UNIT-I: Introduction to React JS

Overview of frameworks:

React.js is a **JavaScript library** created by Facebook to build user interfaces, specifically for web applications. Here's a simplified breakdown:

Key Features:

Component-Based: React breaks the UI into smaller parts called components. Each component can work on its own and manage its own small part of the UI, making it easy to build, maintain, and update complex web applications.

Example: A webpage could have a header, footer, and content section, each being a separate component.

JSX (JavaScript XML): React uses a special syntax called JSX, which looks like HTML but is actually written in JavaScript. This makes it easier for developers to create dynamic and interactive UIs.

Example: You can write something that looks like HTML (<h1>Hello, World!</h1>) inside JavaScript code.

Virtual DOM (Document Object Model): React keeps a virtual copy of the DOM in memory. When something changes, React updates only the parts of the page that need it, rather than reloading the entire page. This makes React apps faster and more efficient.

Example: If a user clicks a button to change text on a page, React will only update the text instead of reloading the entire page.

One-Way Data Flow: React ensures that data flows in one direction, making it easier to understand and debug the application. This structure simplifies how components communicate and how data is passed around.

Example: If you enter a value in a form, the data flows from the form to the component managing it.

Reusable Components: You can reuse components across different parts of your application, saving time and reducing code duplication.

Example: You can create a button component and use it in multiple places without having to write the button's code again.

In Short:

React helps developers build fast, interactive, and modular web applications by breaking everything into small, reusable components, managing the interface efficiently with its Virtual DOM, and making code more organized through JSX.

NPM (Node Package Manager) is a tool that helps developers manage packages (reusable code libraries) and dependencies in JavaScript projects. It comes bundled with **Node.js** and allows you to easily install, update, or remove packages and run scripts.

Here's a breakdown of common NPM commands:

1. **npm init**

- **Usage**: npm init
- **What It Does**: Initializes a new Node.js project. It creates a package.json file where all your project's settings, dependencies, and scripts are stored.
- **Example**: If you're starting a new project, you'd run npm init to set it up this will ask a series of questions (like project name, version, etc.) to configure the package.json.

2. **npm install (or npm i)**

- **Usage**: npm install <package-name> or npm i <package-name>
- **What It Does**: Installs a package from the NPM registry into your project. By default, it will save the package as a dependency in your package.json.
- **Example**: To install a package like React, run:

3. npm start

- **Usage**: npm start
- **What It Does**: Runs the start script specified in the package.json. This is commonly used to start a development server or build process.

4. npm update

Usage: npm update

What It Does: Updates all the installed packages to their latest versions based on the constraints in the package.json.

5. npm list

Usage: npm list

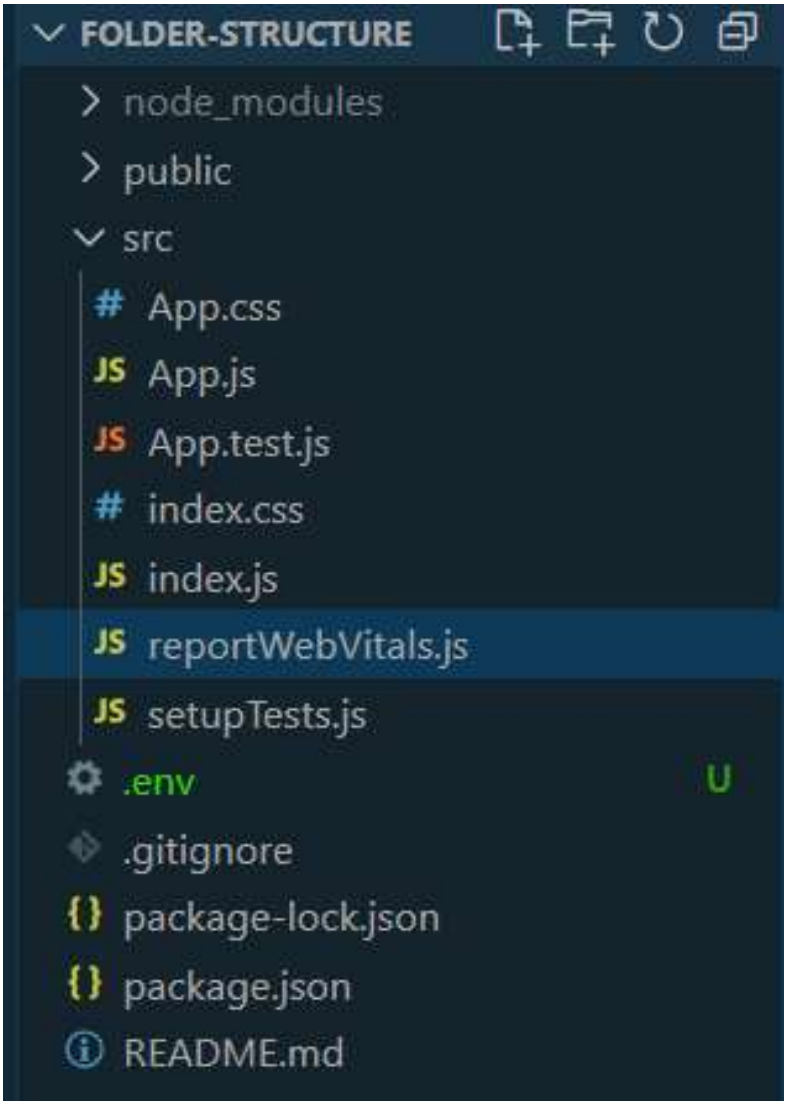
What It Does: Lists all installed packages and their versions in the current project.

A **React app** is a web application built using **React**, a popular JavaScript library developed by Facebook for building user interfaces. React focuses on creating interactive and dynamic user interfaces efficiently by breaking down the UI into reusable components.

Key Features of a React App:

1. **Component-Based Architecture**: The UI is divided into small, reusable components that manage their own state and can be composed to build complex interfaces.
    - Example: A button, a form, and a navigation bar can each be a separate component.
2. **Declarative Syntax**: React allows you to describe what the UI should look like for a given state, and it takes care of updating the UI when the state changes.
    - Example: You declare a component's appearance based on its state and React updates the DOM efficiently when the state changes.
3. **JSX (JavaScript XML)**: React uses JSX, a syntax extension that allows you to write HTML-like code within JavaScript. This makes the code easier to read and write.
    - Example: <button onClick={handleClick}>Click Me</button>
4. **Virtual DOM**: React maintains a virtual representation of the actual DOM to optimize updates and rendering. It only updates the parts of the DOM that have changed, improving performance.
    - Example: If you update a piece of state, React calculates the difference between the current and previous virtual DOM and applies only the necessary changes to the real DOM.
5. **State Management**: React components manage their own state and can pass data and state between components through props, enabling interactive and dynamic features.
    - Example: A form component can manage its own input values and pass the data to a parent component.
6. **React Hooks**: Modern React features that allow functional components to manage state and side effects, providing a more concise and functional approach to writing components.
    - Example: useState hook for managing component state and useEffect for handling side effects like data fetching.

Project Structure:



Folder structure contains the files and folders that are present in the directory. There are multiple files and folders, for example, Components, Utils, Assets, Styles, Context, Services, Layouts, etc.

Steps to Create Folder Structure:

Step 1: Open the terminal, go to the path where you want to create the project and run the command with the project name to create the project.

```
npx create-react-app folder-structure
```

Step 2: After project folder has been created, go inside the folder using the following command.

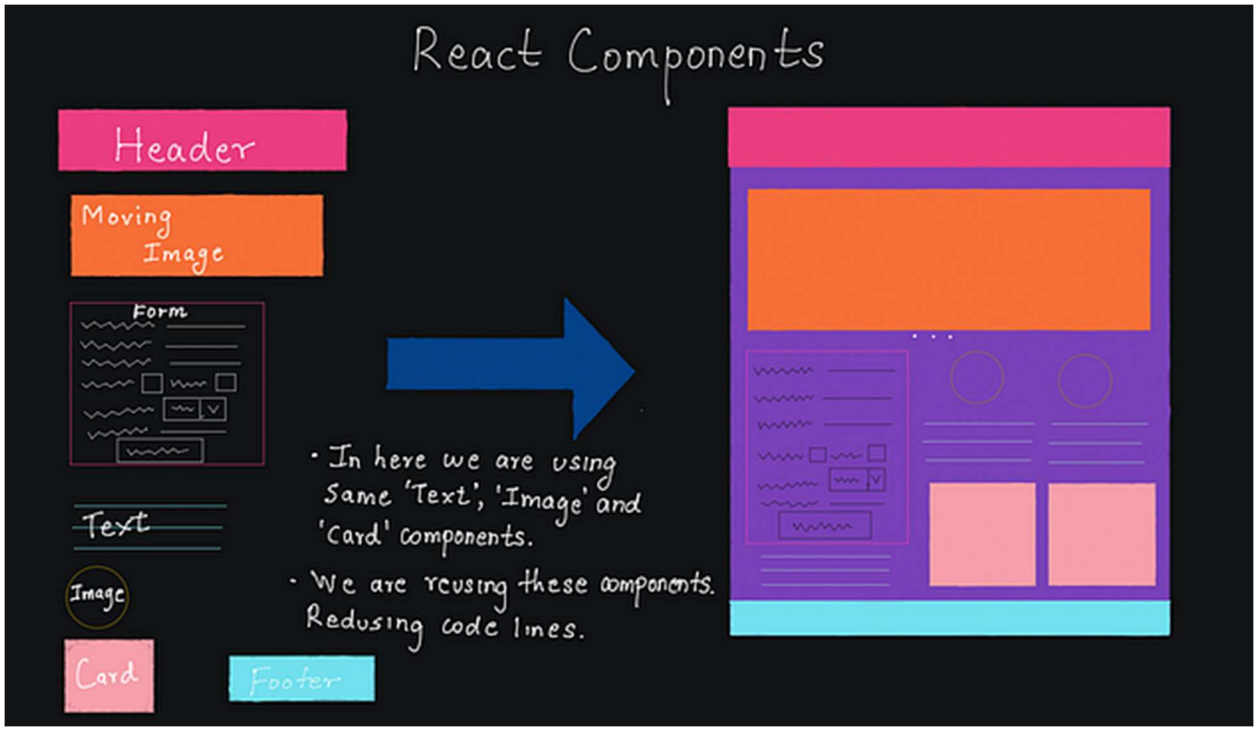
```
cd folder-structure
```

Why project structure react is important?

Project structure shows the organization of code and files, and a clean, simple, and decent project structure shows a clean written code, which helps to debug code, and another developer can easily read it. Also, while deploying code on the server, it can recognize files easily, which is why developers are implementing clean, simple, and decent project structures.

Best Practices for React Project Structure

It is best practice to create a project structure for the React application, separate files according to their work, and separate them in the correct directories. For example, single components which can be used multiple places should be present in the components folder, and also standardize the naming conventions of the application so that it will be easy to verify the logic and presence of any particular file or folder. Creating a well-organized folder structure is important for maintaining a React project. It increases readability, maintainability, and scalability of the application. By adopting a well structure, it will be easy to manage code.



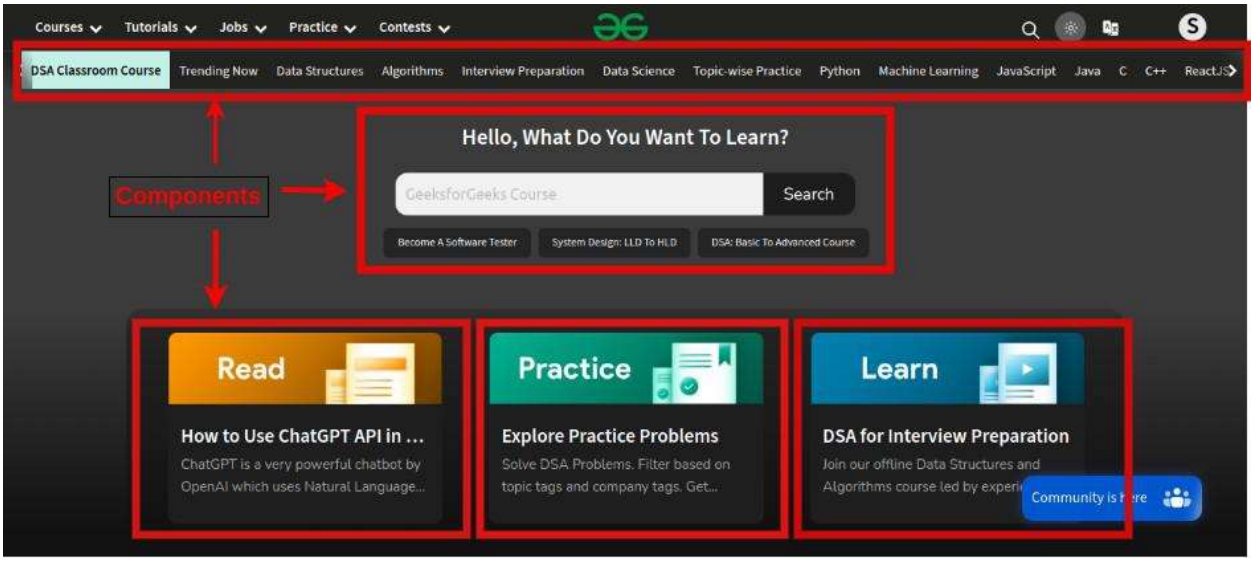
Components in React *serve as independent and reusable code blocks for UI elements. They represent different parts of a web page and contain both* ***structure*** *and* ***behavior****. They are similar to* ***JavaScript functions*** *and make creating and managing complex user interfaces easier by breaking them down into smaller, reusable pieces.*

What are React Components?

React Components are the building block of React Application. They are the reusable code blocks containing logics and and UI elements. They have the same purpose as JavaScript functions and return HTML. Components make the task of building UI much easier.

A UI is broken down into multiple individual pieces called components. You can work on components independently and then merge them all into a parent component which will be your final UI.

Components promote efficiency and scalability in web development by allowing developers to compose, combine, and customize them as needed.



Types of Components in React:

In React, we mainly have two types of components:

1. Functional Components

2. Class based Components

Functional components are just like JavaScript functions that accept properties and return a React element.
We can create a functional component in React by writing a JavaScript function. These functions may or may not receive data as parameters.
The below example shows a valid functional component in React:

Syntax:

```
function demoComponent() {
    return (<h1>
                Welcome Message!
            </h1>);
}
```

Example: Create a function component called welcome.

The class components are a little more complex than the functional components. A class component can show **inheritance** and access data of other components.
Class Component must include the line "**extends React.Component**" to pass data from one class component to another class component. We can use JavaScript ES6 classes to create class-based components in React.

Syntax:

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

The below example shows a valid class-based component in React:

Example: Create a class component called welcome.

```
class Welcome extends Component {
    render() {
        return <h1>Hello, Welcome to GeeksforGeeks!</h1>;
    }
}
```

The components we created in the above two examples are equivalent, and we also have stated the basic difference between a functional component and a class component.

Functional Component vs Class Component

A functional component is best suited for cases where the component doesn't need to interact with other components or manage complex states. Functional components are ideal for **presenting static UI elements** or composing multiple simple components together under a single parent component.
While class-based components can achieve the same result, they are generally **less efficient** compared to functional components. Therefore, it's recommended to not use class components for general use.

Aspect	Functional Components	Class Components
Definition	A simple JavaScript pure function that takes props and returns a React element (JSX).	Must extend from `React.Component` and define a `render()` method that returns a React element.
Render Method	No `render` method is used. JSX is returned directly from the function.	A `render()` method is required to return JSX, similar to HTML in syntax.
Execution	Executes from top to bottom. Once the function is returned, it's done with its execution.	Instantiates a class instance with lifecycle methods that can be invoked at different component phases.
State Management	Initially known as stateless, but can now manage state using hooks like `useState`.	Known as stateful components and manage their own state traditionally through `this.state`.
Lifecycle Methods	Cannot use lifecycle methods but can mimic them using hooks like `useEffect`.	Can use lifecycle methods such as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount`.
Hooks Usage	Implements hooks easily within the function body to handle state and effects.	Does not use hooks. State and lifecycle logic are handled differently through class methods.
Constructor	Does not require a constructor. Hooks handle state initialization and effects without it.	Requires a constructor for initializing state and binding event handlers.
Efficiency	More efficient due to less boilerplate and direct usage of hooks for state and effects.	Slightly less efficient. Can have more boilerplate code due to lifecycle methods and state management.
Code Complexity	Requires fewer lines of code, leading to simpler, more readable components.	Typically requires more code, which can lead to more complexity and verbosity.

Rendering React Components

Rendering Components means turning your component code into the UI that users see on the screen.
React is capable of rendering user-defined components. To render a component in React we can initialize an element with a user-defined component and pass this element as the first parameter to **ReactDOM.render()** or directly pass the component as the first argument to the ReactDOM.render() method.

The below syntax shows how to initialize a component to an element:

```
const elementName = <ComponentName />;
```

In the above syntax, the *ComponentName* is the name of the user-defined component.

Note: The name of a component should always start with a capital letter. This is done to differentiate a component tag from an **HTML tag**.

Example: This example renders a component named Welcome to the Screen.

javascript

```
// Filename - src/index.js:

import React from "react";
import ReactDOM from "react-dom";

// This is a functional component
const Welcome = () => {
    return <h1>Hello World!</h1>;
};

ReactDOM.render(
    <Welcome />,
    document.getElementById("root")
);
```

Output: This output will be visible on the **http://localhost:3000/** on the browser window.

Explanation:
Let us see step-wise what is happening in the above example:

- We call the ReactDOM.render() as the first parameter.
- React then calls the component Welcome, which returns <h1>Hello World!</h1>; as the result.
- Then the ReactDOM efficiently updates the DOM to match with the returned element and renders that element to the DOM element with id as “root”.

Components in Components

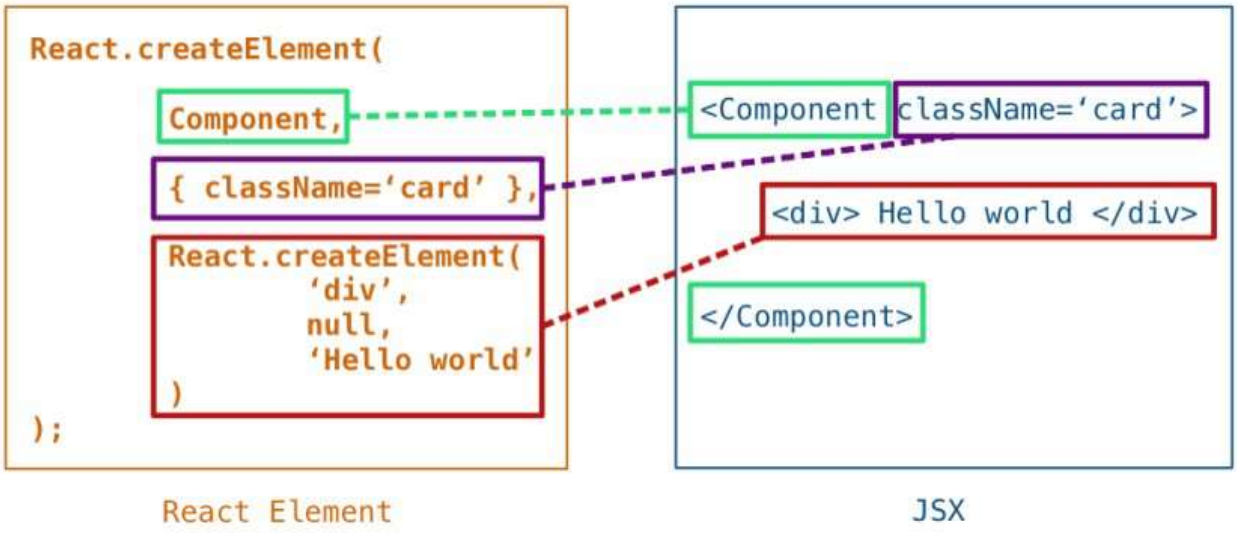
We can call components inside another component

```
// Filename - src/index.js:

import React from "react";
import ReactDOM from "react-dom";

const Greet = () => {
   return <h1>Hello Geek</h1>
}

// This is a functional component
const Welcome = () => {
    return <Greet />;
};

ReactDOM.render(
    <Welcome />,
    document.getElementById("root")
);
```

What is JSX ?

JSX stands for **JavaScript XML**. JSX is basically a syntax extension of JavaScript.

React JSX helps us to write HTML in JavaScript and forms the basis of React Development. Using JSX is not compulsory but it is highly recommended for programming in React as it makes the development process easier as the code becomes easy to write and read.
JSX creates an element in React that gets rendered in the UI. It is transformed into JavaScript functions by the compiler at runtime. Error handling and warnings become easier to handle when using JSX



React JSX sample code:

```
const ele = <h1>This is sample JSX</h1>;
```

This is called JSX (JavaScript XML), it somewhat looks like HTML and also uses a JavaScript-like variable but is neither HTML nor JavaScript. With the help of JSX, we have directly written the HTML syntax in JavaScript.

Expressions in JSX

In React we are allowed to use normal JavaScript expressions with JSX. To embed any JavaScript expression in a piece of code written in JSX we will have to wrap that expression in curly braces {}. The below example specifies a basic use of JavaScript Expression in React.

Syntax:

```
const example = "JSX"
const ele = <div>This component uses {example} </div>
```

```
// Filename - App.js

import React from "react";

const name = "Learner";

const element = (
    <h1>
        Hello,
        {name}.Welcome to GeeksforGeeks.
    </h1>
);

ReactDOM.render(element, document.getElementById("root"));
```

In the code provided, const element refers to a constant variable that holds a JSX expression. JSX (JavaScript XML) is a syntax extension used in React to describe the structure of the UI in a format that looks similar to HTML but is actually JavaScript under the hood.

Inside the JSX, {name} is JavaScript syntax, which allows you to embed a JavaScript expression (in this case, the value of name) into the JSX. The value of name is "Learner".

Attributes in JSX

JSX allows us to use attributes with the HTML elements just like we do with normal HTML. But instead of the normal naming convention of HTML, JSX uses the camelcase convention for attributes.

- **The change of class attribute to className:** The *class* in HTML becomes *className* in JSX. The main reason behind this is that some attribute names in HTML like '*class*' are reserved keywords in JavaScript. So, in order to avoid this problem, JSX uses the camel case naming convention for attributes.

- **Creation of custom attributes:**We can also use custom attributes in JSX. For custom attributes, the names of such attributes should be prefixed by **data-*** attribute.

Example: This example has a custom attribute with the <h2> tag and we are using className attribute instead of class.

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

const element = (
    <div>
        <h1 className="hello">Hello Geek</h1>
        <h2 data-sampleAttribute="sample">
            Custom attribute
        </h2>
    </div>
);

ReactDOM.render(element, document.getElementById("root"));
```

Specifying attribute values:

JSX allows us to specify attribute values in two ways:

- **As for string literals:** We can specify the values of attributes as hard-coded strings using quotes:

```
const ele = <h1 className = "firstAttribute">Hello!</h1>;
```

- **As expressions:** We can specify attributes as expressions using curly braces {}:

```
const ele = <h1 className = {varName}>Hello!</h1>;
```

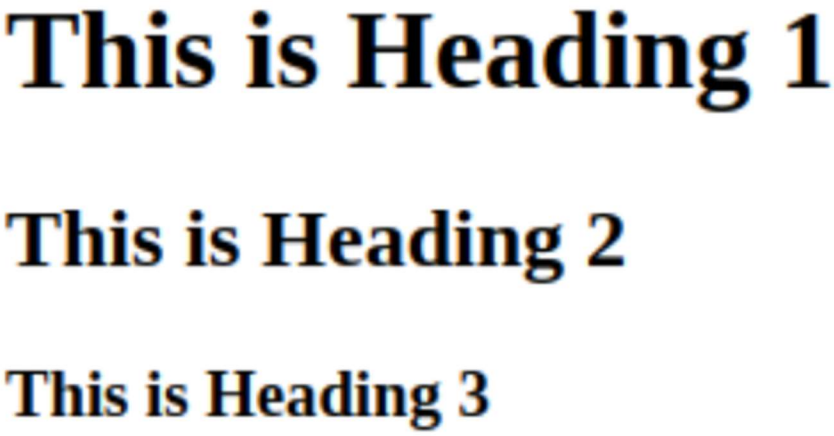
Wrapping elements or Children in JSX

Consider a situation where you want to render multiple tags at a time. To do this we need to wrap all of these tags under a parent tag and then render this parent element to the HTML. All the subtags are called child tags or children of this parent element.

Example: In this example we have wrapped h1, h2, and h3 tags under a single div element and rendered them to HTML:

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

const element = (
    <div>
        <h1>This is Heading 1 </h1>
        <h2>This is Heading 2</h2>
        <h3>This is Heading 3 </h3>
    </div>
);

ReactDOM.render(element, document.getElementById("root"));
```

Output:



Note: JSX will throw an error if the HTML is not correct or if there are multiple child elements without a parent element.

Comments in JSX:

JSX allows us to use comments as it allows us to use JavaScript expressions. Comments in JSX begin with */** and ends with ***/**. We can add comments in JSX by wrapping them in curly braces {} just like we did in the case of expressions. The below example shows how to add comments in JSX:

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

const element = (
    <div>
        <h1>Hello World !{/*This is a comment*/}</h1>
    </div>
);

ReactDOM.render(element, document.getElementById("root"));
```

Output:

Converting HTML to JSX

```
<!DOCTYPE html>
<html>

<head>
    <title>Basic Web Page</title>
</head>
<body>
    <h1>Welcome to GeeksforGeeks</h1>
    <p>A computer science portal for geeks</p>
</body>

</html>
```

The Converted JSX Code will look like:

```
<>
  <title>Basic Web Page</title>
  <h1>Welcome to GeeksforGeeks</h1>
  <p>A computer science portal for geeks</p>
</>
```

Rules to Write JSX

- **Always return a single Root Element:** When there are multiple elements in a component and you want to return all of them wrap them inside a single component
- **Close all the tags:** When we write tags in HTML some of them are self closing like the <img> tag but JSX requires us to close all of them so image tag will be represented as <img />
- **Use camelCase convention wherever possible:** When writing JSX if we want to give class to a tag we have to use the className attribute which follows camelCase convention.

"Till now we were working with components using static data only. Now, we will learn about how we can pass information to a Component."

Props and states

1. What are Props?

- Props (short for properties) are a fundamental concept in React.
- Think of them as attributes you pass to a component, similar to HTML attributes in tags.
- Props allow you to send data from a parent component to a child component.
- They are read-only and help make your components dynamic and reusable.

. Creating Props:

- To create props, you include attributes in your component like this:
- `<Contact name="Ali" />`

3. Accessing Props:

- In your child component, you access props using the `props` object:
- ```
  function Contact(props) {
    return (
      <h1>{props.name}</h1>
    );
  }
  ```

4. Destructuring Props:

- Destructuring allows you to extract props more concisely.
- Example:
- ```
  function Contact({ name }) {
    return (
      <h1>{name}</h1>
    );
  }
  ```
- This is especially useful when you have multiple props.

6. Example:

- Let's create a simple component that receives and displays user information using props and destructuring:

```
// ParentComponent.js
import React from 'react';
import UserProfile from './UserProfile';

function ParentComponent() {
  const user = {
    name: 'Ali',
    age: 30,
    city: 'New York',
  };

  return (
    <UserProfile user={user} />
  );
}

// UserProfile.js
import React from 'react';

function UserProfile({ user }) {
  return (
    <div>
      <h1>{user.name}</h1>
      <p>Age: {user.age}</p>
      <p>City: {user.city}</p>
    </div>
  );
}
```

State :

React separates passing data & manipulating data with props and state. I've explained how to pass data between components by using React **"props"**.

Props are only being used for passing data. They are **read-only** which means that components receiving data by props are not able to change it. However, in some cases, a component may need to manipulate data and that's not possible with props.

So React provides another feature for data manipulation which is known as **State**. Now you will learn what React's **"State"** is and how to use it.

What is State?

We can explain **state** under 5 pieces:

- State is a special object that holds dynamic data, which means that state can change over time and anytime based on user actions or certain events.
- State is private and belongs only to its component where defined, cannot be accessed from outside, but can be passed to child components via props.

- State is initialized inside its component's **constructor** method.
- When a change in the state is made, state shouldn't be modified directly. Instead, state updates should be made with a special method called **setState( )**.
- State should not be overly-used in order to prevent performance problems.

So these are the points basically explaining what State is, now let's see how to use it...

Using State in a Component

Creating The State

If you're familiar with object-oriented programming, you know that there is a structure called **class**.

A class has a special method called **constructor( )** and it is being called during object creation. We can also initialize our object properties or bind events inside the **constructor( )**.

The same rule applies to **state**. Since state is also an object, it should be initialized inside the **constructor** method:

```
constructor() {
  this.state = {
    id: 1,
    name: "test"
  };
}
```

and later we can render the properties of the state object with JavaScript's dot notation, inside the **render ( )** method:

```
class Test extends React.Component {
 constructor() {
    this.state = {
      id: 1,
      name: "test"
    };
  }  render() {
    return (
      <div>
        <p>{this.state.id}</p>
        <p>{this.state.name}</p>
      </div>
    );
  }
}
```

Now let's move a step forward and see how to update the state.

Updating The State

A Component's state can change under some circumstances like a server response or user interaction (clicking on a button, scrolling the page etc).

So when data changes, when a change in the state happens, React takes this information and updates the UI.

The important point here is that we should not modify the state directly.

Do Not Modify State Directly — React Official Docs

```
this.state.name = "testing state";  // wrong
```

So we shouldn't change a property of the state as we do it for other objects in JavaScript. Instead, React provides another way for handling state changes: the **setState( )** method.

Using setState( )

Below you can see the right way of state changes in React:

```
this.setState({
  name: "testing state"
});
```

The reason why we should use **setState( )** is that because it's the only way to notify React for data changes. Otherwise React won't be notified and won't be able to update the UI.

Example 1:

```
class Greeting extends Component {
  constructor(props) {
    super(props);
    this.state = {
```

```
    message: `Hello, ${props.name}!` // Embedding props.name into the string
    };
  }
  render() {
    return <h1>{this.state.message}</h1>;
  }
}
```

Example 2:

```
class Counter extends Component {
  // Initialize state inside the constructor
  constructor(props) {
    super(props);
    this.state = {
      count: 0 // Initial state value
    };
  }
  // Method to handle increment
  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };
  // Method to handle decrement
```

```
decrement = () => {
  this.setState({ count: this.state.count - 1 });
};
render() {
  return (
    <div>
      <h1>Count: {this.state.count}</h1>
      <button onClick={this.increment}>Increment</button>
      <button onClick={this.decrement}>Decrement</button>
    </div>
  ); } }
```

Key Differences Between Props and State:

Feature	Props	State
Definition	Props (short for "properties") are arguments passed into React components.	State is an internal data storage specific to the component, used to manage dynamic data.
Mutability	Immutable – cannot be modified by the component that receives them.	Mutable – can be updated using `setState` (in class components) or `useState` (in functional components).
Usage	Passed from a parent component to a child component.	Managed within a component to track changing data.
Updates	Cannot be changed by the receiving component but can be passed down again with new values from the parent.	Can be changed by the component itself using `setState()` or `useState()`.

Feature	Props	State
Responsibility	Managed by the parent component that passes them down.	Managed within the component itself.
Re-rendering	Changing props in the parent component will trigger a re-render of the child component.	Updating state triggers a re-render of the component to reflect the new state.
Access	Available to both functional and class components via `this.props` or directly in functional components via `props`.	Available via `this.state` in class components or via `useState` in functional components.
Example Use Case	Use props to pass data or functions from parent to child components.	Use state to manage data that changes over time (e.g., user input, form fields, toggles).

Stateless and Stateful Components:

1. Stateless Components:

- **Definition**: A **stateless** component (also known as a **presentational** or **functional** component) is one that does not manage or hold any internal state. It purely relies on **props** to render data and doesn't handle state updates.
- **Key Characteristics**:
  - They receive data via **props**.
  - They don't have their own internal state.
  - Typically simpler, easier to understand, and easier to test.

Example of Stateless (Functional) Component:

```
function Greeting (props) {

 return <h1>Hello, {props.name}!</h1>; }

// Usage:

<Greeting name="John" />
```

2. Stateful Components:

- **Definition**: A **stateful** component (also known as a **container** component) is one that maintains and manages its own **state**. These components can modify their own state using `setState` (in class components) or the `useState` hook (in functional components), and they typically manage more complex logic.
- **Key Characteristics**:
  - They hold their own **internal state**.
  - State changes trigger re-rendering of the component.
  - Often more complex than stateless components as they handle logic and state management.

Example of Stateful (Class) Component:

```
class Counter extends Component {
 constructor(props) {
  super(props);
  this.state = { count: 0 };  // Initialize state
 }
 // Method to update the state
 increment = () => {
  this.setState({ count: this.state.count + 1 });
 };
 render() {
  return (
   <div>
    <h1>Count: {this.state.count}</h1>
    <button onClick={this.increment}>Increment</button>
   </div> ); } }
```

The `Counter` component has its own `state` (`count`) and can update it using the `setState` method, making it stateful.

Stateless vs. Stateful:

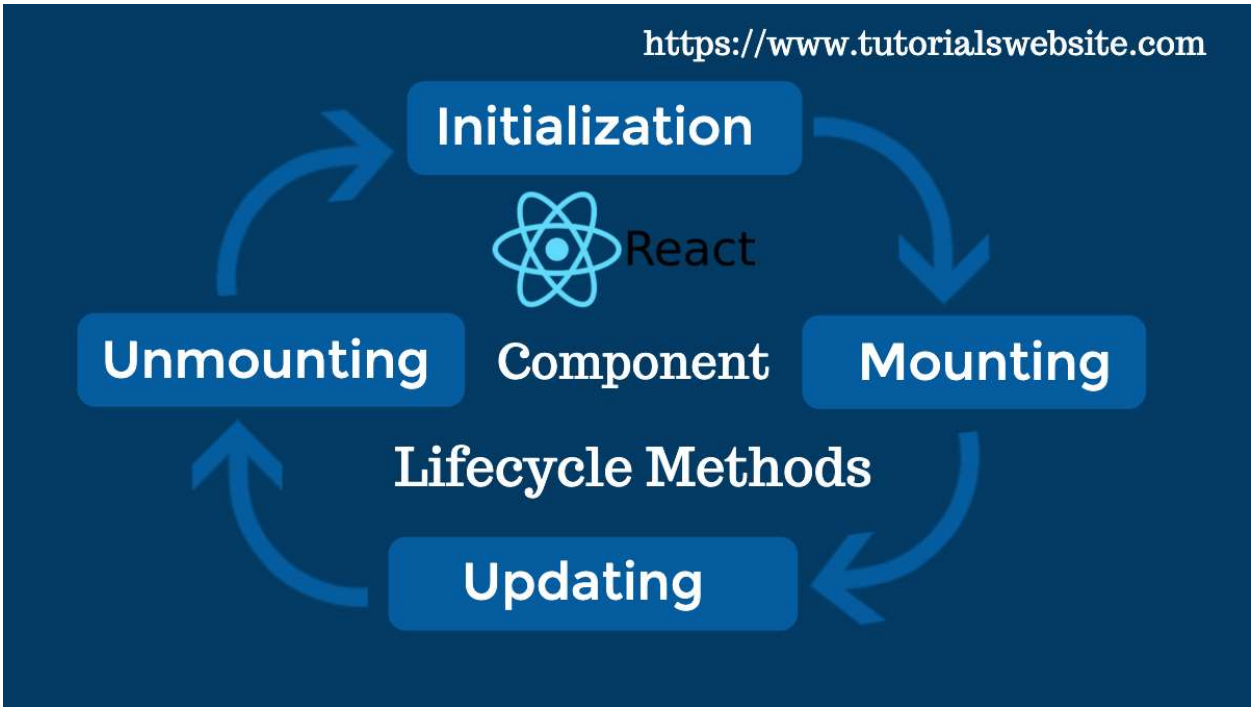
Feature	Stateless (Functional)	Stateful (Class/Functional with Hooks)
State Management	No internal state, only uses props	Manages its own state
Complexity	Simple and easy to test	More complex, handles logic and state
Re-rendering	Re-renders only when props change	Re-renders when state or props change
Performance	Generally faster, less overhead	Slightly more overhead due to state
Example	Displaying static data or UI	Managing forms, user input, dynamic data

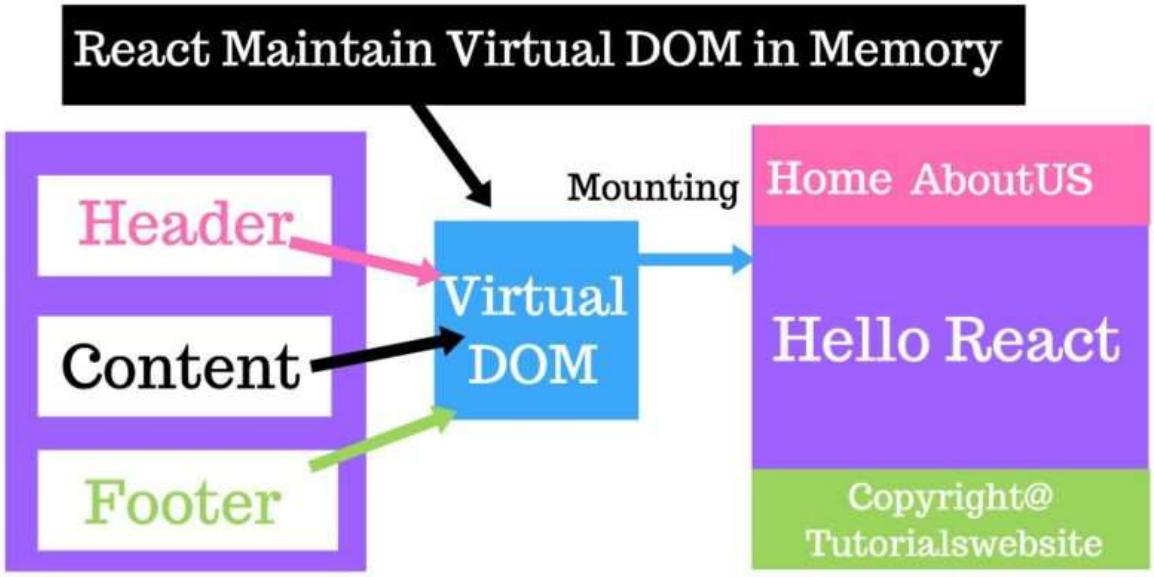
Component life cycle:

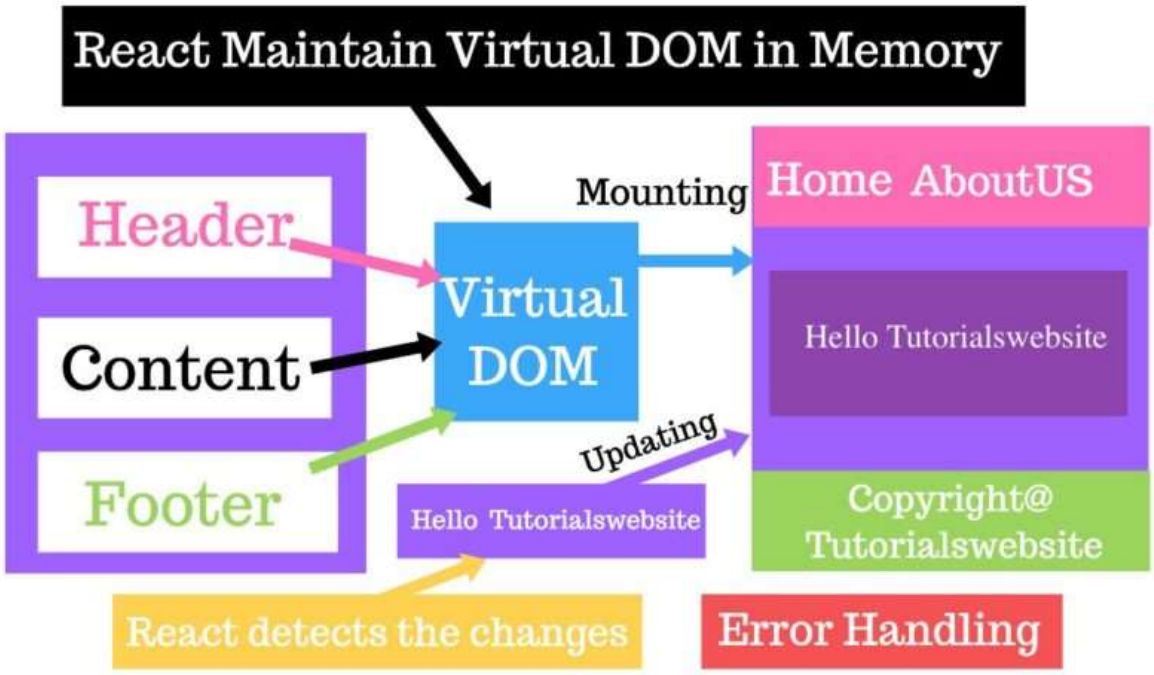
In React, **Component Lifecycle** refers to the different stages that a component goes through from the moment it's created (mounted) to when it's updated or removed from the DOM (unmounted). React provides specific lifecycle methods that allow you to control what happens at each stage.

React Class Component Lifecycle has three main phases:

1. **Mounting**: When the component is being created and inserted into the DOM.
2. **Updating**: When the component's state or props change and it needs to re-render.
3. **Unmounting**: When the component is removed from the DOM.


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


React Maintain Virtual DOM in Memory
Header
Content
Footer
Mounting
Virtual DOM
Home AboutUS
Hello React
Copyright@ Tutorialswebsite



1. Mounting Phase (Creating the component)

When the component is first created and added to the DOM, the following methods are called in order:

- `constructor()`: Initializes state and props.
- `render()`: Renders the JSX to the DOM.
- `componentDidMount()`: Called after the component has been rendered to the DOM. It is commonly used to fetch data or run side effects.

2. Updating Phase (Re-rendering when state or props change)

When the component's props or state change, it goes through the updating phase:

- `render()`: React re-renders the component.
- `componentDidUpdate()`: Called after the component has updated, useful for side effects after an update.

3. Unmounting Phase (Removing the component)

When the component is removed from the DOM:

- `componentWillUnmount()`: Called right before the component is removed, often used for cleanup (e.g., clearing timers or event listeners).

Example:

```
class LifecycleDemo extends Component {
  // Mounting Phase:
  constructor(props) {
    super(props);
    this.state = { count: 0 };
    console.log("Component is being created");
  }
  // Mounting Phase: Called after the component is rendered
  componentDidMount() {
    console.log("ComponentDidMount: Component has been rendered");
  }
  // Updating Phase: Called after the component has updated
  componentDidUpdate(prevProps, prevState) {
    console.log("ComponentDidUpdate: Component updated");
  }
```

```
  // Unmounting Phase: Called right before the component is removed
  componentWillUnmount() {
    console.log("ComponentWillUnmount: Component is being removed");
  }
  // Method to update state
  increment = () => {
    this.setState({ count: this.state.count + 1 });
  };
  // Rendering phase: Renders the JSX
  render() {
    console.log("Render: Rendering the component");
    return (
      <div>
        <h1>Count: {this.state.count}</h1>
        <button onClick={this.increment}>Increment</button>
      </div>
    ); } }
```

Summary of Lifecycle Methods:

- **Mounting**: `constructor()`, `render()`, `componentDidMount()`
- **Updating**: `render()`, `componentDidUpdate()`
- **Unmounting**: `componentWillUnmount()`

These methods help manage how your component behaves during its lifetime, like fetching data when it mounts, updating the DOM when the state changes, and cleaning up when it's removed.

Additional information:

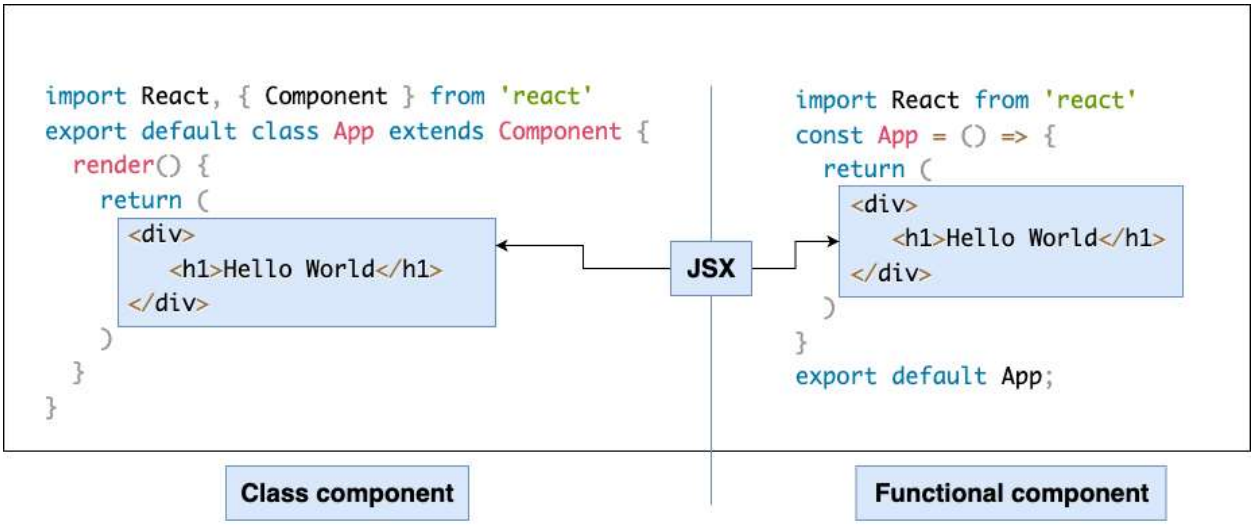
When Would You Use `componentWillUnmount` or Cleanup Functions?

- **Cancelling API requests** when the user navigates away.
- **Removing event listeners** attached to DOM elements or external systems.
- **Clearing intervals or timeouts** that were set up in `componentDidMount`.
- **Cleaning up WebSocket connections** or closing network connections

The **unmounting phase** is crucial for cleaning up any ongoing processes or side effects to ensure your app is efficient, prevents memory leaks, and frees up resources when a component is no longer in use.

Hooks

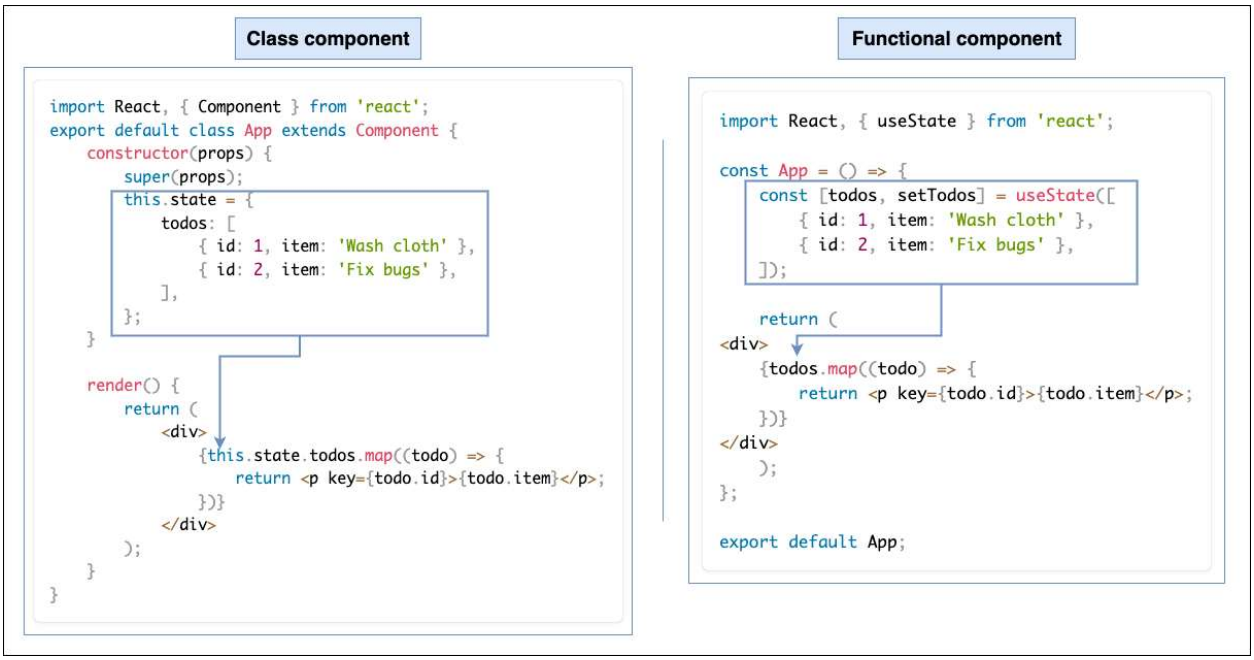
Hooks in React are like **special tools** that allow you to do more powerful things with **functional components**.



Class component had always been the superior method of creating a component when you need to work with state (data management) and handle lifecycle methods, like component mounts, renders, updates, and unmounts.

React Hooks help us Hook into React features in functional components. These features include state and side effects (similar to lifecycle methods).

The two most common React Hooks are useState() and useEffect(). The useState() Hook acts as a store that allows it to use state variables and update them. You can even destructure the useState() Hook to hold the state variable and update function separately. It also allows you to pass an initial value directly into the Hook.

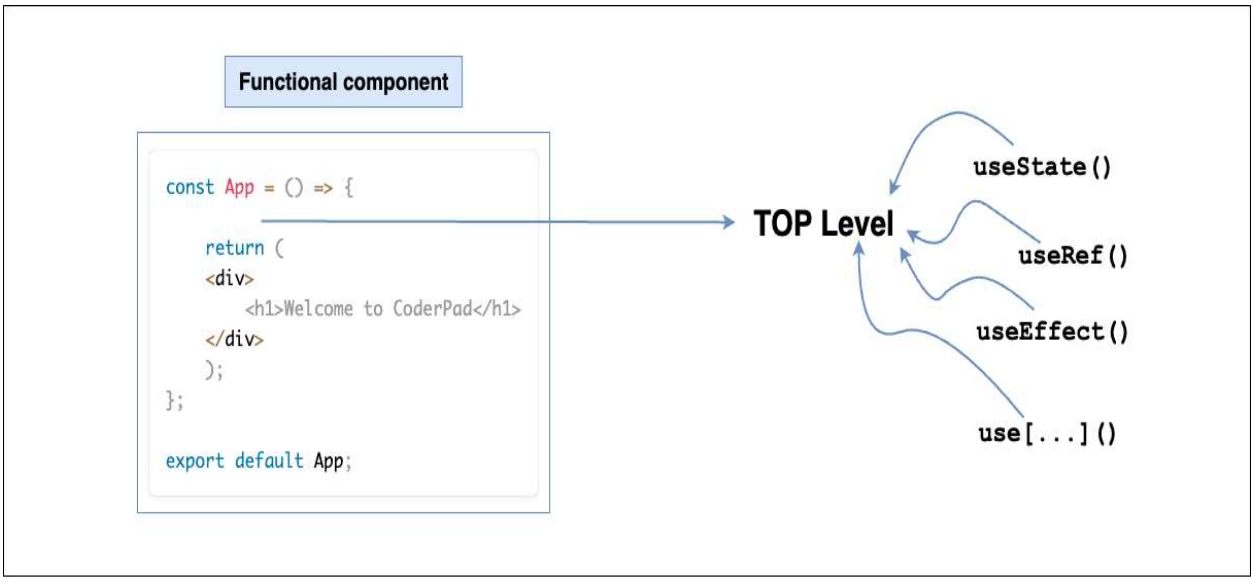


The two rules of React Hooks:

Two significant rules need to be followed when working with Hooks. These rules are essential to maintain order, avoid unnecessary bugs, and help us write clean code. These rules are:

1. Only call Hooks at the top level
2. Only call Hooks from React functional components or other Hooks

1. **The top level of a functional component** is the base of your function body before you return your JSX elements. This is where you can call all your Hooks, so React can preserve and take note of the order in which these Hooks are called.



Calling your Hooks at the top of the function means you **must not be calling them within loops, conditions, and nested functions**.

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

const App = () => {
    // ✅ - correct
    const [todos, setTodos] = useState([]);

    // ❌ - Breaks the call order
    if (todos) {
        const [count, setSetcount] = useState(todos.length);
    }

    // ❌ - Breaks the call order
    todos.forEach(todo => {
        useEffect(() => {
            console.log(todos);
        });
    });
```

2. Only call Hooks from React functional components or other Hooks

It is not a functional component

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

function getName() {
    const [name, setName] = useState("John Doe");
    return name;
}
document.getElementById("user-name").innerHTML = getName();
```

If you want to use then you can use

```
export default function useMyName(name) {
    const [state, setState] = useState(value);

    useEffect(() => {
    // ...
    });

    return anyThing;
}
```

Example 1: useState

import React, { useState } from 'react';

function SimpleCounter() {

  const [count, setCount] = useState(0); // Initialize count to 0

```
  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
export default SimpleCounter;
```

Example 2: UseEffect

```
import React, { useState, useEffect } from 'react';
function Timer() {
  const [count, setCount] = useState(0);
  // Use useEffect to run code when the component mounts
  useEffect(() => {
    // Set up a timer that increments the count every second
    const timer = setInterval(() => {
      setCount(prevCount => prevCount + 1);
    }, 1000);

    // Cleanup function: Stop the timer when the component unmounts
```

```
    return () => clearInterval(timer);
  }, []); // Empty array means it runs only once (on mount)
  return (
    <div>
      <h1>Timer: {count}</h1>
    </div>
  );
}
export default Timer;
```

Common Hooks used in react:

1. **`useState`**: Lets you add state to functional components.
2. **`useEffect`**: Lets you perform side effects (like data fetching, DOM manipulation, or setting up timers).
3. **`useContext`**: Used to access values from a React context.
4. **`useReducer`**: An alternative to `useState` for managing more complex state logic.
5. **`useRef`**: Allows you to create a reference to a DOM element or keep a persistent value across renders without re-rendering the component.

React Router vs. React Router DOM:

Routing is an essential technique for navigation among pages on a website based on user requests and actions.

A separate library named **React Router** enables routing in React applications and allows defining multiple routes in an application. But whether to install the **react-router** or **react-router-dom** package can be confusing.

Why Is React Router Needed?

React is a famous JavaScript framework ideal for building single-page applications. Although it is one of the best solutions for building websites, React does not include many advanced features or routing by default. Therefore, React Router is an excellent choice of navigation for these single-page applications to render multiple views.

What Is React Router?

React Router is a popular standard library for routing among various view components in React applications. It helps keep the user interface in sync with the URL. In addition, React Router allows defining which view to display for a specified URL.

The three main packages related to React Router are:

- react-router: Contains the core functionality of React Router, including route-matching algorithms and hooks.
- react-router-dom: Includes everything in **react-router** and adds a few DOM-specific APIs.

- react-router-native: Includes everything in **react-router** and adds a few React Native-specific APIs.

What Is React Router DOM?

The primary functionality of react-router-dom is implementing dynamic routing in web applications. Based on the platform and the requirements of the application, **react-router-dom** supports component-based routing, which is the ideal solution for routing if the React application is running on the browser.

How to Use React Router DOM

Step 1: Install the package.

```
npm install react-router-dom
```

Step 2: Import **<BrowserRouter>.**

```
import { BrowserRouter, Route } from 'react-router-dom';

function App() {
 return (
  <BrowserRouter>

  </BrowserRouter>
 );
}

export default App;
```

Step 3: Import and use the child component, **<Route>**.

```
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import About from './components/about';
```

```
import Home from './components/home';

function App() {
 return (
  <BrowserRouter>
   <div><Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
    </Routes>
   </div>
  </BrowserRouter>
 );
}
```

As the URL changes to the specified paths, the user interface also changes to display the specific component. Following are several components that you can use in your application.

```
function Navbar() {
   return (
     <nav>
           <Link to="/">Home</Link>
           <Link to="/profile">Profile</Link>
     </nav>
   )
}
```