UNIT IV ADVANCED CLIENT SIDE PROGRAMMING

React JS: React DOM - JSX - Components - Properties – Fetch API - State and Lifecycle - -JS Local Storage - Events - Lifting State Up - Composition and Inheritance.

REACTJS

- ReactJS is one of the most popular JavaScript front-end libraries which has a
- strong foundation and a large community.
- ReactJS is a declarative, efficient, and flexible JavaScript library for building reusable UI components.
- It is an opensource, component-based front end library which is responsible only for the view layer of the application. It was initially developed and maintained by Facebook and later used in its products like WhatsApp & Instagram

Why we use ReactJS?

The main objective of ReactJS is to develop User Interfaces (UI) that improves the speed of the apps. It uses virtual DOM (JavaScript object), which improves the performance of the app. The JavaScript virtual DOM is faster than the regular DOM. We can use ReactJS on the client and server-side as well as with other frameworks. It uses component and data patterns that improve readability and helps to maintain larger apps.

Advantage of ReactJS

1. Easy to Learn and USe
2. Creating Dynamic Web Applications Becomes Easier
3. Reusable Components
4. Performance Enhancement
5. The Support of Handy Tools
6. Known to be SEO Friendly
7. The Benefit of Having JavaScript Library
8. Scope for Testing the Codes

Disadvantage of ReactJS

1. The high pace of development
2. Poor Documentation
3. View Part
4. JSX as a barrier

REACT DOM

What is DOM?

DOM, abbreviated as Document Object Model, is a World Wide Web Consortium standard logical representation of any webpage. In easier words, DOM is a tree-like structure that contains all the elements and it's properties of a website as its nodes. DOM provides a language-neutral interface that allows accessing and updating of the content of any element of a webpage.

Before React, Developers directly manipulated the DOM elements which resulted in frequent DOM manipulation, and each time an update was made the browser had to recalculate and repaint the whole view according to the particular CSS of the page, which made the total process to consume a lot of time. As a betterment, React brought into the scene the virtual DOM. The Virtual DOM can be referred to as a copy of the actual DOM representation that is used to hold the updates made by the user and finally reflect it over to the original Browser DOM at once consuming much lesser time.

What is ReactDOM?

ReactDOM is a package that provides DOM specific methods that can be used at the top level of a web app to enable an efficient way of managing DOM elements of the web page. ReactDOM provides the developers with an API containing the following methods and a few more.

- render()
- findDOMNode()
- unmountComponentAtNode()
- hydrate()
- createPortal()

To use the ReactDOM in any React web app we must first import ReactDOM from the react-dom package by using the following code snippet:

import ReactDOM from 'react-dom'

render() Function

This is one of the most important methods of ReactDOM. This function is used to render a single React Component or several Components wrapped together in a Component or a div element. This function uses the efficient methods of React for updating the DOM by being able to change only a subtree, efficient diff methods, etc.

Syntax:

ReactDOM.render(element, container, callback)

Parameters: This method can take a maximum of three parameters as described below.

element: This parameter expects a JSX expression or a React Element to be rendered.

container: This parameter expects the container in which the element has to be rendered.

callback: This is an optional parameter that expects a function that is to be executed once the render is complete.

Return Type: This function returns a reference to the component or null if a stateless component was rendered.

findDOMNode() Function

This function is generally used to get the DOM node where a particular React component was rendered. This method is very less used like the following can be done by adding a ref attribute to each component itself.

Syntax:

ReactDOM.findDOMNode(component)

Parameters: This method takes a single parameter component that expects a React Component to be searched in the Browser DOM.

Return Type: This function returns the DOM node where the component was rendered on success otherwise null**.**

unmountComponentAtNode() Function

This function is used to unmount or remove the React Component that was rendered to a particular container. As an example, you may think of a notification component, after a brief amount of time it is better to remove the component making the web page more efficient.

Syntax:

ReactDOM.unmountComponentAtNode(container)

Parameters: This method takes a single parameter container which expects the DOM container from which the React component has to be removed.

Return Type: This function returns true on success otherwise false.

hydrate() Function

This method is equivalent to the render() method but is implemented while using server-side rendering.

Syntax:

ReactDOM.hydrate(element, container, callback)

Parameters: This method can take a maximum of three parameters as described below.

element: This parameter expects a JSX expression or a React Component to be rendered.

container: This parameter expects the container in which the element has to be rendered.

callback: This is an optional parameter that expects a function that is to be executed once the render is complete.

Return Type: This function attempts to attach event listeners to the existing markup and returns a reference to the component or null if a stateless component was rendered.

createPortal() Function

Usually, when an element is returned from a component's render method, it's mounted on the DOM as a child of the nearest parent node which in some cases may not be desired. Portals allow us to render a component into a DOM node that resides outside the current DOM hierarchy of the parent component.

Syntax:

ReactDOM.createPortal(child, container)

Parameters: This method takes two parameters as described below.

child: This parameter expects a JSX expression or a React Component to be rendered.

container: This parameter expects the container in which the element has to be rendered.

Return Type: This function returns nothing.

JSX

JSX- stands for JavaScript XML. It is simply a syntax extension of JavaScript.

It allows us to directly write HTML in React (within JavaScript code).

It is easy to create a template using JSX in React, but it is not a simple template language instead it comes with the full power of JavaScript.

It is faster than normal JavaScript as it performs optimizations while translating to regular JavaScript. Instead of separating the markup and logic in separated files,

React uses components for this purpose.

Syntax:

const element = <h1>Welcome to GeeksforGeeks.</h1>;

Characteristics of JSX:

- JSX is not mandatory to use there are other ways to achieve the same thing but using JSX makes it easier to develop react application.
- JSX allows writing expression in { }. The expression can be any JS expression or React variable.
- To insert a large block of HTML we have to write it in a parenthesis i.e, ().
- JSX produces react elements.
- JSX follows XML rule.
- After compilation, JSX expressions become regular JavaScript function calls.
- JSX uses camelcase notation for naming HTML attributes. For example, tabindex in HTML is used as tabIndex in JSX.

Advantages of JSX:

- JSX makes it easier to write or add HTML in React.
- JSX can easily convert HTML tags to react elements.
- It is faster than regular JavaScript.
- JSX allows us to put HTML elements in DOM without using appendChild() or createElement() method.

- As JSX is an expression, we can use it inside of if statements and for loops, assign it to variables, accept it as arguments, or return it from functions.
- JSX prevents XSS (cross-site-scripting) attacks popularly known as injection attacks.
- It is type-safe, and most of the errors can be found at compilation time.

Disadvantages of JSX:

- JSX throws an error if the HTML is not correct.
- In JSX HTML code must be wrapped in one top-level element otherwise it will give an error.
- If HTML elements are not properly closed JSX will give an error.

Example

Here, we will write JSX syntax in JSX file and see the corresponding JavaScript code which transforms by preprocessor (babel).

JSX File

 Hello JavaTpoint

Corresponding JSX file is:

React.createElement("div", null, "Hello JavaTpoint");

The above line creates a react element and passing three arguments inside where the first is the name of the element which is div, second is the attributes passed in the div tag, and last is the content you pass which is the "Hello JavaTpoint."

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.

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.

App.JSX

```
1. import React, { Component } from 'react';
2. class App extends Component{
3. render(){
4. return(
5. <div>
6.  JavaTpoint
7. <h2> Training Institutes</h2>
8. <h2> This website contains the best CS tutorials. </h2>
9. </div>
10. );
 11. }
12.}
13.export default App;
```

REACT 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.

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.

In ReactJS, we have mainly two types of components. They are

1. Functional Components
2. Class Components

Functional Components

In React, 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. We can create a function that takes props(properties) as input and returns what should be rendered. A valid functional component can be shown in the below example.

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

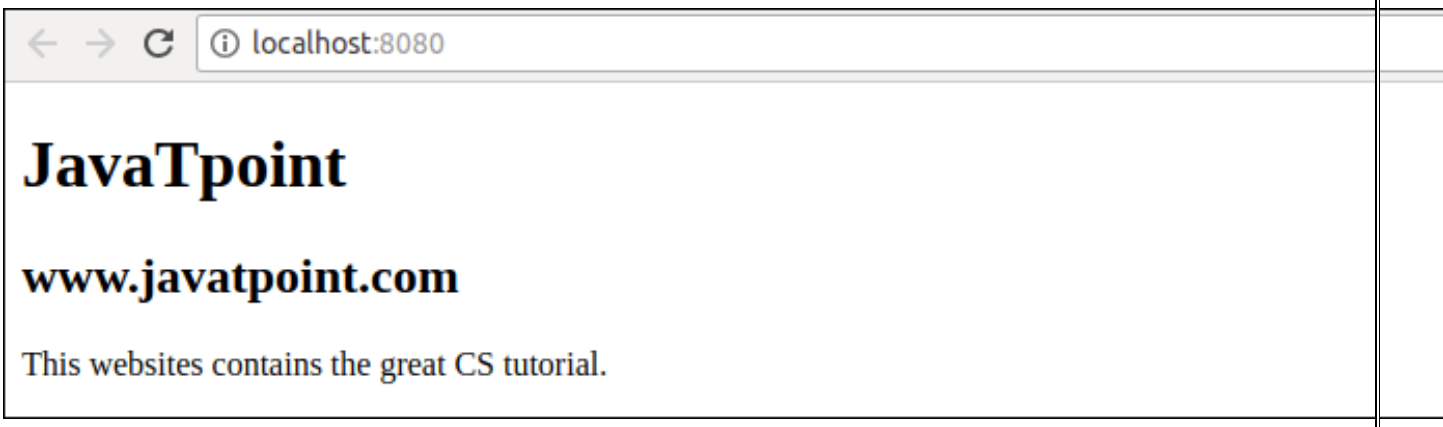
The functional component is also known as a stateless component because they do not hold or manage state.

Example

```
import React, { Component } from 'react';
class App extends React.Component {
   render() {
      return (
         <div>
            <First/>
            <Second/>
         </div>
      );
   }
}
class First extends React.Component {
   render() {
      return (
         <div>
            <h1>JavaTpoint</h1>
         </div>
      );
   }
}
class Second extends React.Component {
   render() {
      return (
         <div>
            <h2>www.javatpoint.com</h2>
            <p>This websites contains the great CS tutorial.</p>
         </div>
```

```
28.       );
29.    }
30. }
31. export default App;
```

Output:



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.

```
1. class MyComponent extends React.Component {
2.   render() {
3.     return (
4.       <div>This is main component.</div>
5.     );
6.   }
7. }
```

The class component is also known as a stateful component because they can hold or manage local state.

Example

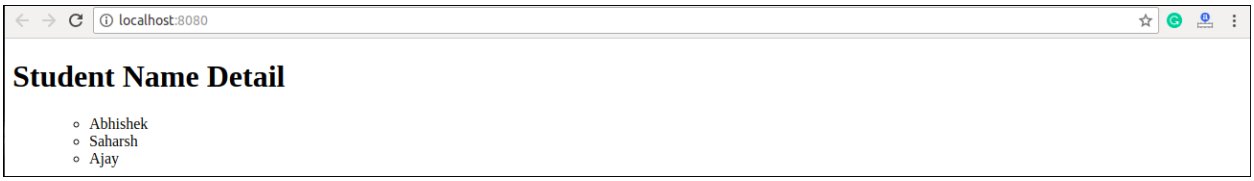
In this example, we are creating the list of unordered elements, where we will dynamically insert StudentName for every object from the data array. Here, we are using ES6 arrow syntax (=>) which looks much cleaner than the old JavaScript syntax. It helps us to create our elements with fewer lines of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

we need to create a list with a lot of items.

```
1.  import React, { Component } from 'react';
2.  class App extends React.Component {
3.   constructor() {
4.       super();
5.       this.state = {
6.          data:
7.          [
8.            {
9.              "name":"Abhishek"
10.           },
11.           {
12.             "name":"Saharsh"
13.           },
14.           {
15.             "name":"Ajay"
16.           }
17.         ]
18.     }
19.  }
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;
```

Output:



REACT PROPS

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.

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.

To send props into a component, use the same syntax as HTML attributes:

Example

Add a "brand" attribute to the Car element:

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

The component receives the argument as a props object:

Example

Use the brand attribute in the component:

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

Pass Data

Props are also how you pass data from one component to another, as parameters.

Example

Send the "brand" property from the Garage component to the Car component:

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

FETCH API

The Fetch API provides the fetch() method defined on a window object. This is used to perform requests. This method returns a Promise which can be further used to retrieve response of the request.

Basic Syntax:

```
fetch(url) //call the fetch function passing the url of the API as a parameter
.then(function(){
  //code for handling data from API
});
.catch(function(){
  //code when the server returns any error
});
```

By default, fetch() will not send or receive any cookies from the server, resulting in unauthenticated requests.

Below are list of methods which can be used when we get a response on what we want to do with the information:

1. clone(): for creating a clone of response.
2. redirect(): for creating strong response with different url.
3. arrayBuffer(): we return a Promise that resolves with an ArrayBuffer.
4. formData(): also returns a Promise that resolves with a FormDataObject.
5. blob(): same as above only that resolves with a blob.
6. text(): this resolves with a string.
7. json(): it resolves promise with JSON.

Request:

A Request object represents the request part of a fetch call. By passing fetch a Request you can make advanced and customized requests:

method: GET, POST, PUT

url: URL of the request

headers: Headers object

referrer: referrer of the request

mode: cors, no-cors, same-origin

credentials: should cookies go with the request? omit, same-origin

cache: cache mode (default, reload, no-cache)

LOADING JSON

We cannot block the user interface waiting until the request completes. That is why fetch() method returns a Promise. A promise is actually an object which represents a future result. The then() method is used to wait for the response from the server-side and log it to the console.

Below code snippet explains the above logic:

```
fetch('https://www.reddit.com/r/javascript/top/.json?limit=5')
.then(res=>res.json())
.then(json=>console.log(json));
```

Async…await

This provides a more concise way to process Promises. Its functionality is that we can mark a function as async, then wait for the promise to finish with the await, and access the result as a normal object.

Simple example demonstrating how async…await can be used:

```
async function demo(subject){
const URL='https://www.reddit.com/r/javascript/top/.json?limit=5';
const Res= fetch(URL);
const response= await Res;
const json= await response.json();
console.log(json);
}
demo('javascript');
```

HANDLING ERRORS:

What if we ask the server for a non-existing resource that requires permissions/authorization? With fetch(), we can handle application-level errors like 404 responses.

Although Fetch API is superseding XMLHttpRequest still a beginner needs to learn XMLHttpRequest to make more effective use of Fetch API.

STATE IN REACTJS

The state is a built-in React object that is used to contain data or information about the component. A component's state can change over time; whenever it changes, the component re-renders. The change in state can happen as a response to user action or system-generated events and these changes determine the behavior of the component and how it will render.

```
class Greetings extends React.Component {
  state = {
    name: "World"
  };
  updateName() {
    this.setState({ name: "Simplilearn" });
  }
  render() {
      return(
          <div>
              {this.state.name}
          </div>
      )
  }
}
```

- A state can be modified based on user action or network changes
- Every time the state of an object changes, React re-renders the component to the browser
- The state object is initialized in the constructor
- The state object can store multiple properties
- this.setState() is used to change the value of the state object
- setState() function performs a shallow merge between the new and the previous state

The setState() Method

State can be updated in response to event handlers, server responses, or prop changes. This is done using the setState() method. The setState() method enqueues all of the updates made to the component state and instructs React to re-render the component

and its children with the updated state.

Always use the setState() method to change the state object, since it will ensure that the component knows it's been updated and calls the render() method.

How it is implemented in a React web application.

```
class Bike extends React.Component {
  constructor(props) {
    super(props);
     this.state = {
      make: "Yamaha",
      model: "R15",
      color: "blue"
    };
  }
  changeBikeColor = () => {
    this.setState({color: "black"});
  }
  render() {
    return (
      <div>
        <h2>My {this.state.make}</h2>
        <p>
          It is a {this.state.color}
          {this.state.model}.
        </p>
        <button
          type="button"
          onClick={this.changeBikeColor}
        >Change color</button>
      </div>
    );
  }
```

```
}
```

State vs. Props

	State	Props
Use Case	State is used to store the data of the components that have to be rendered to the view	Props are used to pass data and event handlers to the children components
✍Mutability	State holds the data and can change over time	Props are immutable—once set, props cannot be changed
Component	State can only be used in class components	Props can be used in both functional and class components
Updation	Event handlers generally update state	The parent component sets props for the children components

REACT LIFECYCLE

Lifecycle of Components

Each component in React has a lifecycle which you can monitor and manipulate during its three main phases.

The three phases are: **Mounting**, **Updating**, and **Unmounting**.

Mounting

Mounting means putting elements into the DOM.

React has four built-in methods that gets called, in this order, when mounting a component:

1. constructor()
2. getDerivedStateFromProps()
3. render()
4. componentDidMount()

The render() method is required and will always be called, the others are optional and will be called if you define them.

1. **constructor**

The constructor() method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state and other initial values.

The constructor() method is called with the props, as arguments, and you should always start by calling the super(props) before anything else, this will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).

Example:

The constructor method is called, by React, every time you make a component:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}
ReactDOM.render(<Header />, document.getElementById('root'));
```

2. **getDerivedStateFromProps**

The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM.

This is the natural place to set the state object based on the initial props.

It takes state as an argument, and returns an object with changes to the state.

The example below starts with the favorite color being "red", but

the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute:

Example:

The getDerivedStateFromProps method is called right before the render method:

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));
```

3. render

The render() method is required, and is the method that actually outputs the HTML to the DOM.

```
class Header extends React.Component {
  render() {
    return (
      <h1>This is the content of the Header component</h1>
    );
  }
}
ReactDOM.render(<Header />, document.getElementById('root'));
```

4. componentDidMount

The componentDidMount() method is called after the component is rendered.

This is where you run statements that requires that the component is already placed in the DOM.

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({favoritecolor: "yellow"})
    }, 1000)
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}
ReactDOM.render(<Header />, document.getElementById('root'));
```

Updating

The next phase in the lifecycle is when a component is updated. A component is updated whenever there is a change in the component's state or props. React has five built-in methods that gets called, in this order, when a component is updated:

1. getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

The render() method is required and will always be called, the others are optional and will be called if you define them.

getDerivedStateFromProps

Also at updates the getDerivedStateFromProps method is called. This is the first method that is called when a component gets updated.

shouldComponentUpdate

In the shouldComponentUpdate() method you can return a Boolean value that specifies whether React should continue with the rendering or not.

The default value is true. render

The render() method is of course called when a component gets updated, it has to re-render the HTML to the DOM, with the new changes.

getSnapshotBeforeUpdate

In the getSnapshotBeforeUpdate() method you have access to
the props and state before the update, meaning that even after the update, you can check what the values were before the update.

If the getSnapshotBeforeUpdate() method is present, you should also include the componentDidUpdate() method, otherwise you will get an error.

componentDidUpdate

The componentDidUpdate method is called after the component is updated in the DOM

Unmounting

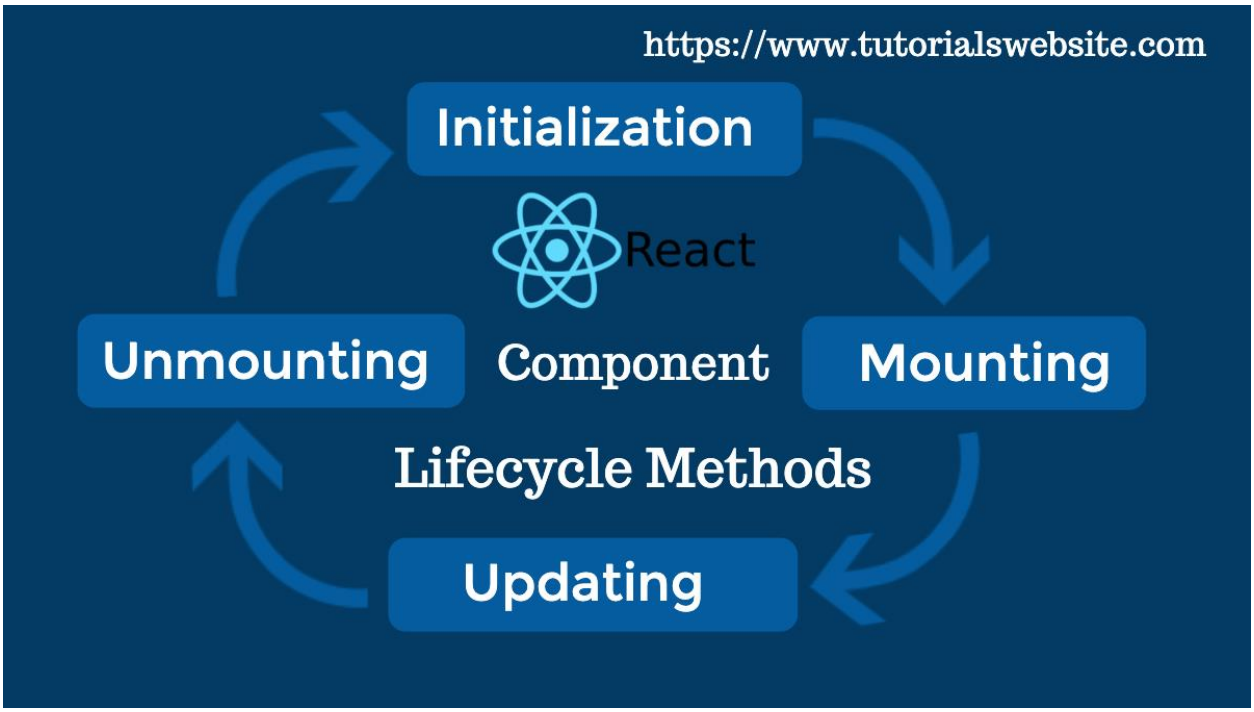
The next phase in the lifecycle is when a component is removed from the DOM, or unmounting as React likes to call it.

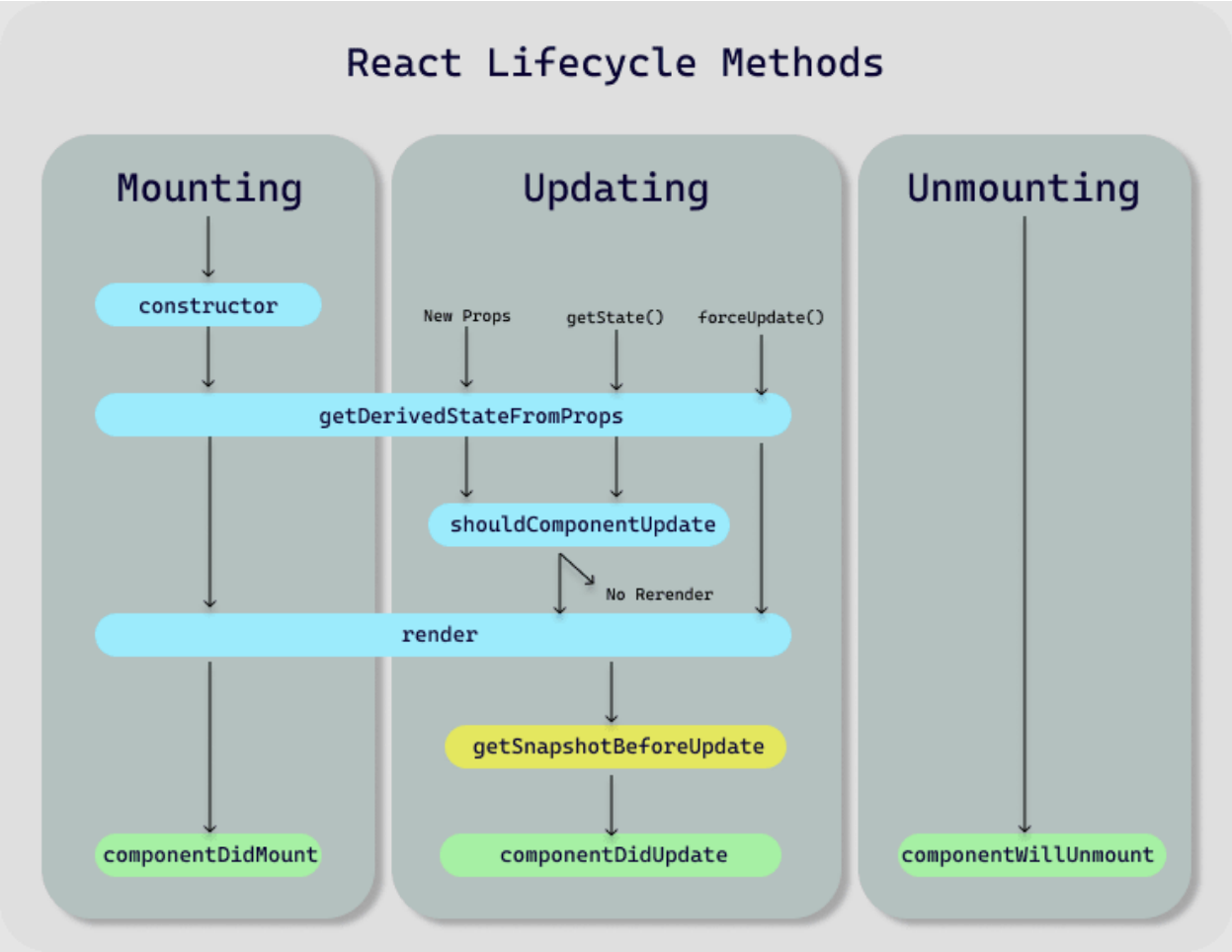
React has only one built-in method that gets called when a component is unmounted: componentWillUnmount()

componentWillUnmount

The componentWillUnmount method is called when the component is about to be removed from the DOM.

It takes state as an argument, and returns an object with changes to the state.


https://www.tutorialswebsite.com
Initialization
React
Unmounting
Component
Mounting
Lifecycle Methods
Updating


React Lifecycle Methods
Mounting
constructor
componentDidMount
Updating
New Props
getState()
forceUpdate()
getDerivedStateFromProps
shouldComponentUpdate
No Rerender
render
getSnapshotBeforeUpdate
componentDidUpdate
Unmounting
componentWillUnmount

JS LOCALSTORAGE

- LocalStorage is a data storage type of web storage. This allows the JavaScript sites and apps to store and access the data without any expiration date. This means that the data will always be persisted and will not expire. So, data stored in the browser will be available even after closing the browser window.
- Local storage is a web storage object to store the data on the user's computer locally, which means the stored data is saved across browser sessions and the data stored has no expiration time.
- As an alternative to cookies, localStorage has much cleaner support in JavaScript and is very easy to work with.
- cookies were the only option to remember this type of temporary and local information, but now we have localStorage as well. Local storage comes with a higher storage limit than cookies (5MB vs 4MB). It also does not get sent with every HTTP request. So, it is a better choice now for client-side storage

Syntax

// To store data

localStorage.setItem('Name', 'Rahul');

// To retrieve data

localStorage.getItem('Name');

// To clear a specific item

localStorage.removeItem('Name');

// To clear the whole data stored in localStorage

localStorage.clear();

Example

It is a basic example of adding a key and value to localStorage and retrieving back by the key. See the code below how localStorage methods work:

```
<html>
<body>
```

```
<div id="result"></div>

<script>
// Check browser support
if (typeof(Storage) !== "undefined") {
  // Store an item to localStorage
.  localStorage.setItem("firstname", "KAMALI");
.
.  // Retrieve the added item
.  document.getElementById("result").innerHTML = localStorage.getItem("firstname");
.
.} else {
.  //display this message if browser does not support localStorage
.  document.getElementById("result").innerHTML = "Sorry, your browser does not suppor
t Web Storage.";
.}
. </script>
. </body>
. </html>
```

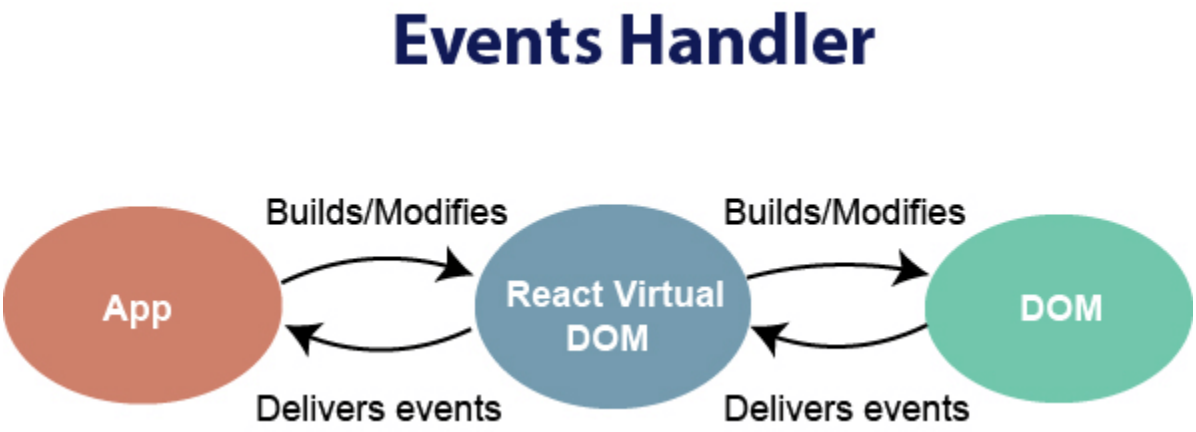
OUTPUT:

KAMALI

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 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>
```

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

In plain HTML, to prevent the default link behavior of opening a new page, we can write:

```
<a href="#" onclick="console.log('You had clicked a Link.'); return false">
    Click_Me
</a>
```

In React, we can write it as:

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

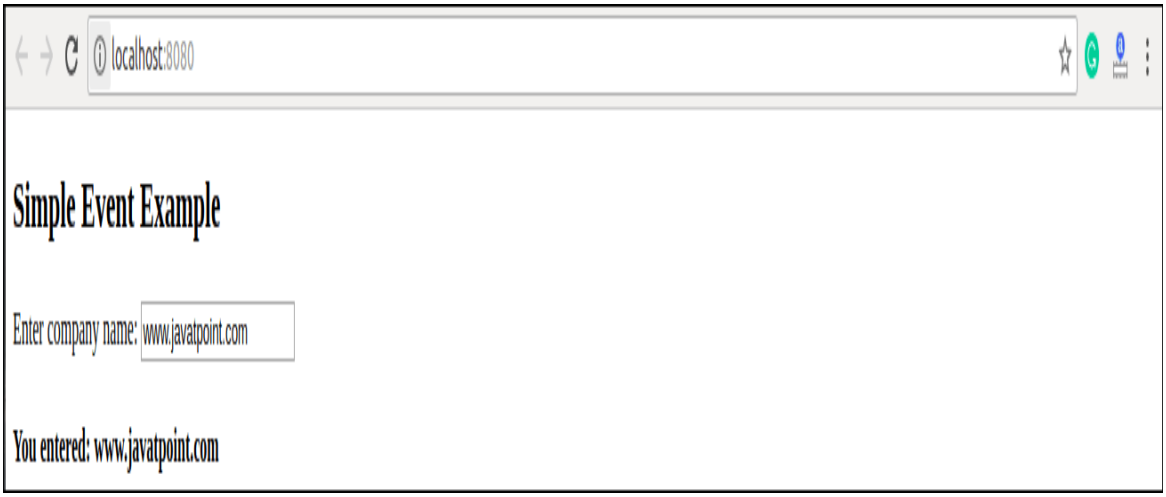
In the above example, e is a **Synthetic Event** which defines according to the **W3C** spec.

Example

In the below example, we have used only one component and adding an onChange event. This event will trigger the **changeText** function, which returns the company name.

```
import React, { Component } from 'react';
class App extends React.Component {
   constructor(props) {
      super(props);
      this.state = {
         companyName: ''
      };
   }
   changeText(event) {
.      this.setState({
.         companyName: event.target.value
.      });
.   }
.   render() {
.      return (
.         <div>
.            <h2>Simple Event Example</h2>
.            <label htmlFor="name">Enter company name: </label>
.            <input type="text" id="companyName" onChange={this.changeText.bind(this
)}/>
.            <h4>You entered: { this.state.companyName }</h4>
.         </div>
.      );
.   }
.}
. export default App;
```

Output



LIFTING THE STATE UP IN REACTJS

- Lifting up the State : Every component in React has its own state. Because of this sometimes data can be redundant and inconsistent. So, by Lifting up the state we make the state of the parent component as a single source of truth and pass the data of the parent in its children.

- Time to use Lift up the State: If the data in "parent and children components" or in "cousin components" is Not in Sync

- Often there will be a need to share state between different components. The common approach to share state between two components is to move the state to common parent of the two components. This approach is called as lifting state up in React.js

- With the shared state, changes in state reflect in relevant components simultaneously. This is a single source of truth for shared state components.

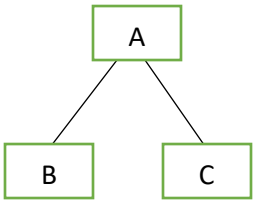
- If we see this Web Page, it has three sections.
  1. Product Information Section where User can select what is the product user is Ordering and the Quantity number required.
  2. Address Section where User can enter the Delivery Address.

3. Summary Section. We display the summary details of the previous two sections here.

When a User Enters the Product details and Address details we display that information in the summary section. But assuming that we have a textbox available where User Can change the product quantity even in the Summary Section. Because just to change the Product quantity, we don't want User to navigate all the way back to Product Information section. If we change the Quantity in Summary section, we want that change to be displayed in Product Information section and vice versa like how it is happening in this Web Page.

Example 1: If we have 2 components in our App. A -> B where, A is parent of B. keeping the same data in both Component A and B might cause inconsistency of data.

Example 2: If we have 3 components in our App.



Where A is the parent of B and C. In this case, If there is some Data only in component B but, component C also wants that data. We know Component C cannot access the data because a component can talk only to its parent or child (Not cousins).

Approach: To solve this, we will Lift the state of component B and component C to component A. Make A.js as our Main Parent by changing the path of App in the index.js file

Before:

import App from './App';

After:

import App from './A';

Filename- A.js:

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

import B from './B'

import C from './C'

class A extends Component {

  constructor(props) {

    super(props);

    this.handleTextChange = this.handleTextChange.bind(this);

    this.state = {text: ''};

  }

  handleTextChange(newText) {
```

```
    this.setState({text: newText});
  }
  render() {
    return (
        <React.Fragment>
          <B text={this.state.text}
             handleTextChange={this.handleTextChange}/>
          <C text={this.state.text} />
        </React.Fragment>
    );
  }
}
export default A;
```

Filename- B.js:

```
import React,{ Component } from 'react';
class B extends Component {
  constructor(props) {
```

```
    super(props);
    this.handleTextChange = this.handleTextChange.bind(this);
}
handleTextChange(e){
    this.props.handleTextChange(e.target.value);
}
  render() {
    return (
        <input value={this.props.text}
              onChange={this.handleTextChange} />
    );
}}    export default B;
```

Filename- C.js:

```
   class C extends Component {
render() {
    return (
        <h3>Output: {this.props.text}</h3>
```

```
  );}}

export default C;
```

Output: Now, component C can Access text in component B through component A.

COMPOSITION AND INHERITANCE

- Composition and inheritance are the approaches to use multiple components together in React.js.
- This helps in code reuse.
- React recommend using composition instead of inheritance as much as possible and inheritance should be used in very specific cases only.

Composition

- In Object-Oriented Programming, composition is a well-known concept. It describes a class that can refer to one or more objects of another class as instances rather than inheriting properties from a base class.
- For example, the composition can be used for building a car's engine.

Inheritance

- Inheritance is an Object-Oriented Programming concept in JavaScript that allows us to inherit the features of a parent from the child.
- For example, suppose we can design a parent that looks like a vehicle and has properties like wheels, engine, gearbox and lights.
- Then, as a vehicle child, we may make a car that inherits the vehicle's(Parent) properties.
- It means that the car(Child) is a vehicle that will have wheels, lights, an engine, and a gearbox from the start.
- We can also extend the functionality of our object(here, car) by adding more features.

Inheritance

```
class UserNameForm extends React.Component {
  render() {
    return (
      <div>
        <input type="text" />
      </div>
    );
  }
}
ReactDOM.render(
  < UserNameForm />,
  document.getElementById('root'));
```

This sis simple to just input the name. We will have two more components to create and update the username field.

With the use of inheritance we will do it like −

```
class UserNameForm extends React.Component {
  render() {
    return (
      <div>
        <input type="text" />
      </div>
    );
  }
}
class CreateUserName extends UserNameForm {
  render() {
    const parent = super.render();
    return (
      <div>
        {parent}
        <button>Create</button>
      </div>
    )
```

```
  }
}
class UpdateUserName extends UserNameForm {
  render() {
    const parent = super.render();
    return (
      <div>
        {parent}
        <button>Update</button>
      </div>
    )
  }
}
ReactDOM.render(
  (<div>
    < CreateUserName />
    < UpdateUserName />
  </div>), document.getElementById('root')
);
```

We extended the UserNameForm component and extracted its method in child component using super.render();

Composition

```
class UserNameForm extends React.Component {
  render() {
    return (
      <div>
        <input type="text" />
      </div>
    );
  }
}
class CreateUserName extends React.Component {
  render() {
    return (
```

```
        <div>
          < UserNameForm />
          <button>Create</button>
        </div>
      )
    }
}
class UpdateUserName extends React.Component {
  render() {
    return (
      <div>
        < UserNameForm />
        <button>Update</button>
      </div>
    )
  }
}
ReactDOM.render(
  (<div>
    <CreateUserName />
    <UpdateUserName />
  </div>), document.getElementById('root')
);
```

Use of composition is simpler than inheritance and easy to maintain the complexity.