Unit – 2

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.

- ReactJs 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.

ReactJS History

Jordan Walke created React, who worked as a software engineer in Facebook has first released an early React prototype called "FaxJS."

In 2011, React was first deployed on Facebook's News Feed, and later in 2012 on Instagram.

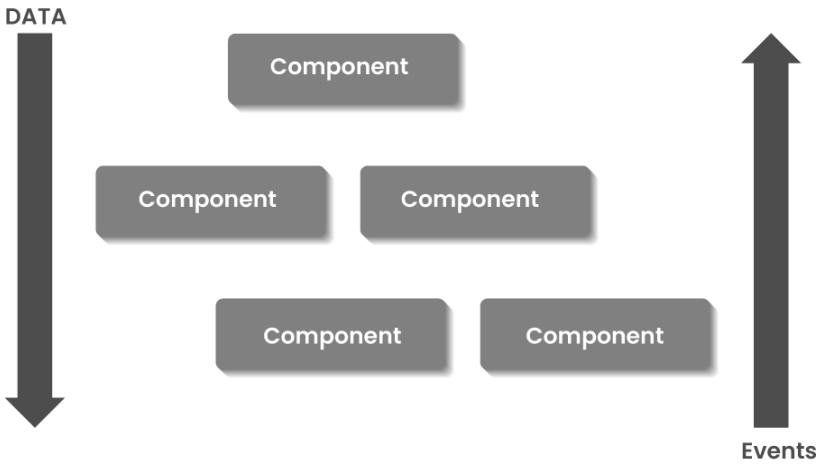
ReactJS Features

1. JSX - JavaScript Syntax Extension

JSX is a preferable choice for many web developers. It isn't necessary to use JSX in React development, but there is a massive difference between writing react.js documents in JSX and JavaScript. JSX is a syntax extension to JavaScript. By using that, we can write HTML structures in the same file that contains JavaScript code.

2. Unidirectional Data Flow and Flux

React.js is designed so that it will only support data that is flowing downstream, in one direction. If the data has to flow in another direction, you will need additional features.

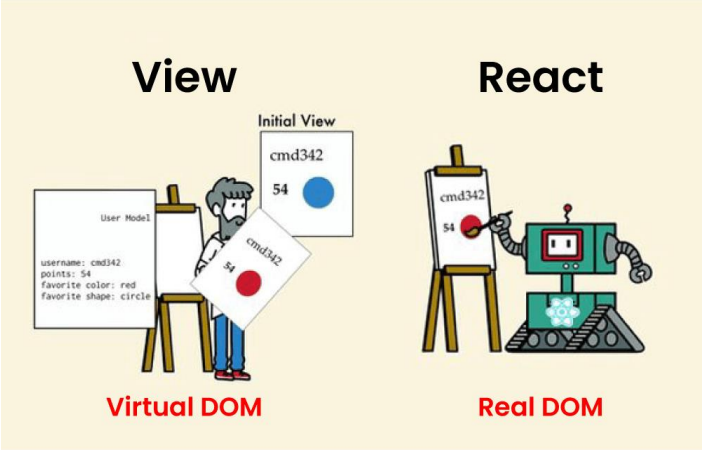


React contains a set of immutable values passed to the component renderer as properties in HTML tags. The components cannot modify any properties directly but support a call back function to do modifications.

3. Virtual Document Object Model (VDOM)

React contains a lightweight representation of real DOM in the memory called Virtual DOM. Manipulating real DOM is much slower compared to VDOM as nothing gets drawn on the screen. When any object's state changes, VDOM modifies only that object in real DOM instead of updating whole objects.

That makes things move fast, particularly compared with other front-end technologies that have to update each object even if only a single object changes in the web application.



4. Extensions

React supports various extensions for application architecture. It supports server-side rendering, Flux, and Redux extensively in web app development. React Native is a popular framework developed from React for creating cross-compatible mobile apps.

5. Debugging

Testing React apps is easy due to large community support. Even Facebook provides a small browser extension that makes React debugging easier and faster.

Hello world App in React

Step 1: Create a react application using the following command

npx create-react-app folder name

Step 2: Once it is done change your directory to the newly created application using the following command
cd foldername

Step 3: Now inside **App.js** and write down the following code as shown below:

```
import React from 'react';

import './App.css';
```

```
function App() {
        return (
        <h1> Hello World! </h1>
        );
}

export default App;
```

Step to run the application: Enter the following command to run the application.
npm start

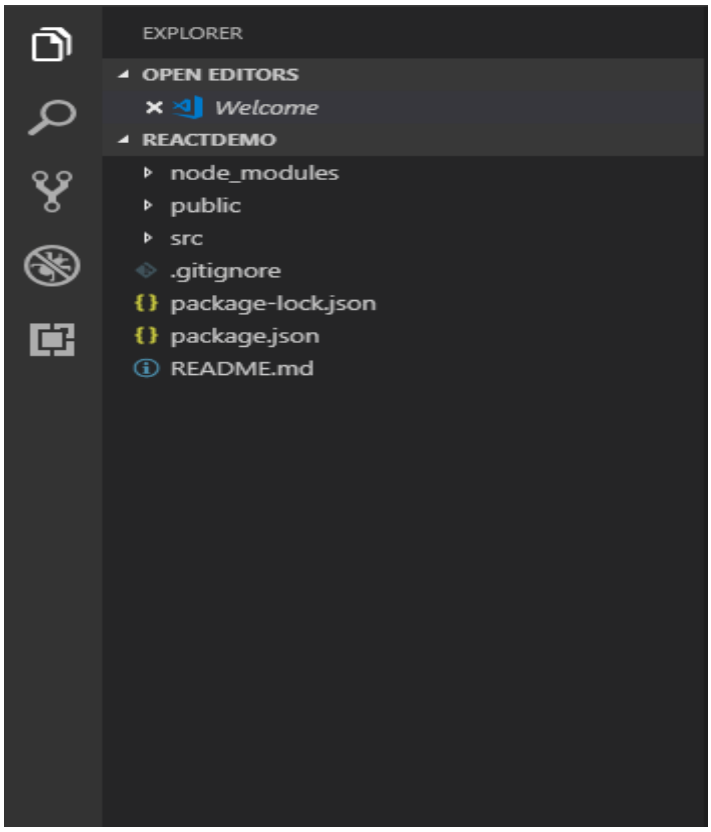
React App project directory structure

Click on Open folder and go to the folder where project got created.

Select the folder and it will open in VS Code editor.

Folder Structure

The React application automatically creates required folders, as shown below.



node_modules

This folder will contain all react js dependencies.

.gitignore

This file is used by source control tool to identify which files and folders should be included or ignored during code commit

package.json

This file contains dependencies and scripts required for the project.

Src folder

src is one of the main folder in react project.

Index.js

index.js is the file that will be called once we will run the project.

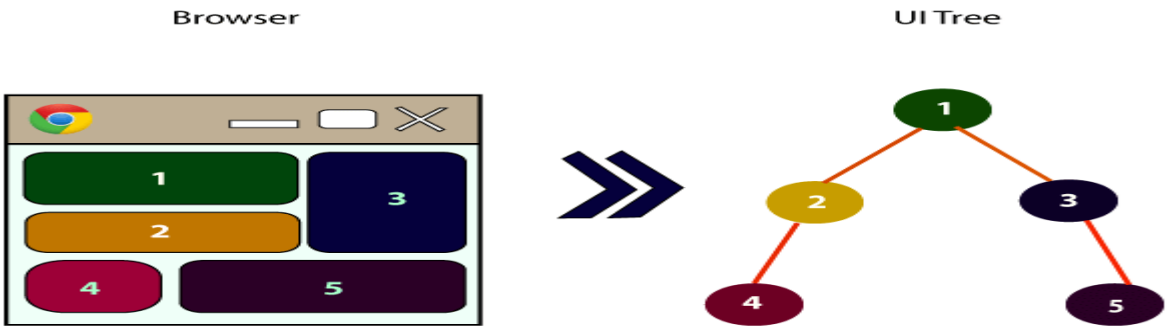
App.js

App.js is a component that will get loaded under index.js file. If we do any change in app.js file HTML component and save it it will reflect in localhost://3000

React Component

A Component is considered as the core building blocks of a React application. It makes the task of building UIs much easier. Each component exists in the same space, but they work independently from one another and merge all in a parent component, which will be the final UI of your application.

Every React component have their own structure, methods as well as APIs. They can be reusable as per your need. For better understanding, consider the entire UI as a tree. Here, the root is the starting component, and each of the other pieces becomes branches, which are further divided into sub-branches.



JSX

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.

Hello React app

Corresponding Output

React.createElement("div", **null**, "Hello React App");

Why use JSX?

It is faster than regular JavaScript because it performs optimization while translating the code to JavaScript.

Instead of separating technologies by putting markup and logic in separate files, React uses components that contain both. We will learn components in a further section.

It is type-safe, and most of the errors can be found at compilation time.

It makes easier to create templates.

Nested Elements in JSX

To use more than one element, you need to wrap it with one container element. Here, we use div as a container element which has three nested elements inside it.

```
1. import React, { Component } from 'react';
2. class App extends Component{
3.   render(){
4.     return(
5.       <div>
6.         <h1>Reactapp</h1>
7.         <h2>JSX</h2>
8.         <p>This website contains the JSX information.</p>
9.       </div>
10.    );
11.  }
12. }
```

```
13. export default App;
```

JSX Attributes

JSX use attributes with the HTML elements same as regular HTML. JSX uses camel case naming convention for attributes rather than standard naming convention of HTML such as a class in HTML becomes className in JSX because the class is the reserved keyword in JavaScript. We can also use our own custom attributes in JSX. For custom attributes, we need to use data- prefix. In the below example, we have used a custom attribute data-demo Attribute as an attribute for the <p> tag.

```
1.  import React, { Component } from 'react';
2.  class App extends Component{
3.    render(){
4.      return(
5.        <div>
6.          <h1>React App</h1>
7.         <h2>JSX</h2>
8.          <p data-demoAttribute = "demo">This website contains the JSX information.</p>
9.        </div>
10.     );
11.   }
12. }
13. export default App;
```

In JSX, we can specify attribute values in two ways:

1. As String Literals: We can specify the values of attributes in double quotes:

var element = <h2 className = "firstAttribute">Hello React</h2>;

```
1.  import React, { Component } from 'react';
2.  class App extends Component{
3.    render(){
4.      return(
5.        <div>
6.          <h1 className = "hello" >Reactapp</h1>
7.          <p data-demoAttribute = "demo">This website contains the JSX information.</p>
8.        </div>
9.      );
10.   }
```

```
}
export default App;
```

2. As Expressions: We can specify the values of attributes as expressions using curly braces {}:

```
var element = <h2 className = {varName}>Hello React</h2>;
```

```
import React, { Component } from 'react';
class App extends Component{
  render(){
    return(
      <div>
        <h1 className = "hello" >{25+20}</h1>
      </div>
    );
  }
}
export default App;
```

JSX Comments

JSX allows us to use comments that begin with /* and ends with */ and wrapping them in curly braces {} just like in the case of JSX expressions. Below example shows how to use comments in JSX.

JSX Styling

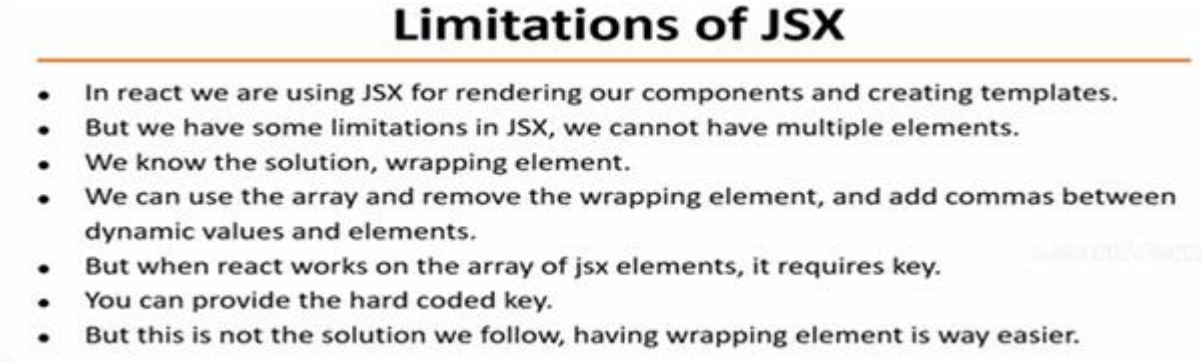
React always recommends using inline styles. To set inline styles, you need to use camelCase syntax. React automatically allows appending px after the number value on specific elements. The following example shows how to use styling in the element.

```
import React, { Component } from 'react';
class App extends Component{
  render(){
    var myStyle = {
      fontSize: 80,
      fontFamily: 'Courier',
      color: '#003300'
    }
    return (
      <div>
```

```
11.           <h1 style = {myStyle}>www.reactjs.org</h1>
12.         </div>
13.       );
14.    }
15. }
16. export default App;
```

NOTE: JSX cannot allow to use if-else statements. Instead of it, you can use conditional (ternary) expressions.

```
1.  import React, { Component } from 'react';
2.  class App extends Component{
3.    render(){
4.       var i = 5;
5.       return (
6.          <div>
7.        <h1>{i == 1 ? 'True!' : 'False!'}</h1>
8.          </div>
9.       );
10.    }
11. }
12. export default App;
```



Original DOM vs Virtual DOM

DOM stands for “Document Object Model,” which represents your application UI and whenever the changes are made in the application, this DOM gets updated and the user is able to visualize the

changes. DOM is an interface that allows scripts to update the content, style, and structure of the document.

Virtual DOM is a node tree similar to Real DOM that lists elements, content, and attributes as objects and properties. React render() method creates a node tree from the react components. Then it updates the node tree in response to the mutations in the data model caused by various actions done in the UI.

Take a simple example of an array

```
let languages = [cpp, java, python]
```

Now we want to replace python with java script. For that, we need to create a new array.

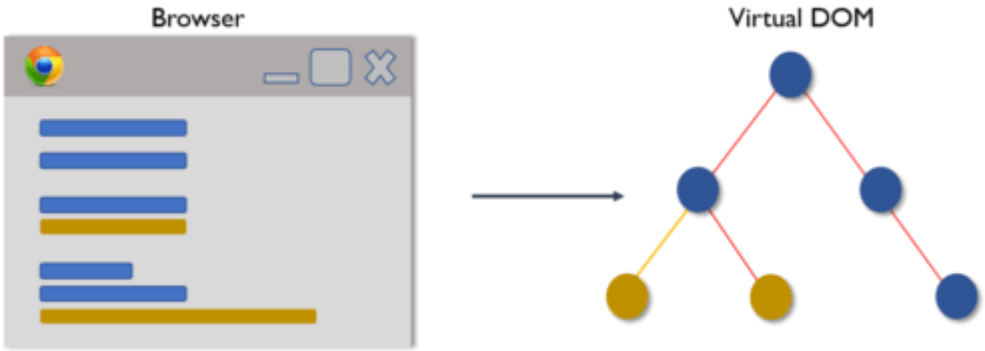
```
let languages = [cpp, java, javascript]
```

Instead of this, we can just traverse to languages[2] and update only the element. Instead of redoing the whole thing, we just changed the element which we needed to update. The same thing is done by Virtual DOM. Instead of updating all the node elements, it just updates the changed elements.

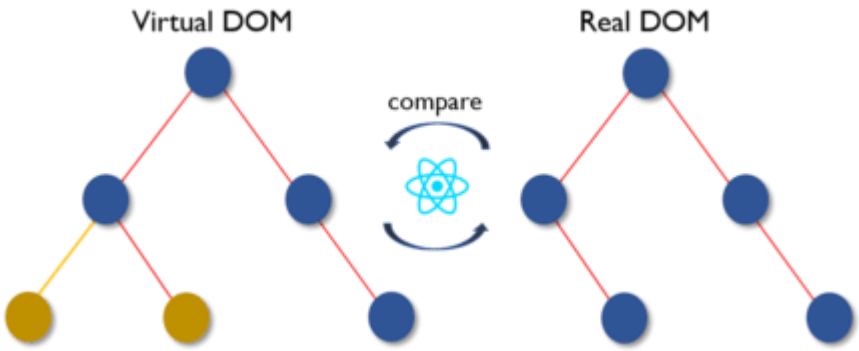
Virtual DOM does the same thing. Instead of updating all the node elements, it just updates the changed elements. Virtual DOM does the same thing. Virtual DOM does the same thing. Instead of updating all the node elements, it just updates the changed elements.

ReactDOM.render() will create a Virtual and real DOM tree of the first load. When events like click, keypress, or API response occur, Virtual DOM tree elements are notified for state or prop change; if that state or props are updated, then the node elements are updated. When changes are done in UI, the changes are also done in Virtual DOM. Instead of updating all the nodes, Virtual DOM updates only those components in which changes are made.

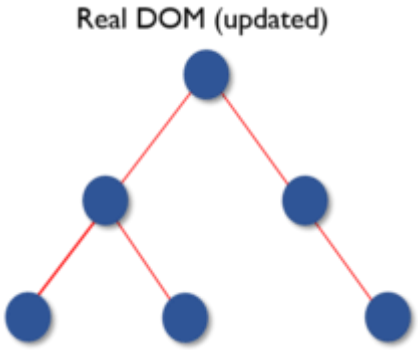
Once Virtual DOM contains all the updated changes, it is then compared with the Real DOM and the difference is calculated between them.



Once Virtual DOM contains all the updated changes it is then compared with the Real DOM and the difference is calculated between them.



Once the difference is calculated the **real DOM will update only the new components that have actually changed**. This is called **Reconciliation**. **Virtual DOM is faster and more effective than Real DOM** as it just focuses on the updated components instead of updating the entire DOM.



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.

A Component is considered as the core building blocks of a React application. It makes the task of building UIs much easier. Each component exists in the same space, but they work independently from one another and merge all in a parent component, which will be the final UI of your application.

Every React component have their own structure, methods as well as APIs. They can be reusable as per your need. For better understanding, consider the entire UI as a tree. Here, the root is the starting component, and each of the other pieces becomes branches, which are further divided into sub-branches.

Functional Components

Class Components

Class Components

Class components are more complex than functional components. It requires you to extend from React. Component and create a render function which returns a React element. You can pass data from one class to other class components. You can create a class by defining a class that extends Component and has a render function. Valid class component is shown in the below example.

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

The class component is also known as a stateful component because they can hold or manage local state. It can be explained in the below example.

In this example, we are creating the list of unordered elements, where we will dynamically insert StudentName for every object from the data array. Here, we are using ES6 arrow syntax (=>) which looks much cleaner than the old JavaScript syntax. It helps us to create our elements with fewer lines of code. It is especially useful when we need to create a list with a lot of items.

```
import React, { Component } from 'react';
class App extends React.Component {
 constructor() {
    super();
    this.state = {
      data:
      [
        {
          "name":"Abhishek"
        },
        {
          "name":"Saharsh"
        },
        {
          "name":"Ajay"
        }
      ]
    }
  }
```

```
20.   render() {
21.     return (
22.       <div>
23.         <StudentName/>
24.         <ul>
25.           {this.state.data.map((item) => <List data = {item} />)}
26.         </ul>
27.       </div>
28.     );
29.   }
30. }
31. class StudentName extends React.Component {
32.   render() {
33.     return (
34.       <div>
35.         <h1>Student Name Detail</h1>
36.       </div>
37.     );
38.   }
39. }
40. class List extends React.Component {
41.   render() {
42.     return (
43.       <ul>
44.         <li>{this.props.data.name}</li>
45.       </ul>
46.     );
47.   }
48. }
49. export default App;
```

When creating a React component, the component's name MUST start with an upper-case letter.

Class Component

To define a React component class, your class needs to extend with React.Component.

The render() method must be defined in the class component. Other React.Component methods are optional like constructor() componentDidMount(), etc.

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

const element = <Hello />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Functional Component

Here we going to change the above class component into a functional component.

```
function Hello() {
  return <h1>Hello World!</h1>;
}

const element = <Hello />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Functional components are easy to write with less code and easier to understand. Because it is just a plain JavaScript function that returns JSX.

Differences between functional components and class components

We go through each difference between class components and functional components.

- 1. Components Rendering
- 2. Handling Props
- 3. Handling States
- 4. Lifecycle Methods
- 5. Accessing Components Children
- 6. Higher-Order Components
- 7. Error Boundaries

1. Components Rendering

In the class component, the render() method is used for rendering the JSX by extending React.Component

The functional component is just a plain JavaScript function that returns JSX.

The first two programs are a perfect example of component rendering.

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

//Function component
function Hello() {
  return <h1>Hello World!</h1>;
}

//Function component with Arrow function
Hello = () => {
  return <h1>Hello World!</h1>;
}
```

Winner: Functional Component

Functional component creation is more simple by using the arrow function. A plain JavaScript function is used as a functional component. No more render method.

2. Handling Props

The props stands for properties and is passed into the React component. It's also used for passing data from one component to another.

Class component with Props

this.props used to access our name props.

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

const element = <Hello name="World"/>;
ReactDOM.render(
  element,
```

```
  document.getElementById('root')
);
```

Functional component with Props

The valid React functional component should have one argument for the function.
The props argument will have all the component props like props.name.

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

const element = <Hello name="World" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Winner: Functional Component

No worries about the this keyword. The syntax is clean and simple, it's a winner.

3. Handling State

state is a built-in object of React component, it is used to control the component behaviors. When the state object changes, the component will be re-renders.

Class component with state

```
class Timenow extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return(
      <div>
        <h1>Hello {this.props.name}!</h1>
        <h2>Time Now {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

const element = <Timenow name="World"/>;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

The class components constructor method is used to initialize the state values.

Functional component with state

In the functional component, the state is handled using the useState Hook.

Hooks *are a new addition to React 16.8. They let you use state and other React features without writing a class.*

The functional component doesn't have this, so we can’t assign or read this.state. Instead, we call the useState Hook directly inside our functional component.

```
function Timenow(props) {
  return (
   <div>
     <h1>Hello {props.name}!</h1>
     <h2>Time Now {date.toLocaleTimeString()}.</h2>
   </div>
 );
}

const element = <Timenow name="World" />;
ReactDOM.render(
 element,
 document.getElementById('root')
);
```

Winner: Functional Component

The state initialization is done by using the useState hook. No constructor method is required.

4. Lifecycle Methods

In React, each component has several lifecycle methods, this method helps you to run code at particular times in the process.

Class Component with Lifecycle Methods

The below clock class component is a perfect example of implementing lifecycle methods.

```
class Clock extends React.Component {
 constructor(props) {
   super(props);
   this.state = {date: new Date()};
 }

 componentDidMount() {
   this.timerID = setInterval(
     () => this.tick(),
     1000
   );
 }
```

```
  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Functional Component with Lifecycle Hook Methods

We converted above the clock class component into a functional component using the useEffect hook.

The useEffect return method is used to clean up.

```
function Clock(props) {
  const [date, setDate] = React.useState(new Date());

  React.useEffect(() => {
    var timerID = setInterval(() => tick(), 1000);

    return function cleanup() {
      clearInterval(timerID);
    };
  });

  function tick() {
    setDate(new Date());
  }
```

```
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {date.toLocaleTimeString()}.</h2>
    </div>
  );
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Winner: Class Component

The functional component useEffect is confusing due to using the same hook for all the lifecycle methods. In the class component, we can directly use the
methods componentDidMount, componentWillUnmount, etc.

5. Accessing Components Children

The special children prop is used to access the component inside content or component like <Layout>inside content</Layout>.

Class Component

this.props.children is used for class components.

```
class Layout extends React.Component {
  render() {
    return(
      <div>
        <h1>Hello {this.props.name}!</h1>
        <div>{this.props.children}</div>
      </div>
     );
  }
}

const element = <Layout name="World">This is layout content</Layout>;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Functional Component

Just props.children is used in functional components to access the children's content.

```
function Layout(props) {
  return(
    <div>
      <h1>Hello {props.name}!</h1>
      <div>{props.children}</div>
    </div>
  );
}

const element = <Layout name="World">This is layout content</Layout>;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Winner: Class & Functional Component

Both use the same props.children to access the component children. In addition, we need to use the this keyword.

6. Higher-Order Components

A higher-order component (HOC) is an advanced technique in React for reusing component logic. HOC is a pure function, so it only returns a new component.

Higher-order component is a function that takes a component and returns a new component.

HOC with Class Component

```
function classHOC(WrappedComponent) {
  return class extends React.Component{
    render() {
      return <WrappedComponent {...this.props}/>;
    }
  }
}

const Hello = ({ name }) => <h1>Hello {name}!</h1>;
const NewComponent = classHOC(Hello);

const element = <NewComponent name="World" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

HOC with Functional Component

```
function functionalHOC(WrappedComponent) {
  return (props) => {
    return <WrappedComponent {...props}/>;
  }
}

const Hello = ({ name }) => <h1>Hello {name}!</h1>;
const NewComponent = functionalHOC(Hello);

const element = <NewComponent name="World" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Winner: Functional Component

To create HOC, we should use the JavaScript function (classHOC, functionalHOC). Don't Use HOCs inside the render method. Inside the function, we can use a class or functional components.

Props

In ReactJS, the data can be passed from one component to another component using these props, similar to how the arguments are passed in a function. Inside the component, we can add the attributes called props; however, we cannot change or modify props inside the component as they are immutable.

Using the "this.props", we can make the props available inside the components. Then, the dynamic data can be rendered through the render method. We need to add the props to the reactDOM.render() in the Main.js file of our ReactJS project of ReactJS if we need immutable data in the component. Then we can use it in the component where we want to use those dynamic data.We pass the dynamic data for name attribute within the App component through the render method called reactDOM.render.

```
import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.js';

ReactDOM.render(<App name = "World of Programming!" />, document.getElementById('app'));
```

The below code is for App.js. Here, we are using the dynamic data for the name attribute which the Main.js file has passed as props, and making the props available within this App component using "this.props".

```
import React, { Component } from 'react';
class App extends React.Component {
  render() {
    return (
       <div>
         <h1> Welcome to the { this.props.name} </h1>
         <p> <h4> Here you will get to know so many things and your knowledge will be enhanced .
</h4> </p>
       </div>
    );
  }
}

export default App;
```

Default Props in ReactJS

Generally, the default props can be sent directly to the constructor of our component. Thus, the props need to be added to the reactDOM.render by us.

we can see how the default props are set within App.js.

```
import React, { Component } from 'react';
class App extends React.Component {
  render() {
    return (
       <div>
          <h1>Example for the default props : </h1>
         <h3>Welcome to the {this.props.name}</h3>
         <p>Here you will get to know so many things and your knowledge will be enhanced
.</p>
       </div>
      );
   }

}

App.defaultProps = {

  name: "World of Programming!"
```

```
}

export default App;
```

State and Props in ReactJS

States are the type of built-in object in ReactJS

We can create, handle, or manage our data within the component using the state object. Data can be passed by the props but the data cannot be passed by state itself. Using the state, the data is managed internally within the component.

We can combine both the state and props within our application in ReactJS. The steps for this combination process of the states and props within our ReactJS are as follows: First, the state needs to be set within our parent component. Then, the state can be passed as the props within the child component.

The code given below is for the App.js file. This code shows how we can set the state in our parent component and then pass it within the child component as the props and use it.

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

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Programming World",
    }
  }
  render() {
    return (
      <div>
        <PW pwProp = {this.state.name}/>
      </div>
    );
  }
}
class PW extends React.Component {
```

```
  render() {
    return (
      <div>
        <h1>The example for the combination of states and props within an application.</h1>
        <h3>Welcome to the this.props.pwProp}</h3>
        <p>Here you will get to know so many things and your knowledge will be enhanced .</p>
      </div>
    );
  }
}
export default App;
```

Example:

We pass the dynamic data for name attribute within the App component through the render method called reactDOM.render.

```
import React from 'react';

import ReactDOM from 'react-dom';

import App from './App.js';

ReactDOM.render(<App name = "World of Programming!" />, document.getElementById('app'));
```

The below code is for App.js. Here, we are using the dynamic data for the name attribute which the Main.js file has passed as props, and making the props available within this App component using “this.props”.

```
import React, { Component } from 'react';
class App extends React.Component {
  render() {
    return (
      <div>
        <h1> Welcome to the { this.props.name } </h1>
```

```
            <p> <h4> Here you will get to know so many things and your knowledge will be enhanced .
</h4> </p>
          </div>
      );
    }
}
export default App;
```

Difference between State and Props

SN	Props	State
1.	Props are read-only.	State changes can be asynchronous.
2.	Props are immutable.	State is mutable.
3.	Props allow you to pass data from one component to other components as an argument.	State holds information about the components.
4.	Props can be accessed by the child component.	State cannot be accessed by child components.
5.	Props are used to communicate between components.	States can be used for rendering dynamic changes with the component.
6.	Stateless component can have Props.	Stateless components cannot have State.
7.	Props make components reusable.	State cannot make components reusable.
8.	Props are external and controlled by whatever renders the component.	The State is internal and controlled by the React Component itself.

Changing in props and state

SN	Condition	Props	State
1.	Can get initial value from parent Component?	Yes	Yes
2.	Can be changed by parent Component?	Yes	No

3.	Can set default values inside Component?	Yes	Yes
4.	Can change inside Component?	No	Yes
5.	Can set initial value for child Components?	Yes	Yes
6.	Can change in child Components?	Yes	No

The component State and Props share some common similarities.

SN	State and Props
1.	Both are plain JS object.
2.	Both can contain default values.
3.	Both are read-only when they are using by this.

Destructuring

The destructuring is an ES6 feature that makes it possible to unpack values from arrays or properties from objects into distinct variables. In React, destructuring props and states improve code readability.

Destructuring was introduced in ES6. It's a JavaScript feature that allows us to extract multiple pieces of data from an array or object and assign them to their own variables. Consider we have an employee object with the following properties:

```
const employee = {
  firstName: "Ramesh",
  lastName: "Fadatare",
  emailId: ramesh@gmail.com

}Before ES6, you had to access each property individually:
console.log(employee .firstName) // Ramesh
console.log(employee .lastName) // Fadatare
console.log(employee .emailId) // ramesh@gmail.com
```

Destructuring lets us streamline this code:

const { firstName, lastName, emailId } = employee ;

is equivalent to

const firstName = employee .firstName

const lastName = employee .lastName

const emailId= employee .emailId

So now we can access these properties without the employee. prefix:

console.log(firstName) // Ramesh

console.log(lastName) // Fadatare

console.log(emailId) // ramesh@gmail.com

Destructuring props in Functional Components

Employee functional component

Create a functional component called Employee with the following code:

```
import React from 'react'
export const Employee = props => {
   return (
      <div>
         <h1> Employee Details</h1>
         <h2> First Name : {props.firstName} </h2>
         <h2> Last Name : {props.lastName} </h2>
         <h2> Eamil Id : {props.emailId} </h2>
      </div>
   )
```

} Note that we access employee data in the Employee component using props.

App component

We are passing employee **firstName**, **lastName** and **email** to *Employee* component using *props*:

```
function App() {
  return (
    <div className="App">
```

```
        <Employee firstName = "Ramesh" lastName = "Fadatare" emailId = "ramesh@gmail.com" />
    </div>
  );
}
```

Two ways to destructure props in functional component

The first way is destructuring it in the function parameter itself:

```
import React from 'react'
export const Employee =     ) => {
    return (
        <div>
            <h1> Employee Details</h1>
            <h2> First Name : {firstName} </h2>
            <h2> Last Name : {lastName} </h2>
            <h2> Eamil Id : {emailId} </h2>
        </div>
    )
}
```

The second way is destructuring props in the function body:

```
import React from 'react'
export const Employee = props => {
    const {firstName, lastName, emailId} = props;
    return (
        <div>
            <h1> Employee Details</h1>
            <h2> First Name : {firstName} </h2>
            <h2> Last Name : {lastName} </h2>
            <h2> Eamil Id : {emailId} </h2>
        </div>
    )
}
```

Destructuring props in class Components

```
import React, { Component } from 'react'
class Employee extends Component {
   render() {
      return (
        <div>
            <h1> Employee Details</h1>
            <h2> First Name : {this.props.firstName} </h2>
            <h2> Last Name : {this.props.lastName} </h2>
            <h2> Eamil Id : {this.props.emailId} </h2>
         </div>
      )
   }
}
export default Employee;
```

In class components, we destructure props in the render() function:

```
import React, { Component } from 'react'
class Employee extends Component {
   render() {
      return (
        <div>
            <h1> Employee Details</h1>
            <h2> First Name : {firstName} </h2>
            <h2> Last Name : {lastName} </h2>
            <h2> Eamil Id : {emailId} </h2>
         </div>
      )
   }
}
```

```
export default Employee;
```

Destructuring state

Destructuring states is similar to props. Here is syntax to Destructuring states in React:

```
class StateDemp extends Component {
   render() {
      const {state1, state2, state3} = this.state;
      return (
        <div>
            <h1> State Details</h1>
            <h2> state1 : {state1} </h2>
            <h2> state2 : {state2} </h2>
            <h2> state3 : {state3} </h2>
         </div>
      )
   }
}

export default StateDemo;
```

Sateless vs. Stateful components

When would you use a stateless component??

1. When you just need to present the props
2. When you don’t need a state, or any internal variables
3. When creating element does not need to be interactive
4. When you want reusable code

When would you use a stateful component?

1. When building element that accepts user input
2. ..or element that is interactive on page

3. When dependent on state for rendering, such as, fetching data before rendering
4. When dependent on any data that cannot be passed down as props

Class Component(Stateful)

```
import React, { Component } from 'react';
class StateExample extends Component {
  constructor(){
     super();
     this.state={

     }
  }
  render(){
     return (
        <div>
   <p> Class Component </p>
           <p>{this.state.first_name}</p>
           <p>{this.state.last_name}</p>
        </div>
     )
  }
}
export default StateExample;
```

Functional Component(Stateless)

```
import React from 'react';
function Example(props) {
   return(
```

```
        <div>
        <p>{props.first_name}</p>
        <p>{props.last_name}</p>
        </div>
    )
}
export default Example;
```

Stateful Components

Stateful components are those components which have a state. The state gets initialized in the constructor. It stores information about the component's state change in memory. It may get changed depending upon the action of the component or child components.

Stateless Components

Stateless components are those components which don't have any state at all, which means you can't use this.setState inside these components. It is like a normal function with no render method. It has no lifecycle, so it is not possible to use lifecycle methods such as componentDidMount and other hooks. When react renders our stateless component, all that it needs to do is just call the stateless component and pass down the props.

A stateless component renders output which depends upon props value, but a stateful component render depends upon the value of the state. A functional component is always a stateless component, but the class component can be stateless or stateful.

Container components vs Presentational components

Smart components vs Dumb components

Having a parent component pass data down to its children also assures that if there is any debugging needed regarding state management, we can go to the parent component to see what's up, instead of checking state in each child component. All the children components have to worry about is receiving the information as props properly (no pun intended).

So presentational components can vary depending on what information it receives. The difference is that a stateful component keeps track of the information itself, instead of just taking it via props and outputting it.

If information is completely static and you know it will never change, we have a very 'presentational' component indeed.

```
const Rules = () => {
```

```
  return (
    <div>
      <p>The rules are simple and unchanging:</p>
      <ol>
        <li>You don’t talk about the rules.</li>
        <li>You must follow the first rule.</li>
      </ol>
    </div>
  )
}
```

Parent-child communication

Following are the steps to pass data from child component to parent component:

In the parent component, create a callback function. This callback function will retrieve the data from the child component.

Pass the callback function to the child as a props from the parent component.

The child component calls the parent callback function using props and passes the data to the parent component.

Filepath- src/App.js

```
import React from 'react';
import Child from './Child'
class App extends React.Component{
        state = {
                name: "",
        }
        handleCallback = (childData) =>{
                this.setState({name: childData})
        }
        render(){
                const {name} = this.state;
```

```
            return(
                <div>
                    <Child parentCallback = {this.handleCallback}/>
                    {name}
                </div>
            )
        }
}
export default App
```

Filepath- src/component/Child.js

```
import React from 'react'

class Child extends React.Component{
        onTrigger = (event) => {
            this.props.parentCallback(event.target.myname.value);
            event.preventDefault();
        }
        render(){
            return(
            <div>
                <form onSubmit = {this.onTrigger}>
                    <input type = "text"
                    name = "myname" placeholder = "Enter Name"/>
                    <br></br><br></br>
                    <input type = "submit" value = "Submit"/>
                    <br></br><br></br>
                </form>
            </div>
            )
        }
}
export default Child.
```