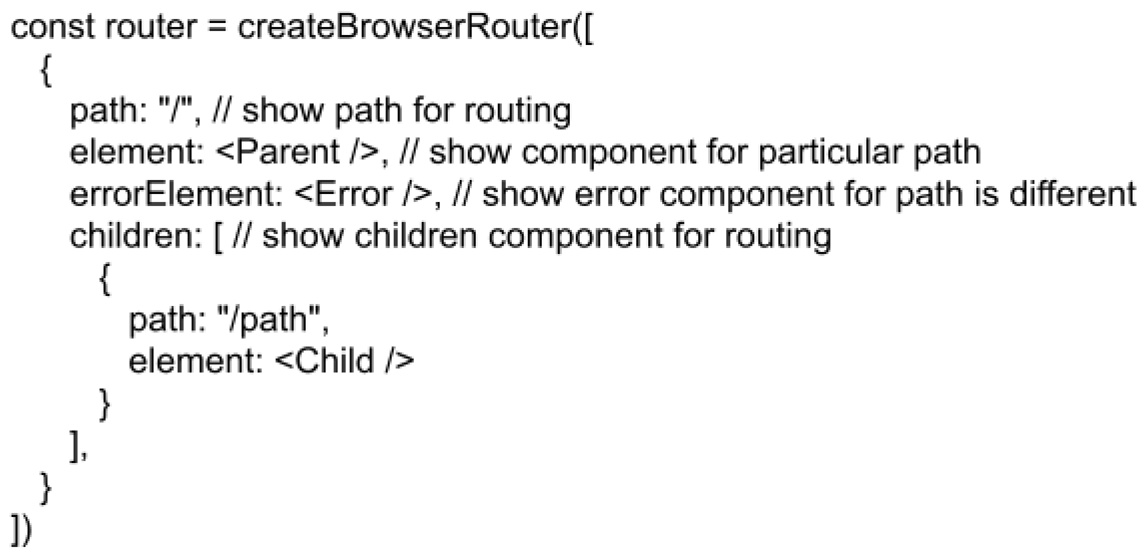
Theory Assignment:

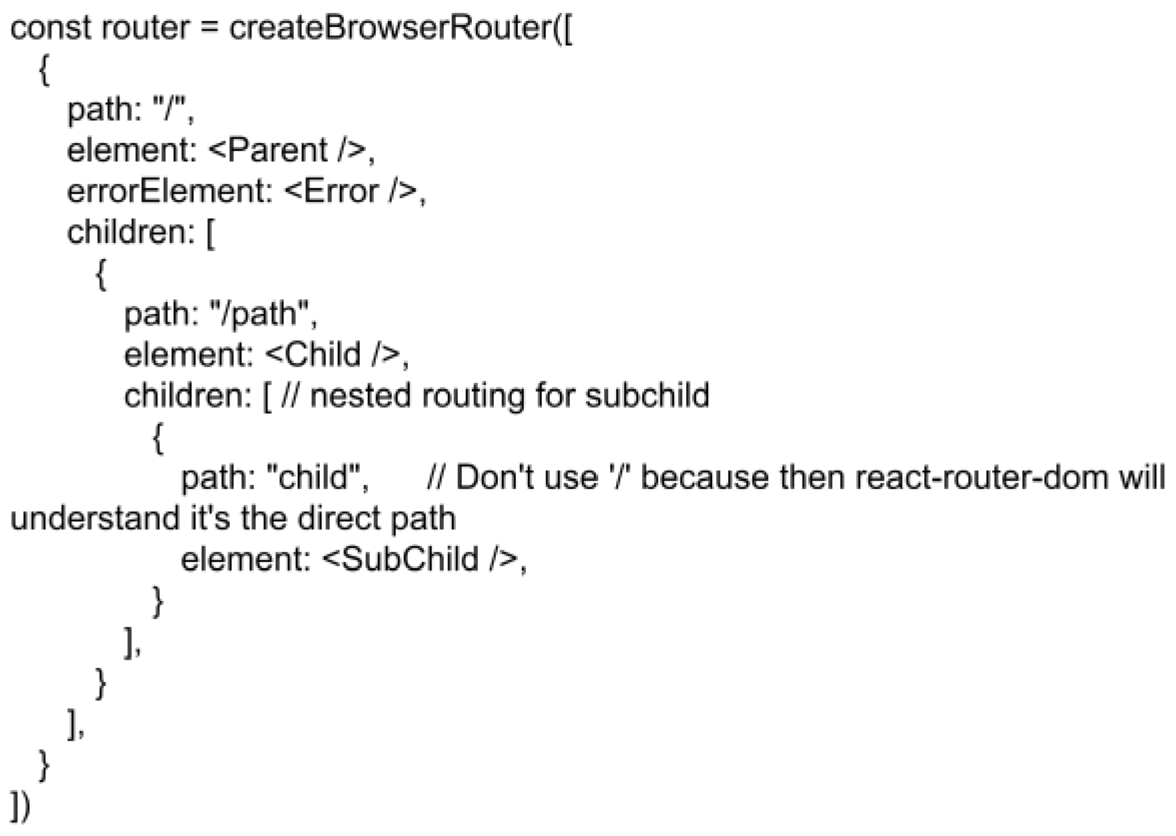
● **How do you create Nested Routes react-router-dom configuration**

We can create a Nested Routes inside a react router configuration as follows: first call createBrowserRouter for routing different pages



```
const router = createBrowserRouter([
  {
    path: "/", // show path for routing
    element: <Parent />, // show component for particular path
    errorElement: <Error />, // show error component for path is different
    children: [ // show children component for routing
      {
        path: "/path",
        element: <Child />
      }
    ],
  }
])
```

Now we can create a nested routing for /path using children again as follows:



```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Parent />,
    errorElement: <Error />,
    children: [
      {
        path: "/path",
        element: <Child />,
        children: [ // nested routing for subchild
          {
            path: "child",      // Don't use '/' because then react-router-dom will
understand it's the direct path
            element: <SubChild />,
          }
        ],
      }
    ],
  }
])
```

● **Read abt createHashRouter, createMemoryRouter from React Router docs.**
createHashRouter is useful if you are unable to configure your web server to direct all traffic to your React Router application. Instead of using normal URLs, it will use the hash (#) portion of the URL to manage the "application URL". Other than that, it is functionally the same as createBrowserRouter.

createMemoryRouter Instead of using the browsers history a memory router manages it's own history stack in memory. It's primarily useful for testing and component development tools like Storybook, but can also be used for running React Router in any non-browser environment.

● **What is the order of life cycle method calls in Class Based Components**

Constructor - The constructor method is the first to be called when a component is created. It's where we typically initialize the component's state and bind event handlers.

Render - The render method is responsible for rendering the component's UI. It must return a React element (typically JSX representing the component's structure.

ComponentDidMount - This method is called immediately after the component is inserted into the DOM. It's often used for making AJAX requests, setting up subscriptions, or other one-time initializations.

ComponentDidUpdate - This method is called after the component has been updated (re-rendered) due to changes in state or props. It's often used for side effects, like updating the DOM in response to state or prop changes.

ComponentWillUnmount - This method is called just before the component is removed from the DOM. It's used to clean up resources or perform any necessary cleanup.

● **Why do we use componentDidMount?**
Here are some common use cases for componentDidMount:

Fetching Data - It's often used to make asynchronous requests to fetch data from APIs or external sources. This is a common scenario for components that need to display dynamic content.

DOM Manipulation - When we need to interact with the DOM directly, such as selecting elements, setting attributes, or applying third-party libraries that require DOM elements to be present, we can safely do so in componentDidMount. This is because the component is guaranteed to be in the DOM at this point.

● **Why do we use componentWillUnmount? Show with example**
The componentWillUnmount lifecycle method in React class-based components is used to perform cleanup and teardown tasks just before a component is removed from the DOM. It's

a crucial part of managing resources and subscriptions to prevent memory leaks and ensure that the component's behavior is properly cleaned up.

Here's an example of using componentWillUnmount to remove an event listener when a component is unmounted:

```
class MyComponent extends React.Component {
 constructor() {
 super();
 this.handleResize = this.handleResize.bind(this);
 }
 componentDidMount() {
 // Add a window resize event listener when the component
is mounted
 window.addEventListener('resize', this.handleResize);
 }
 componentWillUnmount() {
 // Remove the window resize event listener when the component is unmounted
 window.removeEventListener('resize', this.handleResize);
 }
 handleResize(event) {
 // Handle the resize event
 console.log('Window resized:', event);
 }
 render() {
 return <div>My Component</div>;
 }
}
```

In this example, the component adds a resize event listener to the window when it's mounted, and it removes that listener in the componentWillUnmount method.

● **(Research) Why do we use super(props) in constructors?**
super(props) is used to inherit the properties and access variables of the React parent class when we initialize our component. super() is used inside the constructor of a class to derive the parent's all properties inside the class that extended it. If super() is not used, then Reference Error : Must call super constructor in derived classes before accessing 'this' or returning from derived constructor is thrown in the console. The main difference between super() and super(props) is the this.props is undefined in child's constructor in super() but this.props contains the passed props if super(props) is used.

● **(Research) Why can't we have the callback function of useEffect async?**
useEffect expects it's callback function to return nothing or return a function (cleanup function that is called when the component is unmounted). If we make the callback function as async, it will return a promise and the promise will affect the clean-up function from being called.