REACT :

Open source library for building user interfaces.

Not a framework

Focus on UI

Rich ecosystem

Created and maintained by Facebook

More than 100k starts on github.

Huge community.

In-demand skillset.

Component based architecture :

Reusable code:

React is declarative :

React will handle efficiently updating and rendering of the components.

DOM updates are handles gracefully in React.

Seamlessly integrate react into any of your applications.

Portion of your page or a complete page or even an entire application itself.

React native for mobile application

NPX :

npx create-react-app <project_name>

npm package runner

npm :

npm install create-react-app -g

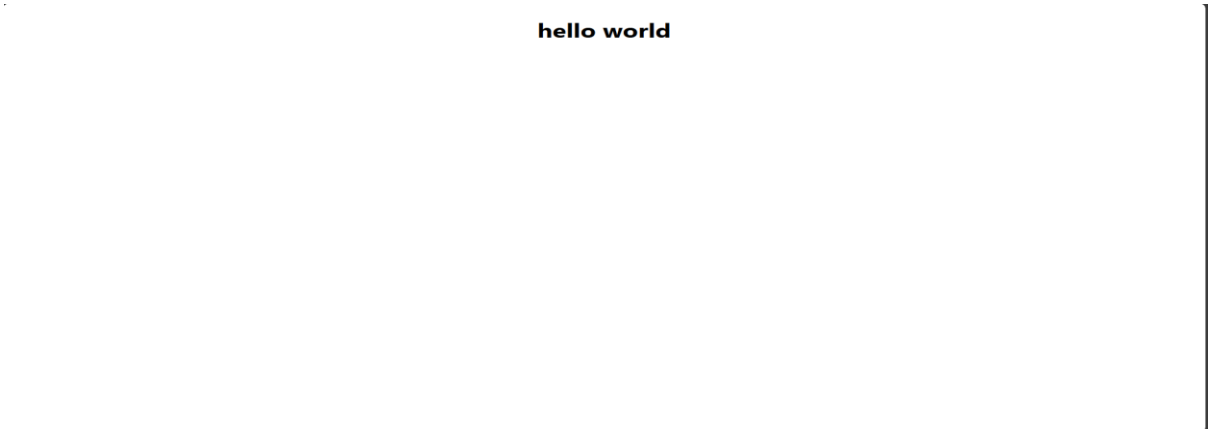
create-react-app<project_name>

Component :

Two types

Stateless functional component:

```
1  import React from 'react';
2
3  function Greet() {
4    return <h1>Hello World</h1>;
5  }
6
7  export default Greet;
8
```

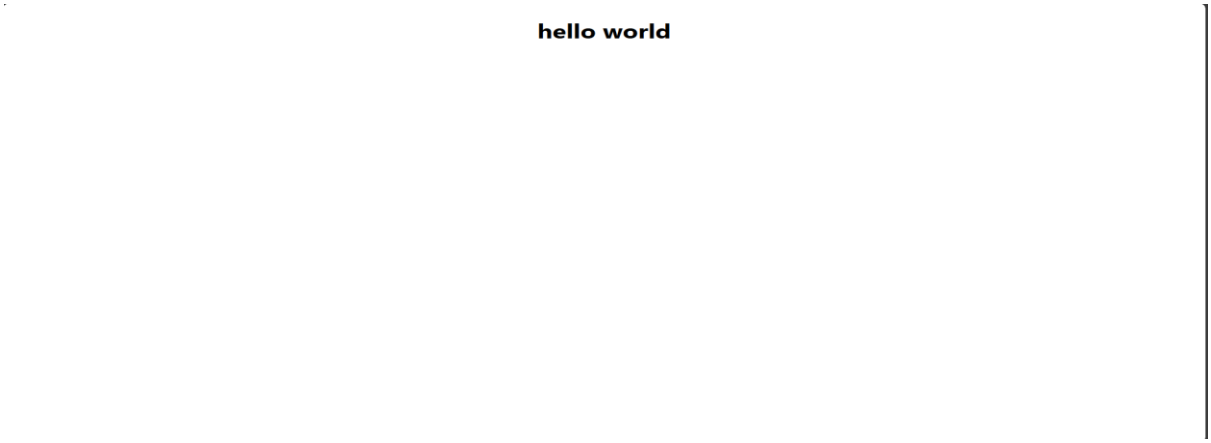


Javascript functions

```
1  import React, { Component } from 'react';
2  import logo from './logo.svg';
3  import './App.css';
4  import Greet from './components/greet';
5
6  class App extends Component {
7    render() {
8      return (
9        <div className="App">
10         <Greet />
11       </div>
12     );
13   }
14 }
15
16 export default App;
17
```

Stateful class component :

```
1   import React,{Component} from 'react';
2
3   class Greet extends Component{
4         constructor(props) {
5       super(props);
6     }
7       render(){
8             return(
9                   <h1>hello world</h1>
10                  );
11      }
12  }
13
14  export default Greet;
15
```



Class extending component class

Render method returning HTML.

Components describe a part of user interface.

They are re-usable and can be nested inside other components.

Functional	Class
Simple functions	More feature rich
Use func components as much as possible	Maintain their own private data -state
Absence of this key word	Complete UI logic
Solution without using state	Provide lifecycle hooks
Mainly responsible for UI	Stateful/smart/container
Stateless/dumbs/presentational	

HOOKS:

No breaking changes

Completely opt-in & 100% backwards-compatible

JSX :

Javascript XML → extension to the javascript language syntax

Write XML-like code for elements and components

JSX tags have a tag name, attributes and children.

JSX is not necessity to write react-applications.

Jsx makes your react code simpler and elegant

Jsx ultimately transpiles to pure javascript which is understood by the browsers

REACT 16.7.0 – alpha :

With jsx :

```
1  import React from 'react';
2
3  const Hello=()=>{
4      /*return(
5          <h1>hello tnr</h1>
6          )*/
7      return React.createElement('div',null,<h1>heloo tnr</h1>)
8  }
9  export default Hello
```

PROPS :

Props (short for "properties") are a way to pass **data from a parent component to a child component**. Props allow components to be **reusable** and **dynamic**. They are like function arguments but for React components.

- **Immutable**: Props are **read-only**. The child component cannot modify props directly.
- **Passed from Parent to Child**: Props are passed as attributes in JSX tags.
- **Accessed via this.props in Class Components** or **as arguments in Functional Components**.
- **Used for Customization**: Props allow you to make components dynamic.
- **Similar to Function Parameters**: Just like you pass arguments to a function, you pass props to components.

Hello.js(child)

```
1  import React from 'react';
2
3  const Hello=(props)=>{
4      return(
5          <h1>hello {props.name}</h1>
6          )
7  }
8  export default Hello
```

App.js(parent)

```
7   class App extends Component {
8     render() {
9       return (
10        <div className="App">
11          {/*<Greet />*/}
12          <Hello name="tnr"/>
13          <Hello name="vijay"/>
14        </div>
15      );
16    }
17  }
```

hello tnr

hello vijay

Class component:

We use this

```
class Greet extends Component{
    render(){
        return(
            <h1>hello {this.props.name}</h1>
            );
    }
}
```

Multiple props :

```
const Hello=(props)=>{
    return(
        <h1>hello {props.name} from {props.collage}</h1>
        )
}
```

```
class App extends Component {
  render() {
    return (
      <div className="App">
        {/*<Greet name="tnr"/>
        <Greet name="vijay"/>*/}
        <Hello name="tnr" collage="srm"/>
        <Hello name="vijay" collage="srm"/>
      </div>
    );
  }
}
```

hello tnr from srm

hello vijay from srm

Default Props :

```
1   import React from 'react';
2
3   const Hello = (props) => {
4     return (
5       <h1>Hello, {props.name}</h1>
6     );
7   }
8
9   Hello.defaultProps = {
10    name: 'TNR'
11  };
12
13  export default Hello;
14
```

Type checking :

```
import PropTypes from 'prop-types';

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

Greet.propTypes = {
  name: PropTypes.string.isRequired
};
```

Child:

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

class Greet extends Component{
    render(){
        return(
            <h1>hello {this.props.name}</h1>
            );
    }
}

export default Greet;
```

```
class App extends Component {
  render() {
    return (
      <div className="App">
        {/*<Greet name="tnr"/>
        <Greet name="vijay"/>*/}
        <Hello name="tnr" >
        <p>srm</p>
        </Hello>
        <Hello name="vijay">
        <button>action</button></Hello>
      </div>
    );
  }
}
```

Destructuring :

 Reusability: When you want a component to be reused with different data (like user names, products, etc.).

 Customization: To customize the appearance or content of a component.

 Data Flow: For parent-to-child data flow.

State:

In React, state refers to an object that holds data or information about the component. It determines the component's behavior and how it renders. When the state changes, the component re-renders to display the updated state.

- **State is Mutable**: Unlike props, state can be changed inside the component.
- **State Triggers Re-Render**: Any change in the state will trigger React to re-render the component.
- **State is Local**: Each component has its own local state.

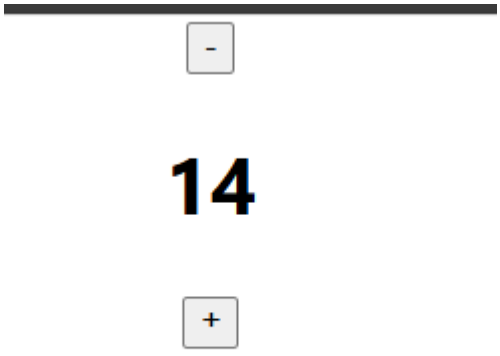
props	State
Props get passed to the component	State is managed with in component
Function parameters	Variables declared in the function body
Props are immutable	State can be changed
Props – functional components This.props→class components	useState Hook →functional component this.state→class component

Message.js:

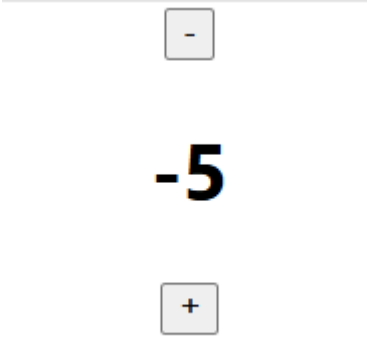
```
import React, { Component } from 'react';

class Message extends Component {
  constructor() {
    super();
    this.state = {
      message: 'welcome'
    };
  }

  changeMessage() {
    this.setState({
      message: 'thank you'
    });
  }

  render() {
    return (
      <div>
        <h1>{this.state.message}</h1>
        <button onClick={() => this.changeMessage()}>subscribe</button>
      </div>
    );
  }
}

export default Message;
```

Count operation using js:

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

class Count extends Component{
    constructor(){
        super()
        this.state={
            count:0
        }
    }
    increment(){
        this.setState({
            count:this.state.count+1
        })
    }
    decrement(){
        this.setState({
            count:this.state.count-1
        })
    }
    render(){
    return (
        <div>
        <button onClick={()=>this.decrement()}>-</button>
        <h1>{this.state.count}</h1>
        <button onClick={()=>this.increment()}>+</button>
        </div>
        )
}
```



```jsx
import React,{useState} from 'react';
const Counterx=()=>{
    const[count,setCount]=useState(0);
    const increment=()=>{
        setCount(count+1)
    }
    const decrement=()=>{
        setCount(count-1)
    }
    return(
        <div>
        <button onClick={decrement}>-</button>
        <h1>{count}</h1>
        <button onClick={increment}>+</button>
        </div>
        )
}

export default Counterx;
```



What is the difference between state and props?

- **Props** are read-only and passed from parent to child.
- **State** is local to the component and can be changed using setState or useState.

How does state affect re-rendering?

- When state changes, React triggers a re-render of the component.

❓ Can you modify state directly?

- No, you must use this.setState() or setCount() to modify state.

❓ Can functional components have state?

- Yes, with the use of the **useState hook**.

Feature	Class Component	Functional Component
Initialization	`this.state = {}`	`const [state, setState] = useState(initialValue)`
Access State	`this.state.count`	`count`
Update State	`this.setState({})`	`setCount(newValue)`
Syntax	More verbose (constructor)	Cleaner and more concise
Hooks	Not needed	Required (`useState`)

Destructuring :

Unpack the values

```jsx
import React from 'react';

const Hello = ({name,collage}) => {
   return (
      <div>
      <h1>Hello {name} {collage}</h1>
      </div>
   )
}


export default Hello;
```

```jsx
import React from 'react';

const Hello = (props) => {
      const {name,collage}=props
   return (
      <div>
      <h1>Hello {name} {collage}</h1>
      </div>
   )
}


export default Hello;
```

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

class Greet extends Component{
    render(){
        const {name}=this.props
        return(
            <h1>hello {name}</h1>
            );
    }
}

export default Greet;
```

Event bind:

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

class EventBind extends Component{
    constructor(props){
        super(props)
        this.state={
            message:"hello"
        }
    }
    click(){
        this.setState({
            message:"goodbye"
        })
    }
    render(){
        return(
            <div>
            <h1>{this.state.message}</h1>
            <button onClick={this.click.bind(this)}>click</button>
            </div>
            )
    }
}
export default EventBind;
```

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

class EventBind extends Component{
    constructor(props){
        super(props)
        this.state={
            message:"hello"
        }
    }
    click=()=>{
        this.setState({
            message:"goodbye"
        })
    }
    render(){
        return(
            <div>
            <h1>{this.state.message}</h1>
            <button onClick={this.click}>click</button>
            </div>
            )
    }
}
export default EventBind;
```

Method as props:

Parent-Child Communication:

- The parent can pass a method to the child as a prop.
- The child can call this method to update the parent's state or trigger actions in the parent.

Reusability and Modularity:

- You can reuse child components and control their behavior from the parent.

Event Handling:

- Child components can handle events (like button clicks) and pass the event data to the parent.

Use Case	Example	Description
Parent controls Child	Call Parent method from Child	Pass method from Parent to Child
Child updates Parent	Send data from Child to Parent	Child calls the parent's method and sends data as an argument
Child triggers Parent's State	Child clicks a button to update Parent state	Pass Parent's method to Child and call it from Child's onClick

Child component:

```
1  import React,{ Component } from 'react'
2  function ChildComponent(props){
3      return(
4          <div>
5          <button onClick={()=>props.greet}>greet parent</button>
6          </div>
7          )
8  }
9  export default ChildComponent;
```

Parent component :

```jsx
import React,{Component} from 'react';
import ChildComponent from './childcomponent'
class ParentComponet extends Component{
    constructor(props){
        super(props)
        this.state={
            parentname:'parent'
        }
        this.greetparent=this.greetparent.bind(this)
    }
    greetparent(){
        alert(`welcom ${this.state.parentname}`)
    }
    render(){
        return(
            <div>
            <ChildComponent greet={this.greetparent}/>
            </div>
            )
    }

}
export default ParentComponet;
```

Conditional rendering:

Conditional Rendering in React means showing or hiding elements based on a condition. Instead of rendering everything at once, React allows you to render specific components or elements based on the state or props.

- **Display Components Conditionally** (e.g., Login/Logout buttons).
- **Show/Hide UI Elements** (e.g., loader, modals, or alerts).
- **Dynamic Component Rendering** (e.g., light/dark mode toggle).

Method	Example	When to Use?
if-else	`if (condition) { return <A /> } else { return <B /> }`	Large conditional logic
Ternary	`{condition ? <A /> : <B />}`	Simple true/false conditions
Logical &&	`{condition && <A />}`	Render only if true
Switch Case	`switch(condition) { case 1: return <A /> }`	Multiple conditions
Short-Circuit	`{condition && <A />}`	Shortest way for true-only cases

If else :

```
1  import React, { Component} from 'react';
2  class UserGreet extends Component{
3      constructor(props){
4          super(props)
5          this.state={
6              isloggedin:false
7          }
8      }
9      render(){
10         if(this.state.isloggedin){
11             return(
12                 <div>welcome tnr</div>
13                 )
14         }
15         else{
16             return(
17             <div>welcome guest</div>
18             )
19         }
20     }
21 }
22 export default UserGreet;
```

Element method :

```
import React, { Component} from 'react';
class UserGreet extends Component{
    constructor(props){
        super(props)
        this.state={
            isloggedin:false
        }
    }
    render(){
        let message
        if(this.state.isloggedin){
            message=<div>welcome tnr</div>
        }
        else{
            message=<div>welcome guest</div>
        }
        return(
            message
            )
    }
}
export default UserGreet;
```

Ternary condition:

```
import React, { Component} from 'react';
class UserGreet extends Component{
    constructor(props){
        super(props)
        this.state={
            isloggedin:false
        }
    }
    render(){
        return (this.state.isloggedin ? <div>welcome tnr</div> :
            <div>welcome guest</div>)
    }
}
export default UserGreet;
```

Short circuit method:

```
import React, { Component} from 'react';
class UserGreet extends Component{
    constructor(props){
        super(props)
        this.state={
            isloggedin:true
        }
    }
    render(){
        return (this.state.isloggedin && <div>welcome tnr</div>)
    }
}
export default UserGreet;
```

Lists :

In React, lists are used to display multiple elements dynamically. They are often created using arrays and the map() method to generate components. Lists are crucial when rendering multiple components like user profiles, products, or notifications.

- **Store data in an array**.
- **Use map()** to transform the array into a list of components.
- **Render the list** inside a component.
- **Add a unique key** to each item to help React efficiently update the DOM.

```
import React,{ Component } from 'react'
 function NameList(){
    const names=['tnr','vijay','bhargav','ahalya']
    return(
        <div>
        {
            names.map((x)=>
                <h2>{x}</h2>
            )
        }
        </div>
        )
 }
 export default NameList;
```

User key:

```
import React from 'react';

function List() {
  const users = [
    { id: 1, name: 'TNR', age: 22 },
    { id: 2, name: 'Vijay', age: 21 },
    { id: 3, name: 'Ravi', age: 23 }
  ];

  return (
    <div>
      <h1>User List</h1>
      <ul>
        {users.map(user => (
          <li key={user.id}>
            Name: {user.name}, Age: {user.age}
          </li>
        ))}
      </ul>
    </div>
  );
}

export default List;
```

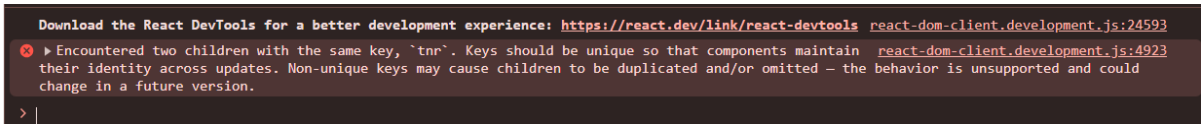
A key is a special string attribute you need to include when creating list of elements.

Keys give the elements a stable identity.

Keys help react identity which items have changed, are added, or removed.

Help in efficient update of the user interface.

```
import React,{ Component } from 'react'
 function NameList(){
    const names=['tnr','vijay','bhargav','ahalya','tnr']
    return(
        <div>
        {
            names.map((x)=>
                <h2 key={x}>{x}</h2>
            )
        }
        </div>
        )
 }
 export default NameList;
```




```
import React,{ Component } from 'react'
 function NameList(){
    const names=['tnr','vijay','bhargav','ahalya','tnr']
    return(
        <div>
        {
            names.map((x,index)=>
                <h2 key={index}>{x}</h2>
            )
        }
        </div>
        )
 }
 export default NameList;
```

To do list:

https://codepen.io/gopinav/pen/gQpepq?editors=0010

styling :

Controlled components:

A controlled component in React is a form element (like <input>, <textarea>, <select>) that is controlled by React state. Instead of relying on the browser's default behavior, the value of the form element is bound to the state of the component.

- **Single Source of Truth**: The value of the input field is stored in the component's state.
- **Real-Time Updates**: Changes to the input reflect immediately in the state, and vice versa.
- **Event Handling**: Uses onChange to capture changes and update the state.

Single element change:

```
class SingleElement extends Component{
    constructor(props){
        super(props)
        this.state={
            username:''
        }
    }
    change=(event)=>{
        //console.log(event.target.value)
        this.setState({
            username:event.target.value
        })
    }
    render(){
        return(
            <form>
            <div>
            <label>username</label>
            <input type='text'
            value={this.state.username}
            onChange={this.change}/>
            </div>
            <div>{this.state.username}</div>
            </form>
            )
    }
}
```

 State: The input's value is stored in this.state.name.

 Controlled Input: The value of the input is tied to this.state.name.

 onChange: Updates the state when the user types.

Multi element:

```
class MultiElement extends Component {
  constructor() {
    super();
    this.state = {
      username: '',
      email: ''
    };
  }

  change = (event) => {
    const { name, value } = event.target;
    this.setState({ [name]: value });
  };

  render() {
    return (
      <form>
        <div>
          <h2>Multi inputs</h2>
          <div>
            <input
              type="text"
              name="username" // Add the name attribute
              value={this.state.username}
              onChange={this.change}
              placeholder="Username"
            />
          </div>
```

```
          <div>
            <input
              type="email"
              name="email" // Add the name attribute
              value={this.state.email}
              onChange={this.change}
              placeholder="Email"
            />
          </div>
          <div>
            <h2>
              Username: {this.state.username}, Email: {this.state.email}
            </h2>
          </div>
        </div>
      </form>
    );
```

Text area:

```
import React,{ Component } from 'react';
class Textarea extends Component{
    constructor(){
        super()
        this.state={
            message:''
        }
    }
    change=(event)=>{
        this.setState({
            message:event.target.value
        })
    }
    render(){
        return(
            <form>
            <div>
            <h1>text area</h1>
            <textarea type="text"
            value={this.state.message}
            onChange={this.change}/>
            </div>
            <div>{this.state.message}</div>
            </form>
            )
    }
}
export default Textarea;
```

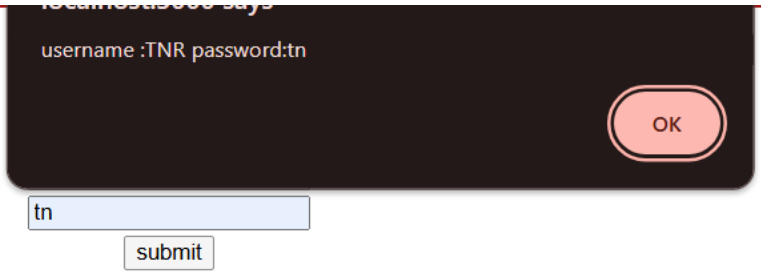
Select :

```
class Select extends Component{
    constructor(){
        super()
        this.state={
            selected:'React'
        }
    }
    change=(event)=>{
        this.setState({
            selected:event.target.value
        })
    }
    render(){
        return(
            <div>
            <select
            value={this.state.selected}
            onChange={this.change}
            >
            <option>React</option>
            <option>Vue</option>
            <option>java</option>
            </select>
            <p>selected:{this.state.selected}</p>
            </div>
            )
    }
}
```

FORM submission :

```
import React, { Component } from 'react';
class Form extends Component{
    constructor(){
        super()
        this.state={
            username:'',
            password:''
        }
    }
    change=(event)=>{
        const {name,value}=event.target
        this.setState({ [name] : value})
    }
    submit=(event)=>{
        event.preventDefault()
        alert(`username :${this.state.username} password:${this.state.password}`)
    }
    render(){
        return(
            <form onSubmit={this.submit}>
            <div>
            <h1>login form</h1>
            <input type="text" placeholder="username"
            name="username"
            value={this.state.username}
            onChange={this.change}/><br />
            <input type="text" placeholder="password"
            name="password"
```

```
            name="password"
            value={this.state.password}
            onChange={this.change}/>
            </div>
            <button type="submit">submit</button>
            </form>
            )
        }
    }
```



Advantages of Controlled Components

1. **Predictable Behavior**: All data is stored in the component state, making it easier to debug and manage.
2. **Validation**: You can validate user input in real-time as the user types.
3. **Custom Logic**: Implement custom logic, like trimming whitespace, or converting to uppercase, before updating the state.

Life cycle methods :

In React, Lifecycle Methods are special methods that allow you to hook into different phases of a component's life. These methods are available in class components and are divided into three main phases:

Mounting : when an instance of a component is being created and inserted into the DOM

constructor():

- Used to initialize state and bind methods.
- Called before the component is mounted.

static getDerivedStateFromProps(props, state):

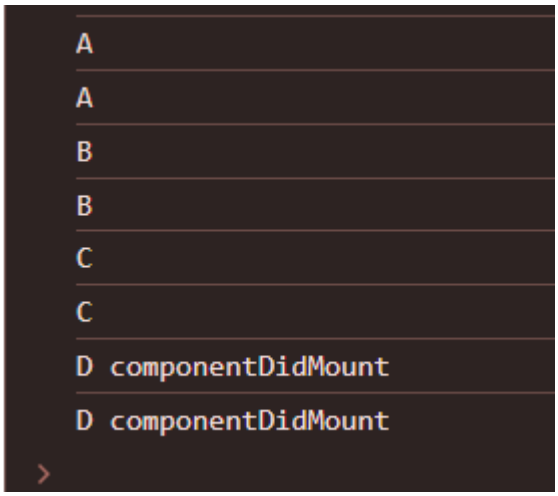
- Syncs state with props if needed.
- Rarely used and must return an object or null.

render():

- The only **required** method.
- Returns JSX to be rendered in the DOM.

componentDidMount():

- Invoked after the component is rendered and inserted into the DOM.
- Commonly used for API calls, subscriptions, or DOM manipulations.



```
import React,{ Component } from 'react';
class LifeCycleMount extends Component{
    constructor(props){
        super(props)
        this.state={
            name:'tnr'
        }
        console.log("A")
    }
    static getDerivedStateFromProps(props,state){
        console.log("B")
        return null
    }
    componentDidMount(){
        console.log("D componentDidMount")
    }
    render(){
        console.log("C")
        return(
            <div>
            tnr</div>
            )
    }
}
export default LifeCycleMount;
```

Updating : when a component is being re-rendered as a result of changes to either its props or state.

static getDerivedStateFromProps(props, state):

- Same as in the mounting phase, called before each render.

shouldComponentUpdate(nextProps, nextState):

- Used for performance optimization. Returns true or false to determine if re-rendering is necessary.

render():

- Same as in the mounting phase.

getSnapshotBeforeUpdate(prevProps, prevState):

- Captures some information about the DOM (e.g., scroll position) before the update.

- Returned value is passed to componentDidUpdate.

componentDidUpdate(prevProps, prevState, snapshot):

- Called after the component is updated.
- Commonly used to perform DOM operations or network requests after an update.

Unmounting : When a component is being removed from DOM

componentWillUnmount():

- Used to clean up (e.g., cancel subscriptions, clear timers, etc.).

Error handling : when there is an error during rendering, in a lifecycle method, or in the constructor of any child component

Methods in Error Handling:

1. **static getDerivedStateFromError(error)**:
    - Updates state to display an error boundary.

Phase	Methods	Purpose
Mounting	`constructor()`	Initialize state, bind methods
	`static getDerivedStateFromProps()`	Sync state with props
	`render()`	Render JSX
	`componentDidMount()`	DOM setup, API calls
Updating	`static getDerivedStateFromProps()`	Sync state with props
	`shouldComponentUpdate()`	Optimize rendering
	`render()`	Render JSX
	`getSnapshotBeforeUpdate()`	Capture DOM info before update
	`componentDidUpdate()`	Perform updates after rendering
Unmounting	`componentWillUnmount()`	Cleanup
Error Handling	`static getDerivedStateFromError()`	Update UI for errors
	`componentDidCatch()`	Log errors

FRAGMENTS :

In React, fragments are used to group multiple elements without adding extra nodes to the DOM. Instead of wrapping elements in a or another wrapper, you can use a fragment to avoid unnecessary nesting in the DOM tree.

Why Use Fragments?

1. **Cleaner DOM Structure**: Prevents adding unnecessary wrapper elements like , which can complicate CSS styling or create unwanted layout issues.
2. **Improves Performance**: Fewer elements in the DOM mean better performance, especially in larger applications.

Using <React.Fragment>

```
1  import React, { Component } from 'react';
2  class FragmentA extends Component{
3      render(){
4          return(
5              <React.Fragment>
6              <h1>tnr</h1>
7              <h2>srm</h2>
8              </React.Fragment>
9              )
10     }
11 }
12 export default FragmentA;
```

Using the Short Syntax (<>...</>)

```
1  import React, { Component } from 'react';
2  class FragmentB extends Component{
3      render(){
4          return(
5              <>
6              <h1>tnr</h1>
7              <h2>srm</h2>
8              </>
9              )
10     }
11 }
12 export default FragmentB;
```

Limitations of Fragments

1. **Short Syntax Limitation: You cannot pass attributes like key to shorthand fragments (<>).**
2. **Not Supported Everywhere: Ensure your build tools and environment support JSX fragment syntax (<>).**

PURE COMPONENT :

A Pure Component in React is a component that only re-renders when there is a change in its props or state. It provides an optimized way to improve performance by avoiding unnecessary rendering.

 Shallow Comparison: Pure Components implement shouldComponentUpdate() with a shallow comparison of props and state.

 Performance Boost: By skipping unnecessary renders, Pure Components improve the app's performance in scenarios with complex UI or large datasets.

 Class-Based Only: Pure Components are a feature of class components. Functional components achieve similar optimization with React.memo.

 When a component renders the same output given the same props and state.

 For stateless UI components where performance matters.

- Component: Always re-renders, even if props or state don't change.
- PureComponent: Only re-renders if a shallow comparison detects changes.

Parent :

```
class Parent extends Component{
    constructor(props){
        super(props)
        this.state={
            name:'tnr'
        }
    }
    componentDidMount(){
        setInterval(()=>{
            this.setState({
                name:'tnr'
            })
        },2000)
    }
    render(){
        console.log("parent")
        return(
            <div>
            <h1>parent</h1>
            <Regular name={this.state.name}/>
            <Pure name={this.state.name}/>
            </div>
            )
    }
}
export default Parent;
```

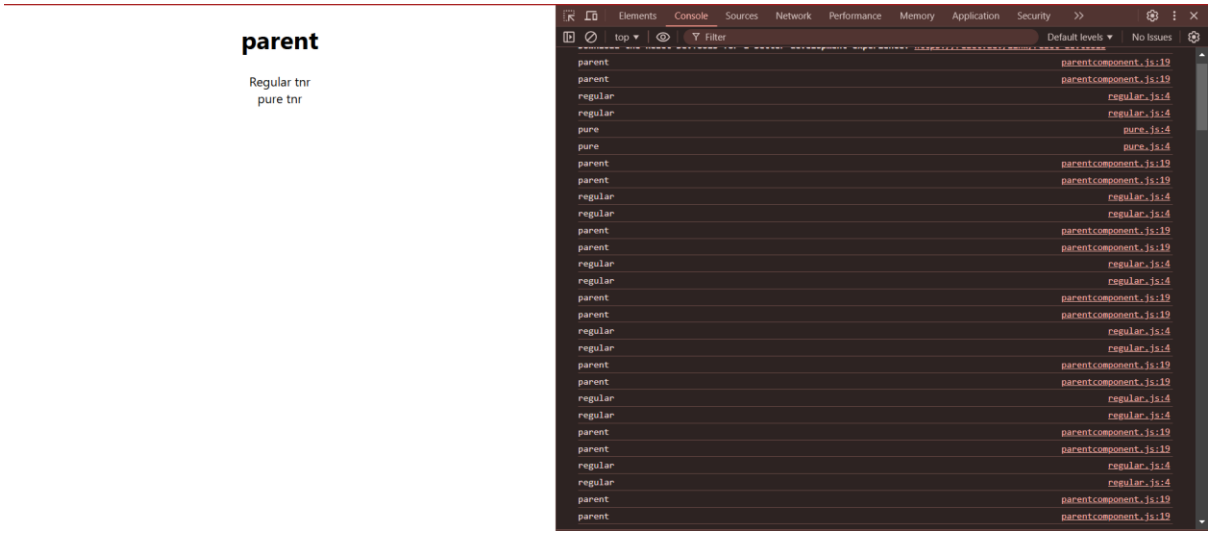
Regular :

```
import React,{ Component } from 'react';
class Regular extends Component{
    render(){
        console.log("regular")
        return(
            <div>Regular {this.props.name}</div>
            )
    }
}
export default Regular;
```

Pure:

```
import React,{ PureComponent } from 'react';
class Pure extends PureComponent{
    render(){
        console.log("pure")
        return(
            <div>pure {this.props.name}</div>
            )
    }
}
export default Pure;
```

Output :



 Use PureComponent when:

- Props and state are simple or immutable.
- You want to optimize performance.

 For nested objects or arrays:

- Use libraries like Immutable.js.
- Avoid mutating objects/arrays directly.

A pure component implements the shouldcomponentupdate lifecycle method by performing a shallow comparison on the props and state of the component.

If there is no difference, the component is not re-rendered-performance boost.

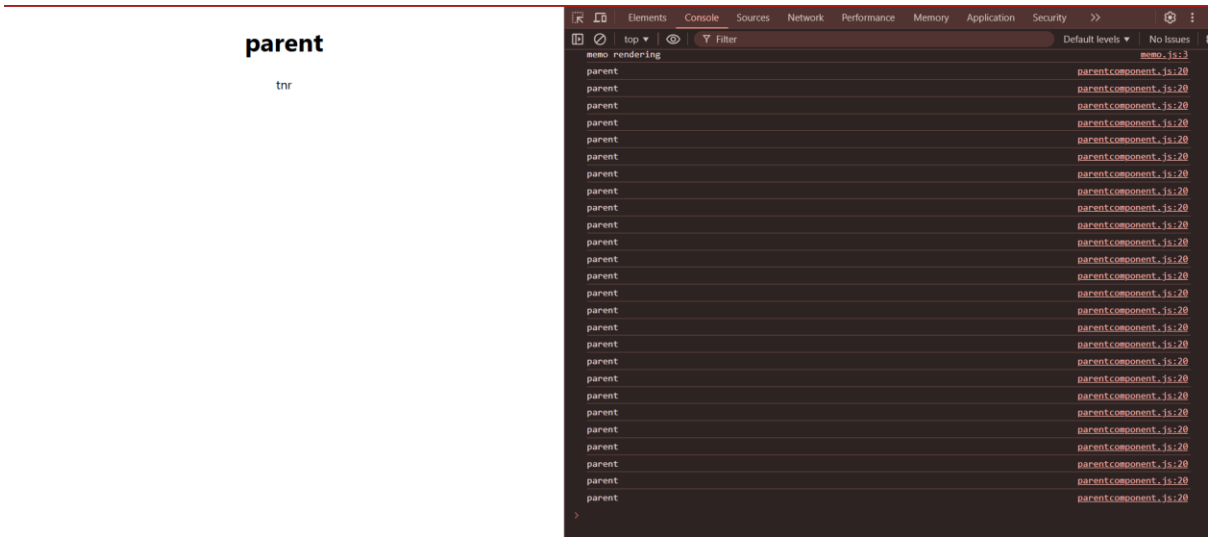
It is a good idea to ensure that all the children components are also pure to avoid unexpected behaviour.

Never mutate the state. Always return a new object that reflects the new state.

MEMO :

React.memo checks the props of a functional component. If the props remain the same between renders (shallow comparison), it skips the rendering of the component and reuses the last rendered output.

```
1  import React from 'react';
2  function Memo({name}){
3      console.log("memo rendering");
4      return(
5          <div>{name}</div>
6          )
7  }
8  export default React.memo(Memo);
```



Key Features of React.memo

1. **Optimization for Functional Components**: Avoids re-renders if props are unchanged.
2. **Shallow Comparison**: Like PureComponent, it performs a shallow comparison of the props.
3. **Custom Comparison Function**: You can provide a custom comparison function to define when the component should re-render.
4. • **Avoid Unnecessary Re-Renders**: For components with unchanged props in a frequently updating parent.

5. • **Performance Optimization**: When rendering a component is computationally expensive.

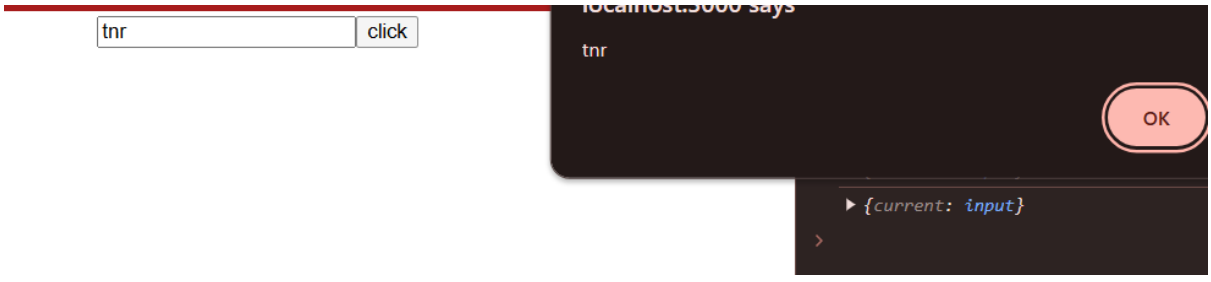
Refs(References):

In React, **refs** (short for references) provide a way to directly access and manipulate DOM elements or React components. They are helpful when you need to:

1. **Manage focus**, text selection, or media playback.
2. **Trigger animations**.
3. **Integrate with third-party DOM libraries**.

Refs.js

```
import React, { Component } from 'react';
class Refs extends Component{
    constructor(props){
        super(props)
        this.inputRef=React.createRef()

    }
    componentDidMount(){
        this.inputRef.current.focus()
        console.log(this.inputRef)
    }
    onclick=()=>{
        alert(this.inputRef.current.value)
    }
    render(){
        return(
            <div>
            <input type="text" placeholder="enter..." ref={this.inputRef}/>
            <button onClick={this.onclick}>click</button>
            </div>
            )
    }
}
export default Refs;
```



Callback refs:

```
class RefsDemo extends Component{
    constructor(props){
        super(props)
        this.inputRef=React.createRef()
        this.cbRef=null
        this.setcbref=(element)=>{
            this.cbref=element
        }
    }
    componentDidMount(){
        if(this.cbref){
            this.cbref.focus()
        }
        //this.inputRef.current.focus()
        //console.log(this.inputRef)
    }
    onclick=()=>{
        alert(this.inputRef.current.value)
    }
    render(){
        return(
            <div>
            <input type="text" placeholder="enter..." ref={this.inputRef}/>
            <input type="text" placeholder="enter..." ref={this.setcbref}/>

            <button onClick={this.onclick}>click</button>
            </div>
            )
```

class CallbackRefExample extends Component {

setInputRef = null;

componentDidMount() {

if (this.setInputRef) {

this.setInputRef.focus(); // Focus the input

}

}

render() {

return (

```
        <input
          type="text"
          ref={(element) => {
            this.setInputRef = element;
          }}
          placeholder="Callback Ref"
        />
      </div>
    );
  }
}

export default CallbackRefExample;
```

- **Avoid Overusing Refs**: Refs should not replace state or props for managing data flow.
- **Use sparingly** for direct DOM manipulation or imperative logic.
- **Prefer declarative approaches** for most UI updates (e.g., controlled components for forms).

Refs in class component:

Forwarding refs:

Ref forwarding is a feature in React that allows you to pass a ref from a parent component to a child component. This is particularly useful when the parent component needs direct access to the child component's DOM node or an instance of the child component.

```
import React, {Component} from 'react';
import FRInput from './forwardrefs';
class FRInputParent extends Component{
    constructor(props){
        super(props)
        this.inputRef=React.createRef()

    }
    change=()=>{
        this.inputRef.current.focus()
    }
    render(){
        return(
            <div>
            <FRInput ref={this.inputRef}/>
            <button onClick={this.chnage}>focus</button>
            </div>
            )
    }
}
export default FRInputParent;
```

```
import React from 'react';
/*function FRInput(){
    return (
        <div>
        <input type="text" />
        </div>
        )
}*/
const FRInput=React.forwardRef((props,ref)=>{
    return (
        <div>
        <input type="text" ref={ref}/>
        </div>
        )
})
export default FRInput;
```

 Use Forward Refs Sparingly: Only use refs for direct DOM manipulations or imperative logic.

 Combine with Functional Components: Ref forwarding works seamlessly with functional components.

 Avoid Overuse: Use React's declarative approach for most tasks instead of refs.

Portals :

Portals provide a way to render children into a DOM node that exists outside the DOM hierarchy of the parent component. This is useful when you want to render components like modals, tooltips, or dropdowns outside the parent DOM tree for styling or other reasons.

```
1  import React from 'react';
2  import ReactDom from 'react-dom';
3  const PortalDemo=()=>{
4      return ReactDom.createPortal(<h1>srm</h1>,
5          document.getElementById('tnr'))
6  }
7  export default PortalDemo;
```

Use Cases for Portals

1. Modals and Dialogs
2. Tooltips
3. Dropdown Menus

Event Bubbling in Portals

Even though the portal content is rendered outside the DOM hierarchy of its parent, **event bubbling** works as if the content is part of the React component tree.

1. **Use Portals When Necessary**: Avoid using portals unless you need to break out of the component's DOM hierarchy.
2. **Styling**: Ensure proper styling for components rendered in portals, especially when dealing with modals or overlays.
3. **Event Handling**: Be mindful of event propagation, especially with nested modals or dropdowns.

ERROR HANDLING

HIGHER ORDER FUNCTIONS

CONTEXT

React Context is a feature that allows you to share state or data across your component tree without having to pass props down manually at every level. It is particularly useful for managing "global" state, like user authentication, themes, or localization.

 Avoid "prop drilling" (passing props down through multiple levels of components).

 Centralize shared state or logic.

 Improve code readability and maintainability.

Create context

Provide context

Consume context

 Use Context for Global State: Don’t overuse context; it’s best for truly shared state like authentication, themes, or settings.

 Separate Contexts: Use multiple contexts if your app has distinct concerns.

 Combine with State Management: Use Context with useReducer or Redux for complex state logic.

 Memoize Values: Use React.memo or useMemo for performance optimization to avoid unnecessary renders.