UNIT-2

REACT BASICS

2.1 DIFFERENT COMPONENTS

2.1.1 Class component

Before React 16.8, **Class components** were the only way to track state and lifecycle on a React component. Function components were considered "state-less".

With the addition of Hooks, Function components are now almost equivalent to Class components. The differences are so minor that you will probably never need to use a Class component in React.

Even though Function components are preferred, there are no current plans on removing Class components from React.

These are more complex components as compared to functional components. Due to their complex nature, they are stateful in nature. **Class components** can be used whenever state management is required in react. The functional components have no idea about other components present in the application whereas class components can communicate with other components that their data can be transferred to other components using class components. Creating a class component requires us to create JavaScript classes.

Syntax

Class components can be created using the below syntax:

```
class Componentname extends React.Component
{
	render(){
		return <h1>Welcome Text!</h1>;
	}
}
```

Example

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

//class component
class Bike extends React.Component {
	constructor() {
		super();
		this.state = {color: "green"};
	}
```

```
        render() {
            return <h2>This is a Bike!</h2>;
        }
}
```

Example

```
import React from "react";
class Sample extends React.Component {
        render()
        {
            return <h1>Diploma Computer/It Engineering</h1>;
        }
}
class App extends React.Component
{
        render()
        {
            return <Sample />
        }
}
export default App;
```

Output:

Diploma Computer/It Engineering.

2.1.2 Function component

A **React functional component** is a straight JavaScript function that takes props and returns a React element. Writing *functional components* has been the traditional way of writing React components in advanced applications since the advent of **React Hooks**.

A React component's primary function is to classify the displayed view and connect it to the code that governs its actions. **React's functional components** reduce this to the most basic profile feasible: a function that accepts properties and returns a JSX definition. The function body contains all of the definitions needed for actions, and the class-related sections of object-oriented components are omitted.

Functional components can be considered as simple **functions build using JavaScript**. These are stateless components that cannot be used to achieve complex tasks.

Syntax

Functional components can be created using the below syntax:

```
function Componentname()
{
        return <h1>Welcome Text!</h1>;
}
```

Example

```
import React from 'react';

const App = () => {
  return <Headline />;
};

const Headline = () => {
  const greeting = 'Hello Function Component!';

  return <h1>{greeting}</h1>;
};

export default App;
```

Example

```
import React from 'react'
import './App.css';
function App()
{
return (
<div className="App">
 <h1>My First Functional Component</h1>
 </div>
);
}
export default App;
```

Output:

My First Functional Component

2.2 PROPS

Props stand for "**Properties**." They are **read-only** components. It is an object which stores the value of attributes of a tag and work similar to the HTML attributes. It gives a way to pass data from one component to other components. It is similar to function arguments. Props are passed to the component in the same way as arguments passed in a function.

Props are **immutable** so we cannot modify the props from inside the component. Inside the components, we can add attributes called props. These attributes are available in the component as **this.props** and can be used to render dynamic data in our render method.

When you need immutable data in the component, you have to add props to **reactDom.render()** method in the **main.js** file of your ReactJS project and used it inside the component in which you need. It can be explained in the below example.

Props are arguments passed into React components.

Props are passed to components via HTML attributes.

React Props are like function arguments in JavaScript *and* attributes in HTML.

Example

```
//Create a variable named carName and send it to the Car component:
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
        return <h2>I am a {props.brand}!</h2>;
}
function Cars() {
        const carName = "Audi";
        return
        (
                <>
                <h1>Describe how the vehicle looks and runs</h1>
                <Car brand={carName} />
                </>
        );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Cars />);
```

Output:

Describe how the vehicle looks and runs

I am a Audi!

Example

```
function Car(props) {
    return <h2>I am a { props.brand }!</h2>;
}
function Garage() {
    const carName = "Ford";
    return (
    <>
    <h1>Who lives in my garage?</h1>
    <Car brand={ carName } />
    </>
    );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

2.3 STATE

The state is an updatable structure that is used to contain data or information about the component. The state in a component can change over time. The change in state over time can happen as a response to user action or system event. A component with the state is known as stateful components. It is the heart of the react component which determines the behavior of the component and how it will render. They are also responsible for making a component dynamic and interactive.

A state must be kept as simple as possible. It can be set by using the **setState()** method and calling setState() method triggers UI updates. A state represents the component's local state or information. It can only be accessed or modified inside the component or by the component directly. To set an initial state before any interaction occurs, we need to use the **getInitialState()** method.

React components has a built-in state object.

The state object is where you store property values that belong to the component.

When the state object changes, the component re-renders.

Defining the state

To define a state, you have to first declare a default set of values for defining the component's initial state. To do this, add a class constructor which assigns an initial state using this.state. The '**this.state**' property can be rendered inside **render()** method.

Conventions of using state in React:

The state of a component should prevail throughout its lifetime, thus we must first have some initial state, to do so we should define the State in the constructor of the component's class.

The state should never be updated explicitly. React uses an observable object as the state that observes what changes are made to the state and helps the component behave accordingly.

React provides its own method setState(). setState() method takes a single parameter and expects an object which should contain the set of values to be updated. Once the update is done the method implicitly calls the render() method to repaint the page. Hence, the correct method of updating the value of a state will be similar to the code below.

State updates should be independent. The state object of a component may contain multiple attributes and React allows to use setState() function to update only a subset of those attributes as well as using multiple setState() methods to update each attribute value independently.

The only time we are allowed to define the state explicitly is in the constructor to provide the initial state.

Example

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

class Car extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      brand: "Ford",
      model: "Mustang",
      color: "red",
      year: 1964
    };
  }
  changeColor = () => {
    this.setState({color: "blue"});
  }
  render() {
    return (
      <div>
```

```
      <h1>My {this.state.brand}</h1>
      <p>
        It is a {this.state.color}
        {this.state.model}
        from {this.state.year}.
      </p>
      <button type="button"onClick={this.changeColor}>Change color</button>
    </div>
  );
 }
}

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

Output:

My Ford

It is a red Mustang from 1964.

Change color

Change state

My Ford

It is a blue Mustang from 1964.

Change color

Example

```
import React from 'react';
import ReactDOM from 'react-dom/client';
class App extends React.Component {
   constructor(props) {
      super(props);
      this.state = {
         count: 0,
      };
   }
increment = () =>{
         this.setState((prevState)=>({
         count: prevState.count+1
         }))
}
```

```
decrement = () =>{
        this.setState((prevState)=>({
        count: prevState.count-1
    }))
}
render() {
    return (
        <div>
            <h1>The current count is : {this.state.count}</h1>
            <button onClick={this.increment}>Increase</button>
            <button onClick={this.decrement}>Decrease</button>
        </div>
    );
}
}
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Output:

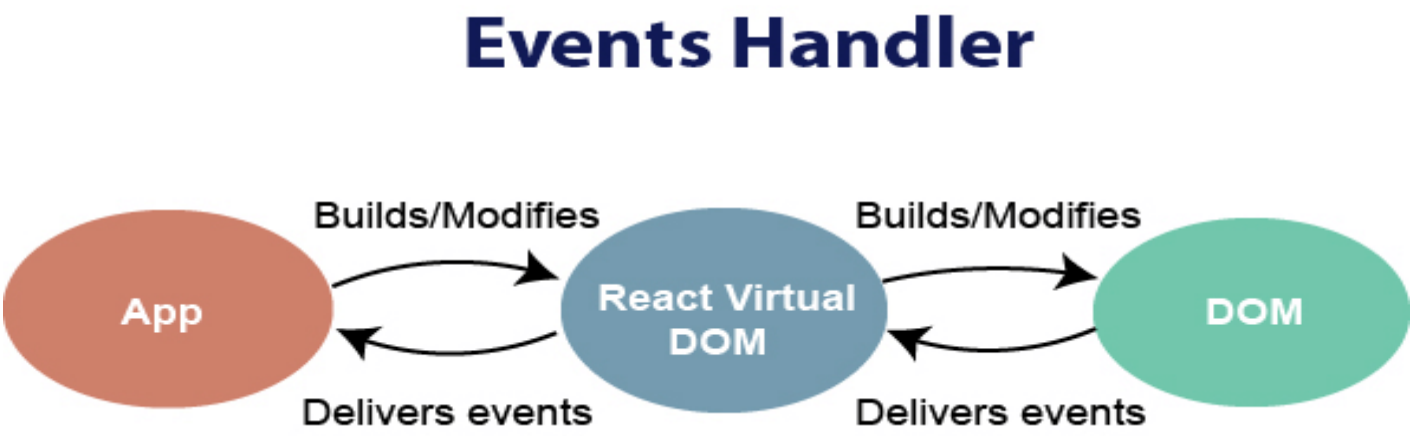
2.4 EVENTS

Each and every modern webpage that we create today tend to have user interactions. When the user interacts with the web application, events are fired. That event can be a mouse click, a keypress, or something rare like connecting the battery with a charger. From the developer side, we need to 'listen' to such events and then make our application respond accordingly. This is called event handling that provides a dynamic interface to a webpage. Like JavaScript DOM, React also provides us with some built-in methods to create a listener that responds accordingly to a specific event.

JavaScript has events to provide a dynamic interface to a webpage. These events are hooked to elements in the Document Object Model(DOM). These events by default use bubbling propagation i.e, upwards in the DOM from children to parent. We can bind events either as inline or in an external script.

An event is an action that could be triggered as a result of the user action or system generated event. For example, a mouse click, loading of a web page, pressing a key, window resizes, and other interactions are called events. React has its own event handling system which is very similar to handling events on DOM

elements. The react event handling system is known as Synthetic Events. The synthetic event is a cross-browser wrapper of the browser's native event.



Handling events with react have some syntactic differences from handling events on DOM. These are:

1. React events are named as **camelCase** instead of **lowercase**.
2. With JSX, a function is passed as the **event handler** instead of a **string**. For example:

Event declaration in plain HTML:

```
<button onclick="showMessage()">
        Hello JavaTpoint
</button>
```

Event declaration in React:

```
<button onClick={showMessage}>
    Hello JavaTpoint
</button>
```

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

React onChange Events:

What is the onChange Event Handler?

JavaScript allows us to listen to an input's change in value by providing the attribute onchange. React's version of the onchange event handler is the same, but camel-cased.

If you're using forms inside of a React component, it's a good idea to understand how the onChange event handler works with forms, state, and how you can pass the value to a function.

Example: Pass an Input Value from the onChange Event in a React Component

```
import React from 'react';
function App() {
  function handleChange(event) {
    console.log(event.target.value);
  }
  return (
    <input name="firstName" onChange={handleChange} />
  );
}
export default App;
```

An onChange event handler returns a Synthetic Event object which contains useful meta data such as the target input's id, name, and current value.

We can access the target input's value inside of the handleChange by accessing e.target.value. Therefore, to log the name of the input field, we can log e.target.name.

Example

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

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

```
      <p>
        It is a {this.state.color}
        {this.state.model}
        from {this.state.year}.
      </p>
      <button type="button"onClick={this.changeColor}  >Change color</button>
    </div>
  );
 }
}

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

Output:

My Ford

It is a red Mustang from 1964.

Change color

Change state

My Ford

It is a blue Mustang from 1964.

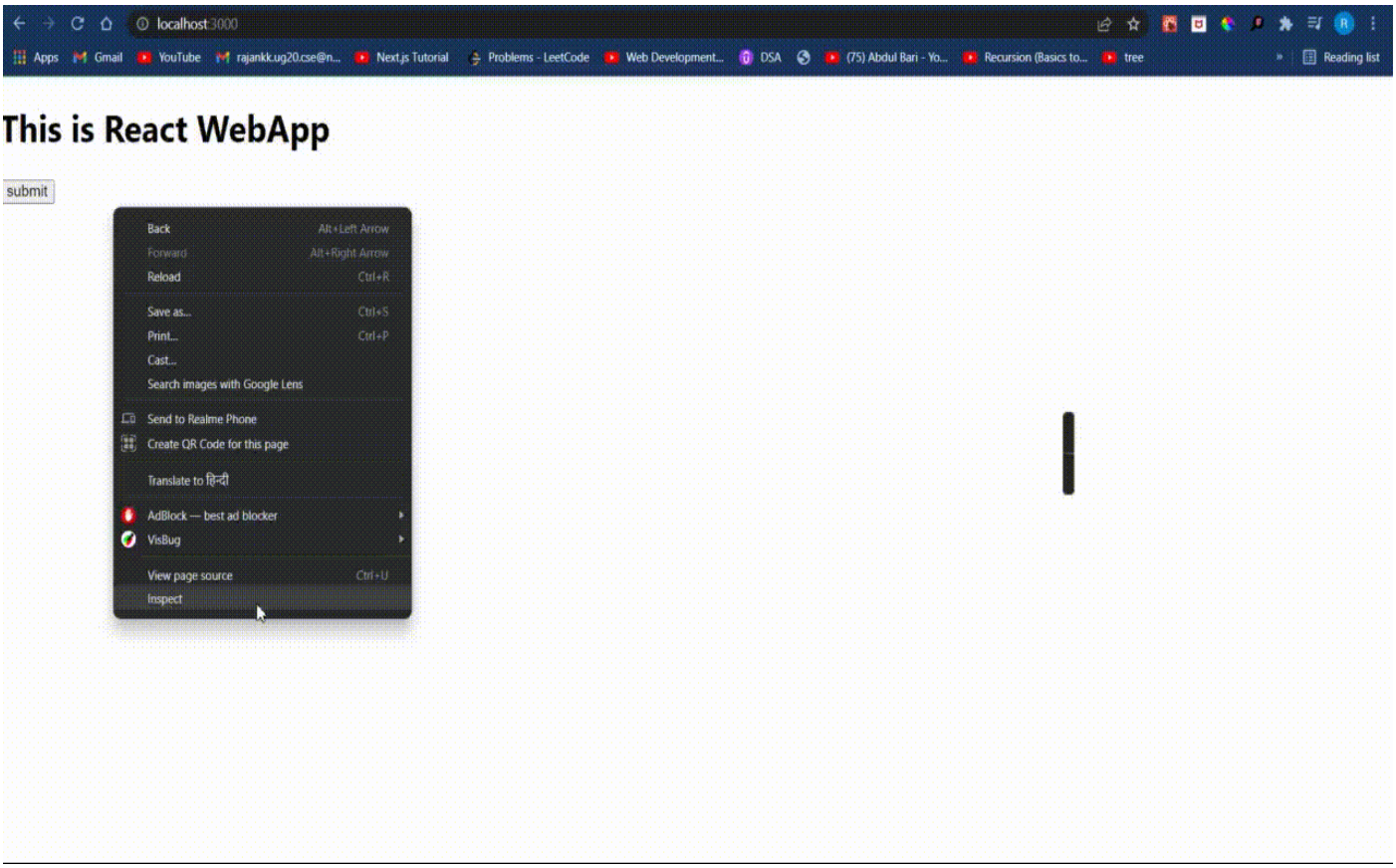
Change color

Example

```
import "./App.css";
function App() {
 const handleOnSubmit = (e) => {
  e.preventDefault();
  console.warn("You clicked on submit function")
 };
 return (
  <>
   <h1>This is React WebApp </h1>
   <form action="">
    <button type="submit" onClick={handleOnSubmit}>
     submit
    </button>
   </form>

  </>
 );
}
 export default App;
```

Output:


localhost:3000
Apps
Gmail
YouTube
rajankk.ug20.cse@n...
Next.js Tutorial
Problems - LeetCode
Web Development...
DSA
(75) Abdul Bari - Yo...
Recursion (Basics to...
tree
Reading list
This is React WebApp
submit
Back
Alt+Left Arrow
Forward
Alt+Right Arrow
Reload
Ctrl+R
Save as...
Ctrl+S
Print...
Ctrl+P
Cast...
Search images with Google Lens
Send to Realme Phone
Create QR Code for this page
Translate to हिन्दी
AdBlock — best ad blocker
VisBug
View page source
Ctrl+U
Inspect