <button onClick={() => setCount(count + 1)}>  Click me </button>
        </div>
    );
}
```

Update state:

1. We can update the state value by using setState method.
2. If want pass a function in setState method, this function takes pre value as parameter.

```
setState( (preValue) => { preValue+1 } );
```

Example:

```
import React, { useState } from 'react';
function Example() {
    const [count, setCount] = useState(0);
    const handle = ()=>{
        setCount( (m)=>( m+1 ) )
    }
    return (
        <p>You clicked {count} times</p>
        <button onClick={handle}> Click me </button>
    );
}
```

3. We can set value as direct in setState.

```
function FavoriteColor() {
        const [color, setColor] = useState("red");
        return (
                <h1>My favorite color is {color}!</h1>
                <button   onClick={() => setColor("blue")} >Blue</button>
        )
}
```

Using Multiple State Variables

```
const [age, setAge] = useState(42);
const [fruit, setFruit] = useState('banana');
const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
```

Use Effect

React are works for –

1. render UI
2. React on user input and action
3. Render JSX
4. manage state & props
5. Evaluate State and props change

React also do various work (side effect)

1. fetching data from API
2. updating Dom
3. setting any Subscription or timer

The side effect handle in class component with the method of-

1. componentDidMount ()
2. componentDidUpdate ()
3. componentWillUpdate ()

Here is some problem to use this method –

1. repeating code
2. unorganized code

In useEffect we solve all problem in functional component.

useEffect

1. Use Effect is a function which run every render.
1. Help us perform side Effect in functional components
2. Solves all the problem of lifecycle methods in class component

3. we do not repeat the code.

Syntax:

```
useEffect ( peram_1 ,  peram_2 );
```

param_1: take a function.

Param_2:

1. it is optional. It is use for which useEffect called or not.

2. if **param_2 = [ ]** then useEffect called only one time.

```
useEffect(() => {
        document.title = `You clicked ${count} times`;
}, [] );
```

3. if **param_2 = [props or state**], then it called when props or state change.

```
function Counter() {
const [count, setCount] = useState(0);
useEffect(() => {
        setCount(() => count * 2);
}, [count]);
return (
        <p>Count: {count}</p>
        <button onClick={() => setCount((c) => c + 1)}>+</button>
);
}
```

4. if we want to cleanup memory, we use return in useEffect. That do as like componentWillUnmount.

```
useEffect(() => {
        document.title = `You clicked ${count} times`;
        return ()=>{
        }
}, [] );
```

Example:

```
import React, { useState, useEffect } from 'react';
function Example() {
        const [count, setCount] = useState(0);
        useEffect(() => {    document.title = `You clicked ${count} times`;  });
         return (
          <div>
            <p>You clicked {count} times</p>
            <button onClick={ () => setCount(count + 1) }> Click me  </button>
          </div>
         );

}
```

useContext

1. useContext are use alternative of context.consumer component.
2. We can access value by useContext.

```
const value = useContext(MyContext);
```

Example:

```
function Component1() {
        const [user, setUser] = useState("Jesse Hall");
        return (
          <UserContext.Provider value={user}>
                <h1>{`Hello ${user}!`}</h1>
                <Component2 user={user} />
          </UserContext.Provider>
        );
}

function Component5() {
  const user = useContext(UserContext);
        return (
           <h2>{`Hello ${user} again!`}</h2>
        );
}
```

useCallback

The React useCallback Hook returns a memoized callback function.

Syntax:

```
const addTodo =  useCallback( param_1,  param_2 );
```

param1: take a function.

```
useCallBack ( () => {   document.title = `You clicked ${count} times` ;  } );
```

param2:

1. dependency variable , for which useCalled called or not.

Example:

```
import { useState, useCallback } from "react";
import ReactDOM from "react-dom/client";
import Todos from "./Todos";
```

```
const App = () => {
        const [count, setCount] = useState(0);
        const [todos, setTodos] = useState([]);

        const increment = () => {
                setCount((c) => c + 1);
        };
        const addTodo =  useCallback( () => {
          setTodos((t) => [...t, "New Todo"]);
        }, [todos] );

        return (
                <>
                  <Todos todos={todos} addTodo={addTodo} />
                  <hr />
                  <div>
                    Count: {count}
                    <button onClick={increment}>+</button>
                  </div>
                </>
        );
};
```

useMemo

1. The React useMemo Hook returns a memorized value. The useMemo Hook only runs when one of its dependency's updates.

Syntax:

```
const addTodo =  useMemo( param_1,  param_2 );
```

param1: take a function.

```
useCallBack ( () => {   document.title = `You clicked ${count} times` ;  } );
```

param2:

1. dependency variable , for which useCallback called or not.

Example:

```
const App = () => {
        const [count, setCount] = useState(0);
        const [todos, setTodos] = useState([]);
```

```
    const calculation = useMemo(  ()=> {
                        for (let i = 0; i < 1000000000; i++) {
                          num += 1;
                        }
                        return num;
                }, [count]);

    const increment = () => {
                setCount((c) => c + 1);
    };
    const addTodo = () => {
                setTodos((t) => [...t, "New Todo"]);
    };

    return (
     <div>
                <div>
                      <h2>My Todos</h2>
                      { todos.map (  (todo, index) => { return <p key={index} > {todo} </p>;  }) }
                        <button onClick = { addTodo }>  Add Todo </button>
                </div>
                <hr />
                <div>
                            Count: {count}
                            <button onClick={increment}>+</button>
                            <h2>Expensive Calculation</h2>
                            {calculation}
                </div>
      </div>
    );
};
```

useRef

4. The useRef is a hook that allows to directly create a reference to the DOM element in the functional component.
5. The useRef returns a mutable ref object. This object has a property called **.current.**
6. The value is persisted in the **refContainer.current** property.

Syntax:

```
const refContainer = useRef(initialValue);
```

Example:

```
function App() {
        const focusPoint = useRef(null);
        const onClickHandler = () => {
                focusPoint.current.value
                focusPoint.current.focus();
        };
        return (
            <button onClick={onClickHandler}>
            <textarea ref={ focusPoint } />
        );
};
```

useReducer

1. The useReducer Hook is like the useState Hook.
2. It allows for custom state logic.

Syntax: The useReducer Hook accepts two arguments.

useReducer(<**reducer**> , < **initialState** > , <init>)

Reducer: The reducer function contains your custom state logic.it take two arguments .

Reducer (**current state**, **action**)

Current state : value of state.
Action : event of component.

Initial State : the initialState be a simple value but generally will contain an object.

Init: You can also create the initial state lazily. To do this, you can pass an init function as the third argument. The initial state will be set to init(initialArg)

Return value of Reducer :

useReducer return two values. State and dispatch.

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

Example:

```
const initialState = {count: 0};
function reducer(state, action) {
        switch (action.type) {
                case 'increment':
                        return {count: state.count + 1};
                case 'decrement':
                        return {count: state.count - 1};
                default:
                        throw new Error();
        }
}
function Counter() {
        const [state, dispatch] = useReducer(reducer, initialState);
        return (
          <>
            Count: {state.count}
            <button onClick={() => dispatch({type: 'decrement'})}>-</button>
            <button onClick={() => dispatch({type  'increment'})}>+</button>
          </>
        );
}
```

React Router (V5)

To add React Router in your application

```
npm i -D react-router-dom
```

Basic Usage:

```
import ReactDOM from "react-dom/client";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Layout from "./pages/Layout";
import Home from "./pages/Home";

export default function App() {
        return (
          <BrowserRouter>
                  <Route path="/"  element={Layout}  />
                  <Route path="/home"  element={ Home }   />
          </BrowserRouter>
        );
}
```

If route is “/” then it open “/”, “/home” both component . becaouse in react route match with Browser Route.

Exact Route:

to use exact route use exact.

```
<BrowserRouter>
        <Route exact  path="/"  element={Layout}  />
        <Route exact path=”/home”  element={ Home }   />
</BrowserRouter>
```

Not found route:

```
<BrowserRouter>
        <Route exact  path="/"  element={Layout}  />
        <Route exact path=”/home”  element={ Home }   />
        <Route  component={ Error } />
</BrowserRouter>
```

Switch:

If we want to found first route, we use switch .

```
<BrowserRouter>
        <switch>
                <Route exact  path="/"  element={Layout}  />
                <Route exact path=”/home”  element={ Home }   />
                <Route  component={ Error } />
        <switch>
</BrowserRouter>
```

Dynamic Route:

```
<Route exact  path="/"  element={Layout}  />
<Route exact path=”/home/: id” element= {Home} />
```

Props in Route Component:

```
<Route exact  path="/">
        <Layout handle={add} />
<Route/>
<Route exact path=”/home/: id” >
        <Home />
<Route/>
```

Route in Render Props:

```
<Route exact  path="/"  render={ ()=> <Layout name=”shuvo” />}  />
```

Redirect Router:

```
<Route exact  path="/"  >
        <Redirect  to=”/home” />
<Route>
```

Link:

```
<Link to="/home ">Home</Link>

<Link to="/ ">layout</Link>
```

Link Parameter:

```
<Link to="/home?name=shuvo ">layout</Link>
```

Link Object:

```
<Link to= {{
        Pathname: “/home”,
        Search: “?name=shuvo”,
        Hash: “mt5”,
        State: { status: true }
}} />
```

NavLink :

To control style we use NavLink.

```
<NavLink  exact
        to="/home?name=shuvo"
        activeStyle={{
                fontWeight : ‘blod’,
                color : ‘red’
        }}
>
layout
</NavLink>
```

React Router (V6)

In react router V6-

1. No **switch** statement.
2. No **Exact** keyword
3. No **redirect** keyword

Basic Usage:

```
import ReactDOM from "react-dom/client";
import {BrowserRouter, Routes, Route} from "react-router-dom";
import Layout from “. /pages/Layout";
import Home from “. /pages/Home";

export default function App () {
```

```
    return (
     <BrowserRouter>
          <Route path="/" element= {<Layout />}
          <Route path="/home" element= {<Home />} />
          <Route path="/about" element= {<Navigate to="/contact" />} />
     </BrowserRouter>
    );
}
```

* keyword:

```
<BrowserRouter>
     <Routes>
          <Route path="/" element= {<Layout />}
          <Route path="/home/*" element= {<Home />} />
          <Route path="/home/:id" element= {<User />} />
          <Route path="/about" element= {<Navigate to="/contact" />} />
     <Routes>
</BrowserRouter>
```

No active Module:

```
<NavLink  to="/home?name=shuvo"
     className={ (value)=>value.isActive ? classes.active : " " }
>
     layout
</NavLink>
```

Route Hooks

useHistory: The useHistory hook gives you access to the history instance that you may use to navigate.

Example:

```
import { useHistory } from "react-router-dom";

function HomeButton() {
    let history = useHistory();
    function handleClick() {
        history.push("/home");
    }

    return (
     <button type="button" onClick={handleClick}>
       Go home
     </button>
    );
}
```

useLocation: The useLocation hook returns the location object that represents the current URL

```
function usePageViews() {
    let location = useLocation();
    React.useEffect(() => {
        ga.send(["pageview", location.pathname]);
    }, [location]);
}
```

useParams: useParams returns an object of key/value pairs of URL parameters

```
function BlogPost() {
    let { slug } = useParams();
    return <div>Now showing post {slug}</div>;
}
```

useRouteMatch:

Data fetch method of react

Fetch API:

```
export default function Quote() {
    const [quote, setQote]= useState(null);
    useEffect(()=>{
        const fetchQuote = async() => {
            const res = await fetch('http://api.quotable.io/random');
            const data = await res.json();
            setQote(data);
        }

        fetchQuote();

    },[])
    return (
        <h1> Quote </h1>
        <div> { quote?._id } </div>
```

```
        )
    }
```

AXIOS API:

With Async :

```
export default function Quote() {
  const [quote, setQote]= useState(null);
  useEffect(()=>{
      const fetchQuote = async() => {
          const res = await axios.get('http://api.quotable.io/random');
          console.log(res);
           setQote(res.data);
      }
      fetchQuote();
  },[])
  return (
      <h1> Quote </h1>
      <div> { quote?._id } </div>
  )
}
```

With promises:

```
useEffect(()=>{
  axios.get(`https://jsonplaceholder.typicode.com/posts/${id}`)
    .then(res=>{
        console.log(res);
        setPost(res.data);
    })
    .catch(err=>{
        console.log(err)
    })
},[id])
```