Lab10 : Applying Styles Dynamically

Lab10 : Working Steps:

1) Copy Lab8 as Lab10
2) Remove All except the following files.
   - *index.html
   - *index.js
   - *index.css
   - *App.js
   - *App.css
3) Update App.js
4) Update App.css

Lab10: Files Required:

1. index.html	2. index.js
3. index.css	4. App.js
5. App.css	

1. index.html
2. index.js
3. index.css
4. App.js

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

import './App.css';
import 'bootstrap/dist/css/bootstrap.min.css';

class App extends Component {

state = {
name:"Srinivas Dande",
styleFlag:false,
classFlag:false
};

changeStyles = () => {
let myflag= this.state.styleFlag;

this.setState({
styleFlag: !myflag
});

}
```

```
changeClasses = () => {
let myflag= this.state.classFlag;

this.setState({
classFlag: !myflag
});

}

render(){

let mystyles={
color:'blue',
fontSize:'25px',
border:'2px solid red',
borderRadius:'5px',
padding:'10px',
fontFamily:'Cambria'
};

if(this.state.styleFlag===true){
    mystyles = {
    color:'blue',
    fontSize:'25px',
    border:'5px solid red',
    borderRadius:'15px',
    padding:'10px'
    };
}else{
    mystyles={
    color:'red',
    fontSize:'25px',
    border:'2px solid blue',
    borderRadius:'5px',
    padding:'10px'
    };
}

//Define Array of CSS Classes
let myclassList = ["myclass1","myclass2"];
console.log(myclassList.join(" "));

if(this.state.classFlag==false){
    myclassList.slice(2,1);
    myclassList.push("myred");
}else{
     myclassList.slice(2,1);
     myclassList.push("myblue");
}

return (
<div className="container">
<h2 className="text-center"> Welcome to Java Learning Center!!!</h2>
<br/>
<p> 1. Appying Styles -Static Way</p>
<div  style={{color:'red',fontSize:'25px',border:'2px solid
blue',borderRadius:'5px',padding:'10px',fontFamily:'Cambria'}}>
You are learning React 16 from { this.state.name}!!!
</div>
<br/>
<p> 2. Appying Styles -Dynamic Way</p>
<div  style={mystyles}>
You are learning React 16 from { this.state.name}!!!
</div>
<br/>
<button onClick={this.changeStyles} className="btn btn-success"> Change
Styles</button>
<br/>

<br/>
<p> 3. Appying Classes -Static Way</p>
<div className="myclass1 myclass2 myblue">
You are learning React 16 from { this.state.name}!!!
</div>
<br/>

<br/>
<p> 4. Appying Classes -Dynamic Way</p>
<div className={myclassList.join(" ")}>
You are learning React 16 from { this.state.name}!!!
</div>
<br/>
<button onClick={this.changeClasses} className="btn btn-success"> Change
Classes</button>
<br/>
</div>
);
}
}

export default App;
```

5. App.css

```
.myclass1{
  padding: 10px;
  background-color: lightcyan;
 }

 .myclass2{
  font-size: 25px;
  font-family: Cambria;
 }

 .myred{
  color:red;
  border:2px solid blue;
  border-radius: 5px;
 }

 .myblue{
  color:blue;
  border:5px solid red;
  border-radius: 15px;
 }

 p{
  font-family: Cambria;
  font-size:25px;
  color:black;
  font-weight: bold;
  margin-top:25px;
 }
```

Lab11 : Managing State with setState() method for Class Components

Lab11: Files Required:

1. index.html	2. index.js
3. index.css	4. App.js
5. MyCourses.js	

1. **index.html**
2. **index.js**
3. **index.css**
4. **App.js**

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

import 'bootstrap/dist/css/bootstrap.min.css';
import MyCourses from './MyCourses';

class App extends Component {

  render() {
   return (
    <div className="container">
    <h1 className="text-center"> Welcome to JLC!!!</h1>
    <br/>
        <MyCourses/>
    </div>

     );
  }
}

export default App;
```

5. **MyCourses.js**

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


 class MyCourses extends Component {


  state = {
    trainerName:"Srinivas Dande",
    trainerEmail:"sri@jlc.com",
    mycourseList : [
      {cid:101,cname:"Angular 7",price:15000,active:true},
      {cid:102,cname:"React 16",price:15000,active:true},
      {cid:103,cname:"Java FSD",price:35000,active:false},
      {cid:104,cname:"MicroServices",price:25000,active:true},
  {cid:105,cname:"DevOps",price:15000,active:false},
 ]
};

showActiveCourses = ()=> {
 console.log("showActiveCourses - called");

 let activeCourses = this.state.mycourseList.filter(mycourse => mycourse.active === true);

 // 3 properties
 this.setState({
  trainerEmail:"srinivas@jlc.com",
  mycourseList : activeCourses
 });
 //setState() merges the new State with Current State
// 3 properties
}

showAllCourses = ()=> {
 console.log("showAllCourses - called");

 let allCourses =  [
  {cid:101,cname:"Angular 7",price:15000,active:true},
  {cid:102,cname:"React 16",price:15000,active:true},
  {cid:103,cname:"Java FSD",price:35000,active:false},
  {cid:104,cname:"MicroServices",price:25000,active:true},
  {cid:105,cname:"DevOps",price:15000,active:false},
 ];

 // 3 properties
 this.setState({
  trainerEmail:"sri@jlc.com",
  mycourseList : allCourses
 });
 //setState() merges the new State with Current State
// 3 properties
}
componentDidMount(){
 this.showAllCourses();
}
```

```
render() {
let courseListToDisplay = this.state.mycourseList.map(
(mycourse) => (
<tr>
<td> <h6> {mycourse.cid} </h6></td>
<td> <h6> {mycourse.cname} </h6></td>
<td> <h6> {mycourse.price} </h6></td>
<td> <h6> {mycourse.active} </h6></td>
</tr>
)
);

return (
<div className="container">
<div className="container">
<button onClick={this.showAllCourses} className="btn btn-success mybutton btn-lg">
All Courses
</button>
<button onClick={this.showActiveCourses} className="btn btn-success mybutton btn-lg">
Active Courses
</button>
</div>
<br/><br/>
<table>
<thead>
<tr>  <th> Course ID</th>
<th> Course Name</th>
<th> Price</th>
<th> Is Active</th>   </tr>
</thead>
<tbody>  {courseListToDisplay}   </tbody>
</table>
<h3> Trainer name : {this.state.trainerName} </h3>
<h3> Trainer Email : {this.state.trainerEmail} </h3>
</div>
)

}
}
export default MyCourses;
```

Lab12 : Managing State with useState() React Hook for functional components

Lab12: Files Required:

1. index.html	2. index.js
3. index.css	4. App.js
5. MyCourses.js	

1. index.html

2. index.js

3. index.css

4. App.js

```
import React from 'react';

import 'bootstrap/dist/css/bootstrap.min.css';
import MyCourses from './MyCourses';

const App = () =>  {

   return (
    <div className="container">
    <h1 className="text-center"> Welcome to JLC !!!</h1>
    <br/>
        <MyCourses/>
    </div>

     );
  }

export default App;
```

5. MyCourses.js

```
import React, { useState } from 'react'

 const MyCourses = () => {

   // const [state, setState] = useState(initialState);

   //1.Using the useState() React Hook
   const [courseState,setCourseState] = useState(
    {
     trainerName:"Srinivas Dande",
     trainerEmail:"sri@jlc.com",
     mycourseList : [
       {cid:101,cname:"Angular 7",price:15000,active:true},
       {cid:102,cname:"React 16",price:15000,active:true},
       {cid:103,cname:"Java FSD",price:35000,active:false},
      {cid:104,cname:"MicroServices",price:25000,active:true},
      {cid:105,cname:"DevOps",price:15000,active:false},
    ]
  }
 );


 const showActiveCourses = ()=> {
   console.log("showActiveCourses - called");

   let activeCourses = courseState.mycourseList.filter(mycourse => mycourse.active
=== true);

   // 3 properties
   setCourseState({
     trainerEmail:"srinivas@jlc.com",
     mycourseList : activeCourses
   });

   //useState() replaces the new State with Current State
   // 2 properties
 }

 const showAllCourses = ()=> {
   console.log("showAllCourses - called");

   let allCourses =  [
     {cid:101,cname:"Angular 7",price:15000,active:true},
     {cid:102,cname:"React 16",price:15000,active:true},
     {cid:103,cname:"Java FSD",price:35000,active:false},
     {cid:104,cname:"MicroServices",price:25000,active:true},
     {cid:105,cname:"DevOps",price:15000,active:false},
   ];

   // 3 properties
   setCourseState({
     trainerName:"Srinivas Dande",
     trainerEmail:"sri@jlc.com",
     mycourseList : allCourses
   });
   //useState()  Hook replaces the new State with Current State
  // 2 properties

 }
```

```
    let courseListToDisplay = courseState.mycourseList.map(
      (mycourse) => (
        <tr>
          <td> <h6> {mycourse.cid} </h6></td>
          <td> <h6> {mycourse.cname} </h6></td>
          <td> <h6> {mycourse.price} </h6></td>
          <td> <h6> {mycourse.active} </h6></td>
        </tr>
      )
    );

    let trainerInfo = null;
    if(courseState.trainerName){
      trainerInfo = ( <h3> Trainer name : {courseState.trainerName} </h3>);
    }

  return (

    <div className="container">
     <div className="container">
<button onClick={showAllCourses} className="btn btn-success mybutton btn-lg">
All Courses   </button>
<button onClick={showActiveCourses} className="btn btn-success mybutton btn-
lg">  Active Courses   </button>
     </div>
<br/>     <br/>
<table>
<thead>
<tr>  <th> Course ID</th>
<th> Course Name</th>
<th> Price</th>
<th> Is Active</th>   </tr>
</thead>
<tbody>  {courseListToDisplay}   </tbody>
</table>
 { trainerInfo }
<h3> Trainer Email : {courseState.trainerEmail} </h3>
    </div>
     )

   }

export default MyCourses;
```

Lifecycle Hooks

- Each Class component has several **Properties and lifecycle methods** that you can override to run code at different stages of the component lifecycle.

A) Instance Properties

- Each component has the following Instance Properties:
  1. **props**
  2. **state**

B)Class Properties

- Each component has the following Class Properties:
  1. **defaultProps**
  2. **displayName**

C) Other APIs

- Each component also provides some other APIs:
  1. **setState()**
  2. **forceUpdate()**

D)Mounting

- Following methods will called when an component instance is created and added to DOM:
  1. **constructor()**
  2. **static getDerivedStateFromProps()**
  3. **render()**
  4. **componentDidMount()**

Note:

- Following method is considered legacy and you should avoid them in new code:

  componentWillMount()

E)Unmounting

- This method is called when a component is being removed from the DOM:
  1. **componentWillUnmount()**

F)Updating

- An update can be caused by changes to props or state.
- Following methods will be called when a component is being re-rendered:

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

Note:

- Following methods are considered legacy and you should avoid them in new code:

  componentWillUpdate()

  componentWillReceiveProps()

G)Error Handling

- Following methods will be called when there is an error in a lifecycle method, or in the constructor of any child component during rendering,
  1. **static getDerivedStateFromError()**
  2. **componentDidCatch()**

state	▪ Represents Component State
props	▪ Represents Component Props
defaultProps	▪ Represents Default Props of Component
displayName	▪ Represents Display Name of Component
setState()	▪ To Change the State
forceUpdate()	▪ Calling forceUpdate() will cause render() to be called on the component
constructor()	▪ Called when an component instance is created and added to DOM
static getDerivedStateFromProps()	▪ Called when an component instance is created and added to DOM ▪ Called when a component is being re-rendered
render()	▪ Called when an component instance is created and added to DOM ▪ Called when a component is being re-rendered
componentDidMount()	▪ Called when an component instance is created and added to DOM
shouldComponentUpdate()	▪ Called when a component is being re-rendered
getSnapshotBeforeUpdate()	▪ Called when a component is being re-rendered
componentDidUpdate()	▪ Called when a component is being re-rendered
componentWillUnmount()	▪ Called when a component is being removed from DOM
static getDerivedStateFromError()	▪ Called after an error has been thrown by a descendant component
componentDidCatch()	▪ Called after an error has been thrown by a descendant component
componentWillMount()	▪ Legacy Lifecycle Method
componentWillUpdate()	▪ Legacy Lifecycle Method
componentWillReceiveProps()	▪ Legacy Lifecycle Method

Instance Properties

state

- The state contains data specific to this component that may change over time.
- The state is user-defined, and it should be a plain JavaScript object.
- If some value isn't used for rendering , you don't have to put it in the state. Such values can be defined as fields on the component instance.
- Never mutate this.state directly, as calling setState() afterwards may replace the mutation you made.

props

- this.props contains the props that were defined by the caller of this component.
- In particular, this.props.children is a special prop, typically defined by the child tags in the JSX expression rather than in the tag itself.

Class Properties

defaultProps

- defaultProps can be defined as a property on the component class itself, to set the default props for the class.
- This is used for undefined props, but not for null props.

Ex:

```
class CustomButton extends React.Component {
        // ...
}

CustomButton.defaultProps = {
  color: 'blue'
};
```

If props.color is not provided, it will be set by default to 'blue':

```
  render() {
    return <CustomButton /> ; // props.color will be set to blue
  }
```

If props.color is set to null, it will remain null:

```
  render() {
    return <CustomButton color={null} /> ; // props.color will remain null
  }
```

displayName

- The displayName string is used in debugging messages.
- Usually, you don't need to set it explicitly because it's inferred from the name of the function or class that defines the component.
- You might want to set it explicitly if you want to display a different name for debugging purposes

constructor(props)

- If you don't want to initialize state and you don't want to bind methods, you no need to implement a constructor for your React component.
- The constructor for a React component is called before it is mounted.
- When you implement the constructor, you should call super(props) before any other statement. Otherwise, this.props will be undefined in the constructor, which can lead to bugs.
- Mainly, in React constructors are only used for two purposes:

  a) Initializing local state by assigning an object to this.state.

  b) Binding event handler methods to an instance.
- You should not call setState() in the constructor().
- Constructor is the only place where you should assign this.state directly.
- In all other methods, you need to use this.setState() instead.

  Ex:

```
constructor(props) {
  super(props);
  // Don't call this.setState() here!
  this.state = { message: '' };
  this.clickHandler = this.clickHandler.bind(this);
}
```

- Avoid introducing any side-effects or subscriptions in the constructor.
- For those use cases, use componentDidMount() instead.
- Avoid copying props into state! This is a common mistake:

  Ex:

```
constructor(props) {
 super(props);
 // Don't do this!
 this.state = { color: props.color };
}
```

render()

- The render() method is the only required method in a class component.
- When render() method is called, it checks this.props and this.state and returns React elements typically created via JSX.
- The render() function should be pure, meaning that it does not modify component state, it returns the same result each time it's invoked, and it does not directly interact with the browser.
- If you need to interact with the browser, perform your work in componentDidMount() or the other lifecycle methods instead. Keeping render() pure makes components easier to think about.

Note

- render() will not be invoked if shouldComponentUpdate() returns false.

componentDidMount()

- componentDidMount() is invoked immediately after a component is mounted (inserted into the DOM Tree).
- If you need to load data from a remote endpoint, this is a good place to instantiate the network request.
- This method is a good place to set up any subscriptions.
- If you do that, don't forget to unsubscribe in componentWillUnmount().

componentDidUpdate(prevProps, prevState, snapshot)

- componentDidUpdate() is invoked immediately after updating occurs.
- This method is not called for the initial render.
- Use this as an opportunity to operate on the DOM when the component has been updated.
- This is also a good place to do network requests as long as you compare the current props to previous props (e.g. a network request may not be necessary if the props have not changed).

```
componentDidUpdate(prevProps) {
  // Typical usage (don't forget to compare props):
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

- You may call setState() immediately in componentDidUpdate() but note that it must be wrapped in a condition like in the example above, or you'll cause an infinite loop.
- If your component implements the getSnapshotBeforeUpdate() lifecycle (which is rare), the value it returns will be passed as a third "snapshot" parameter to componentDidUpdate(). Otherwise this parameter will be undefined.

Note

- componentDidUpdate() will not be invoked if shouldComponentUpdate() returns false.

5) componentWillUnmount()

- componentWillUnmount() is invoked immediately before a component is unmounted and destroyed. Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in componentDidMount().

- You should not call setState() in componentWillUnmount() because the component will never be re-rendered. Once a component instance is unmounted, it will never be mounted again.

6) shouldComponentUpdate(nextProps, nextState)

- The default behavior is to re-render on every state change, and in the vast majority of cases you should rely on the default behavior.

- shouldComponentUpdate() is invoked before rendering when new props or state are being received. Defaults to true. This method is not called for the initial render or when forceUpdate() is used.

- This method only exists as a performance optimization. Do not rely on it to “prevent” a rendering, as this can lead to bugs.

- Consider using the built-in PureComponent instead of writing shouldComponentUpdate() by hand. PureComponent performs a shallow comparison of props and state, and reduces the chance that you’ll skip a necessary update.

- If you are confident you want to write it by hand, you may compare this.props with nextProps and this.state with nextState and return false to tell React the update can be skipped. Note that returning false does not prevent child components from re-rendering when their state changes.

- We do not recommend doing deep equality checks or using JSON.stringify() in shouldComponentUpdate(). It is very inefficient and will harm performance.

- Currently, if shouldComponentUpdate() returns false, then componentWillUpdate(), render(), and componentDidUpdate() will not be invoked.

- In the future React may treat shouldComponentUpdate() as a hint rather than a strict directive, and returning false may still result in a re-rendering of the component.

static getDerivedStateFromProps(props, state)

- This method will be called right before calling the render method.
- This methid will be called both on the initial mount and on subsequent updates.
- It should return an object to update the state, or null to update nothing.
- This method exists for rare use cases where the state depends on changes in props over time.
- If you need to perform a side effect (for example, data fetching or an animation) in response to a change in props, use componentDidUpdate lifecycle instead.
- This method doesn’t have access to the component instance.

getSnapshotBeforeUpdate(prevProps, prevState)

- getSnapshotBeforeUpdate() is invoked right before the most recently rendered output is committed to e.g. the DOM. It enables your component to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this lifecycle will be passed as a parameter to componentDidUpdate().

- This use case is not common, but it may occur in UIs like a chat thread that need to handle scroll position in a special way.

- A snapshot value (or null) should be returned.

forceUpdate()

- By default, when your component's state or props change, your component will re-render.
- If your render() method depends on some other data, you can tell React that the component needs re-rendering by calling forceUpdate().

- Calling forceUpdate() will cause render() to be called on the component, skipping shouldComponentUpdate().
- This will trigger the normal lifecycle methods for child components, including the shouldComponentUpdate() method of each child.
- React will still only update the DOM if the markup changes.

- Normally you should try to avoid all uses of forceUpdate() and only read from this.props and this.state in render().

Error boundaries

- Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.
- Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.
- A class component becomes an error boundary if it defines either (or both) of the lifecycle methods static getDerivedStateFromError() or componentDidCatch().
- Updating state from these lifecycles lets you capture an unhandled JavaScript error in the below tree and display a fallback UI.
- Only use error boundaries for recovering from unexpected exceptions; don't try to use them for control flow.

Note

- Error boundaries only catch errors in the components below them in the tree.
- An error boundary can't catch an error within itself.

static getDerivedStateFromError(error)

- This lifecycle is invoked after an error has been thrown by a descendant component.
- It receives the error that was thrown as a parameter and should return a value to update state.

  Ex:

```
class ErrorBoundary extends React.Component {
 constructor(props) {
  super(props);
  this.state = { hasError: false };
 }

 static getDerivedStateFromError(error) {
  // Update state so the next render will show the fallback UI.
  return { hasError: true };
 }

 render() {
  if (this.state.hasError) {
   // You can render any custom fallback UI
   return <h1>Something went wrong.</h1>;
  }
       ....
 }
}
```

Note

- getDerivedStateFromError() is called during the “render” phase, so side-effects are not permitted.
- For those use cases, use componentDidCatch() instead.

componentDidCatch(error, info)

- This lifecycle is invoked after an error has been thrown by a descendant component.
- It receives two parameters:
  a) **error -** The error that was thrown.
  b) **info -** An object with a componentStack key containing information about which component threw the error.
- componentDidCatch() is called during the “commit” phase, so side-effects are permitted.
- It should be used for things like logging errors:

  Ex:

```
class ErrorBoundary extends React.Component {
 constructor(props) {
  super(props);
  this.state = { hasError: false };
 }
```

```
   static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
   }

   componentDidCatch(error, info) {
    // Example "componentStack":
    //   in ComponentThatThrows (created by App)
    //   in ErrorBoundary (created by App)
    //   in div (created by App)
    //   in App
    console.log("[App] - componentDidCatch", info.componentStack);
   }

   render() {
    if (this.state.hasError) {
     // You can render any custom fallback UI
     return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
   }
  }
```

Note

- In the event of an error, you can render a fallback UI with componentDidCatch() by calling setState.
- but this will be deprecated in a future release. Use static getDerivedStateFromError() to handle fallback rendering instead.

Legacy Lifecycle Methods

- The lifecycle methods below are marked as legacy.
- We don't recommend using them in the new code.

```
componentWillMount()
componentWillReceiveProps()
componentWillUpdate()
```

Lab13 : LifeCycle Hooks for Class Component

Lab13: Files Required:

1. index.html	2. index.js
3. index.css	4. App.js
5. Hello.js	

1. index.html

2. index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import 'bootstrap/dist/css/bootstrap.min.css';
import './index.css';
import App from './App';

ReactDOM.render(<App />, document.getElementById('myroot'));
```

3. index.css

4. App.js

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

import Hello from './Hello';

class App extends Component {


static displayName = "MyAppComponent";

 constructor(props){
  super(props);
  console.log("[App] - constructor()");
  this.state = {
    companyName : "Java Learning Center",
  }
 }

 render() {
  console.log("[App] - render()");
  console.log("[App] - ",App.displayName);
  console.log("[App] - ",App.defaultProps);
  console.log("[App] - ",this.state);
  console.log("[App] - ",this.props);

  return (
  <div className="container">
  <h1 className="text-center"> Welcome to JLC!!!</h1>
```

```
        <br/>
        <Hello/>
        <Hello mytrainer="Dande"/>
        <Hello mytrainer="Srinivas" mycolor="Pink"/>
        </div>

        );
      }
    }

    export default App;
```

5. Hello.js

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

class Hello extends Component {
  static displayName = "MyHelloComponent";
  static defaultProps = {
      mytrainer : "Srinivas Dande",
      mycolor : "Red"
  }

 constructor(props){
   super(props);
   console.log(1,"[Hello] - constructor()");
   this.state = {
    message : "Ok Guys"
   }
 }

 static getDerivedStateFromProps(myprops,mystate){
   console.log(2,"[Hello] - getDerivedStateFromProps()");
  // console.log("[Hello] - ",myprops);
   //console.log("[Hello] - ",mystate);

   return {
      message: "Updated Message here ",
      trainer: myprops.mytrainer,
      color:myprops.mycolor
    };
   }
 render() {
   console.log(3,"[Hello] - render()");
   //console.log("[Hello] - ",Hello.displayName);
   //console.log("[Hello] - ",Hello.defaultProps);
   // console.log("[Hello] - ",this.props);
   //console.log("[Hello] - ",this.state);

  return (
   <div className="container">
   <h3 className="text-center"> I am Hello Component!!!</h3>
   <h4> My Trainer  : {this.props.mytrainer} </h4>
   <h4> My Color : {this.props.mycolor}</h4>
   <br/>

   </div>

    );
 }
}

export default Hello;
```