**JSX Limitations**

* We can only send single root level JSX elements in return. It should be wrapped in some kind of tag.
* Even react.createElement would not work for multiple elements. It needs to be nested as children of one element.
* Instead of using a div tag, we can use a native JS array and have commas in between to separate as per need.
* We need to add key manually, wrapping in an array, separate components with commas etc, instead of using a div.
* But <div> tag creates a soup of div. Unnecessary divs which are only needed for wrapping, it creates a lot of loading lag, due to rendering of unnecessary elements.
* **Solution? Trick:**
  + **Helper Component:** an empty component which only returns the content between it’s tags. Wrapper component for using it as a wrapper, which returns as props.children. It returns all the content between opening and closing tags.

<Fragment>,  <> & <Wrapper>

are not made from a DOM node.

DOM elements (like a <div>) are.

A DOM node requires more processing than

a virtual wrapper like:  <Fragment>,  <> & <Wrapper>.

It's not a big savings,  but,  "why not!!!".

- - - - - - - - - - - - - - - - - - - - - - - - - - -

 The only attribute (at this time) <Fragment> accepts is key.

https://reactjs.org/docs/fragments.html#keyed-fragments

- - - - - - -

<> .... </> syntax does not accept attributes.

Not even the key attribute.

- - - - - - -

Short-syntax: https://reactjs.org/docs/fragments.html#short-syntax

<> .... </> doesn't require an additional import.

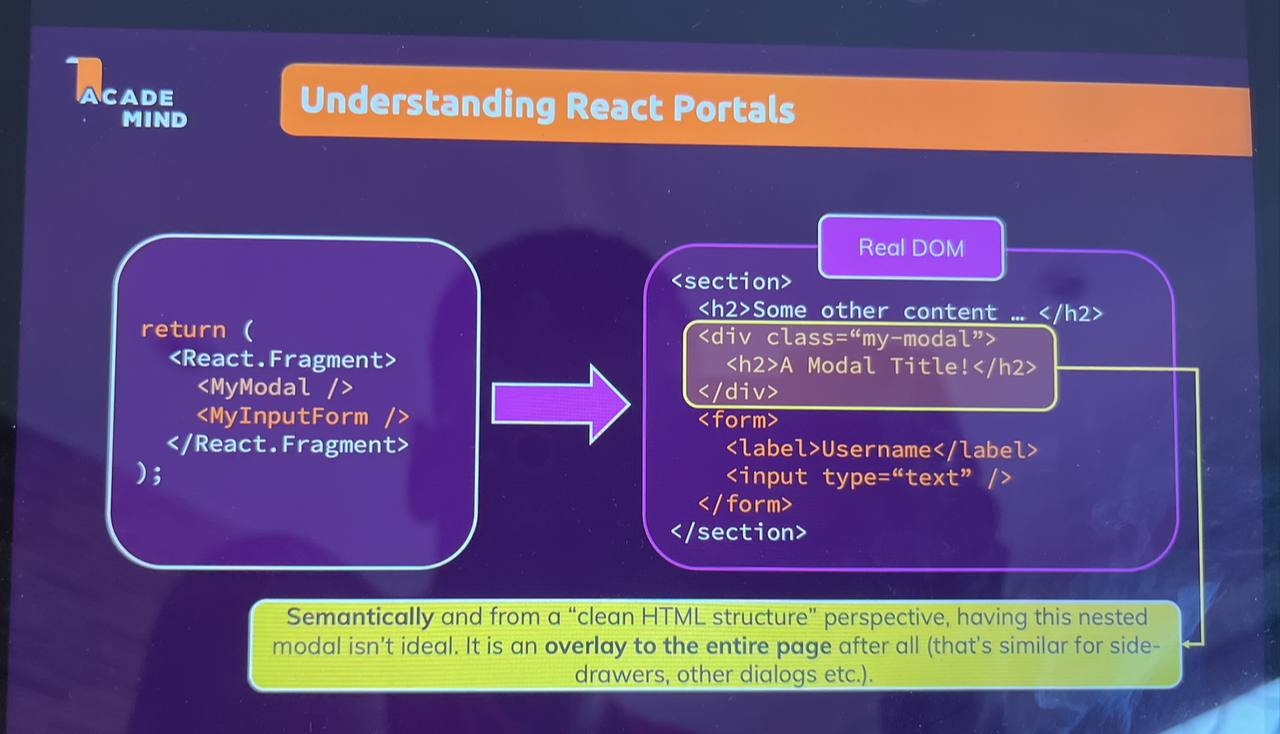
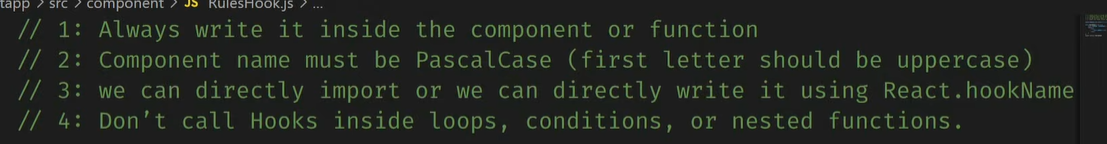
---------------------------------------

Wrapper is a "random" (self-evident)

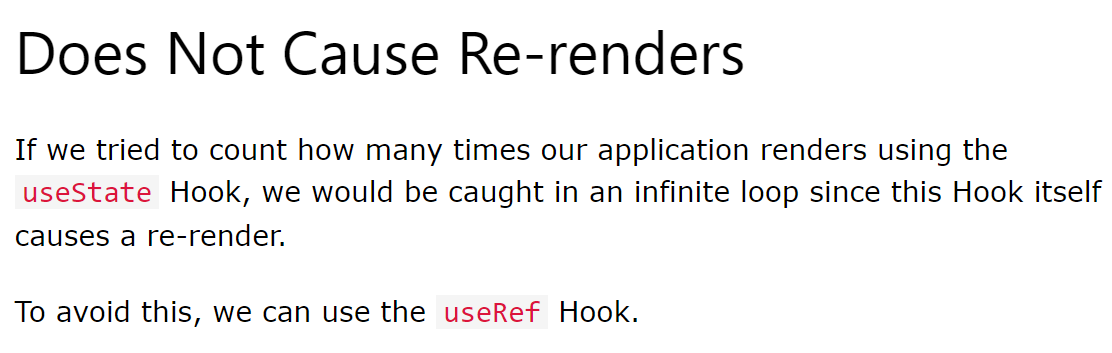
component name Mr. Schwarzmüller chose.

Wrapper isn't a keyword.

Wrapper is a very simple higher order component (HOC).

* + **Empty Wrapper or Fragment: in-built wrapper default component.**
  + Your project set up needs to support the <> </> empty wrapper syntax in jsx. It is not a usually used wrapper.
* **React.Fragment** always works, but <> </> might not always work, but both of these are the substitutes for using inbuilt wrapper components, which are not DOM nodes, and saves processing time for us.
* **REACT PORTALS:** [**Amazing link to understand**](https://blog.logrocket.com/learn-react-portals-example/#:~:text=custom%20React%20Hook-,What%20is%20React%20Portals%3F,-React%20Portals%20are)
* 
* [Use-cases of react portal](file:///E:\Tech\AngelaYu%20WebDev\Notes\Max%20Notes\What%20Are%20All%20the%20Things%20React%20Portals%20Can%20Do%3f)
* React Portals is a feature in the React JavaScript library that provides a way to render a React component at a different location in the DOM instead of where it is declared. It is useful when you want to have a parent-child relationship in your components while you want to render a child component outside of the parent.
* To use React Portals, you will need to create a container element in the normal HTML DOM you will use as the target for your React content. It can be any valid HTML element or renderable React children, such as a div, span, section, string, or fragment.
* *Extra: React hook rules..*
* EXTRA: [**History and meaning of React Hooks**](https://youtu.be/eX_L39UvZes)
* [REACT SUBSTRUCTURE EXPLANATION](https://youtu.be/s2skans2dP4)

REFS

* Allow us to work with other html dom elements to JS.
* React hook for refs -> useRef();
* There is also a ref value associated with the html element like we have key property associated with it.
* Updating the state value at each key stroke input is redundant, and we should only record it once the form is submitted. So we use refs in such case. It allows to set us a connection between the DOM elements and our javascript.
* How do refs work?
  + With refs we can set up a connection between JS and html element.
  + We use the useRef hook by calling it inside a functional component.
  + It returns a value which allows us to work with the HTML element. We connect refs to inputs in a form.
  + We need to add the ref prop to the HTML element and pass the name of the refhook.
  + 
  + It returns an object with a prop name current. It stores the actual DOM node of HTML.
  + It is recommended to not write/change DOM data, we can read from it.

**Controlled vs Uncontrolled components**

* Uncontrolled components – when we access values with a ref. Because their internal state reflected in the ref is not controlled by react, where the user decides the entered value and we just fetch it with ref, and we do not reset the input form to blank with the formal react method, instead we do the dom manipulation.
* Controlled components – when we use the react hooks to manipulate our input values.