**React** is all about “components” because all user interface in the end are made up of components. With React, you **define the target UI state(s)** – not the steps to get there! Instead, React will figure out & perform necessary steps. (Declarative programming)

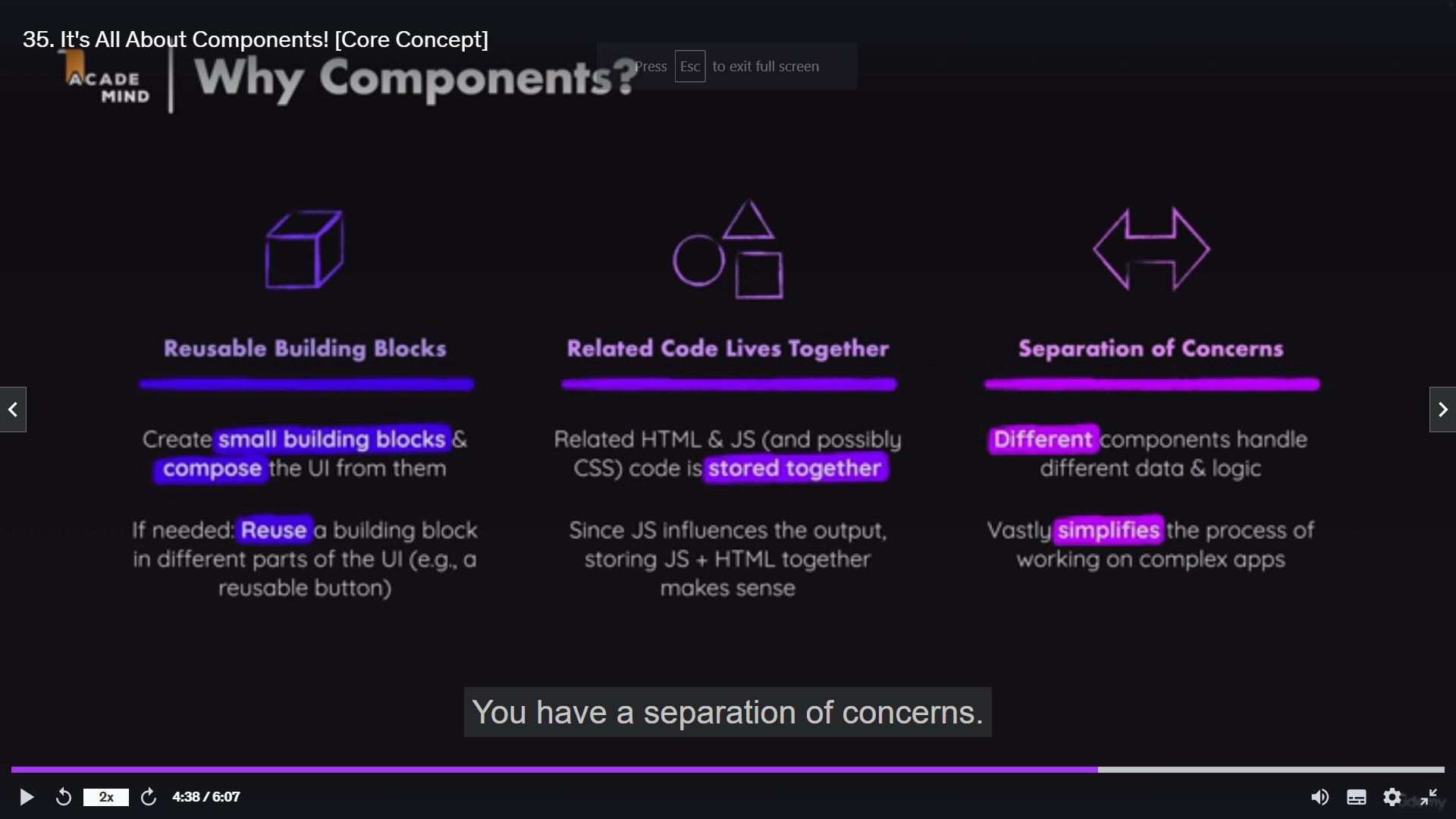
React project uses a build process:

* Raw, unprocessed React code **won’t execute** in the browser because React uses JSX feature.
* In addition, the code would **not be optimized for production** (e.g. not minified)

**Components** are reusable building blocks in your user interface (just a combination of HTML code, CSS styling, JS logic).

Ex:

Reason why we use components:



Graphical user interface, application

Description automatically generatedHow components are built:

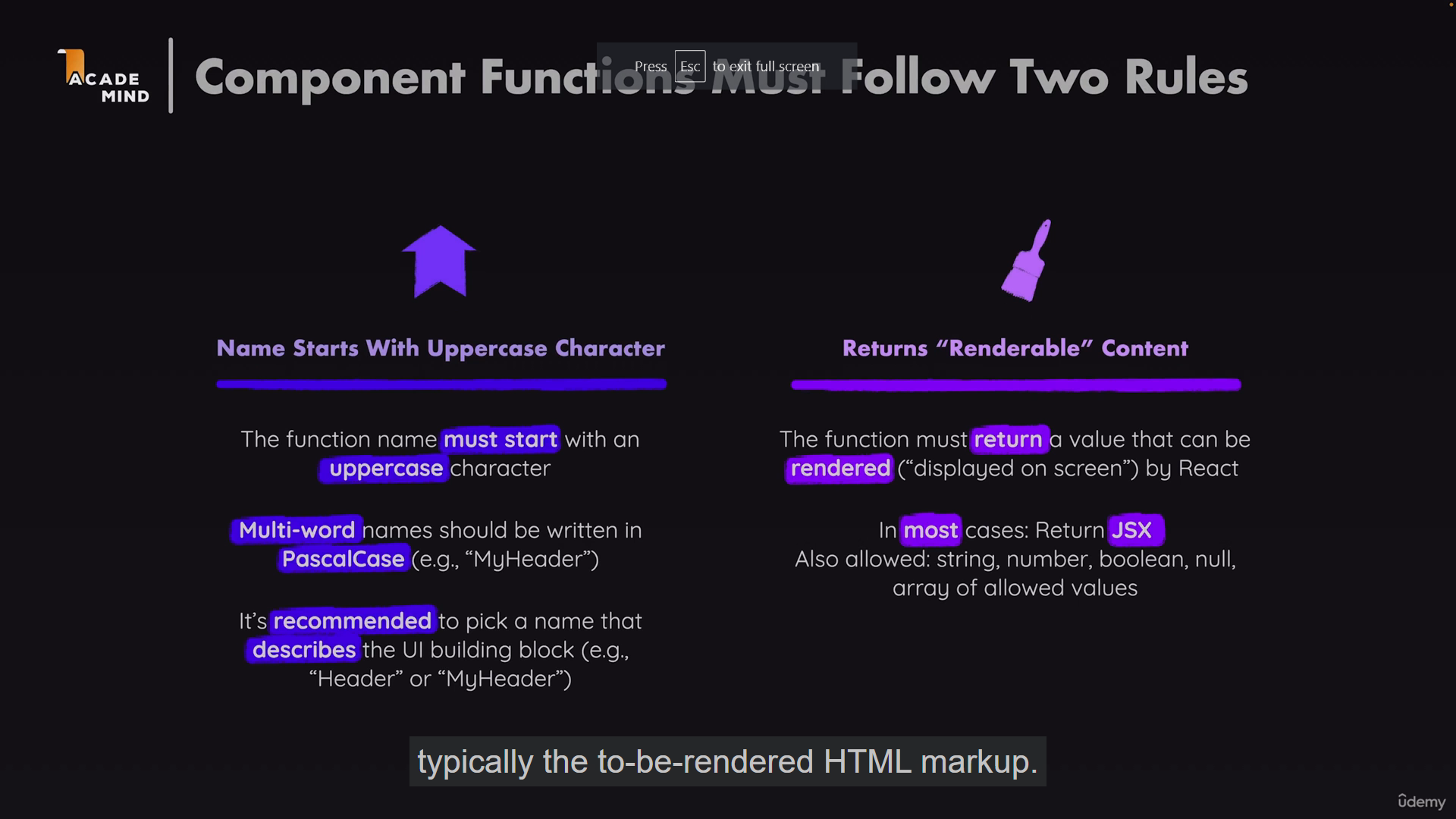
**React code** is written in a “declarative way”. With React, we will not tell that a certain HTML element should be created and inserted into a specific place. We will always define the desired end state, and React’s job to figure out which elements on actual webpage might be added or removed or updated.

Graphical user interface, website

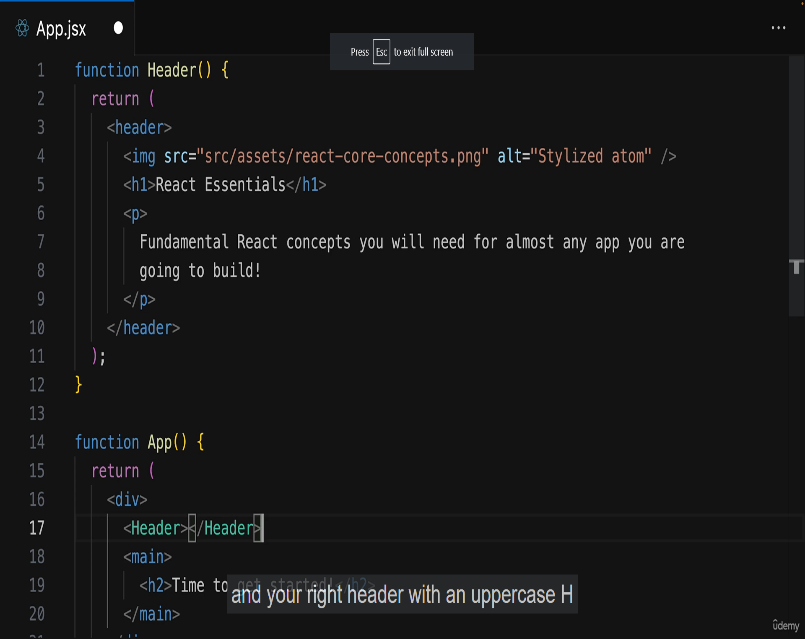
Description automatically generated

**JSX (JavaScript XML)** is basically HTML code inside of JavaScript.

Component function rules:

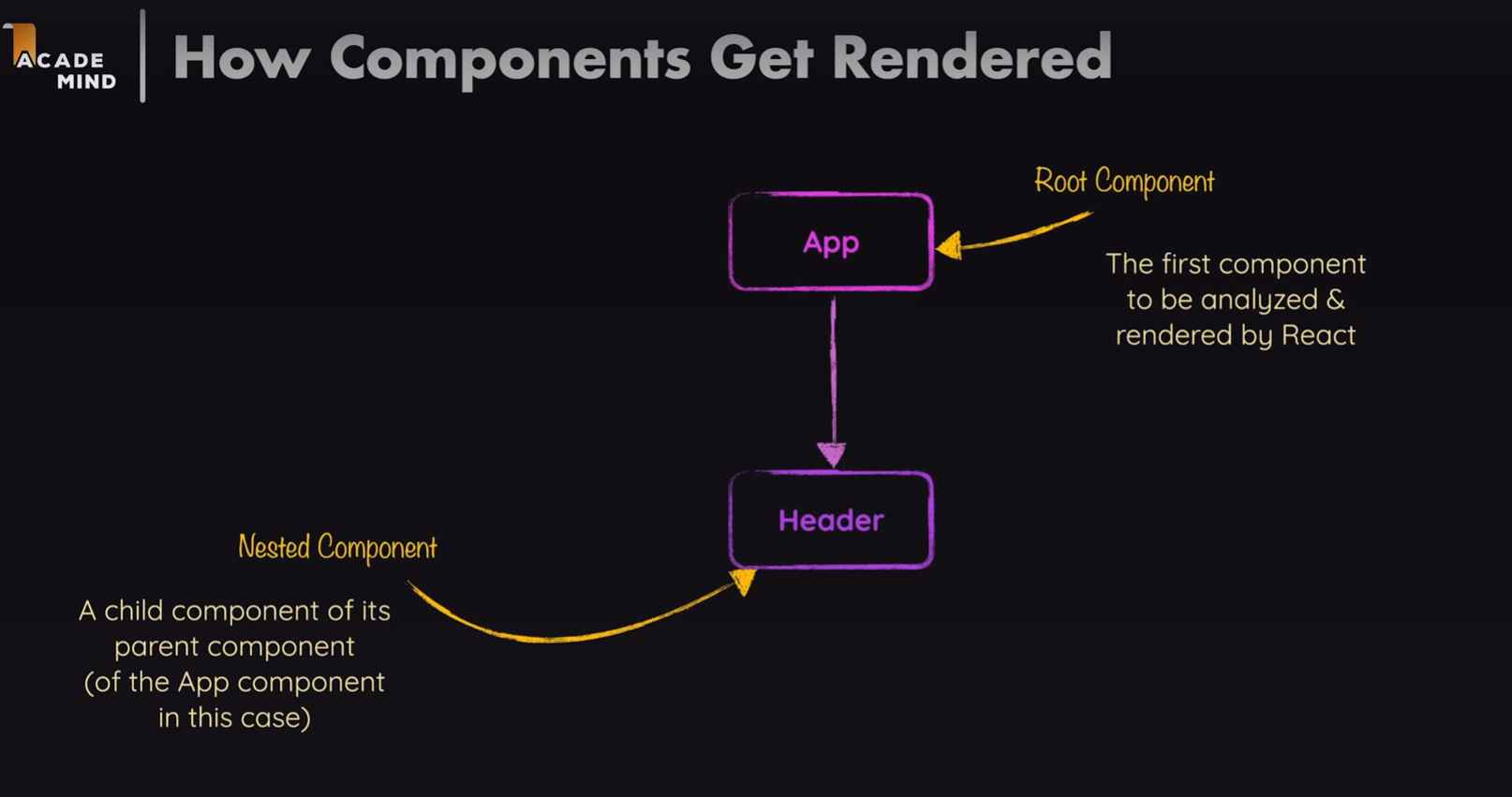


With React, we basically create component by building up the component tree

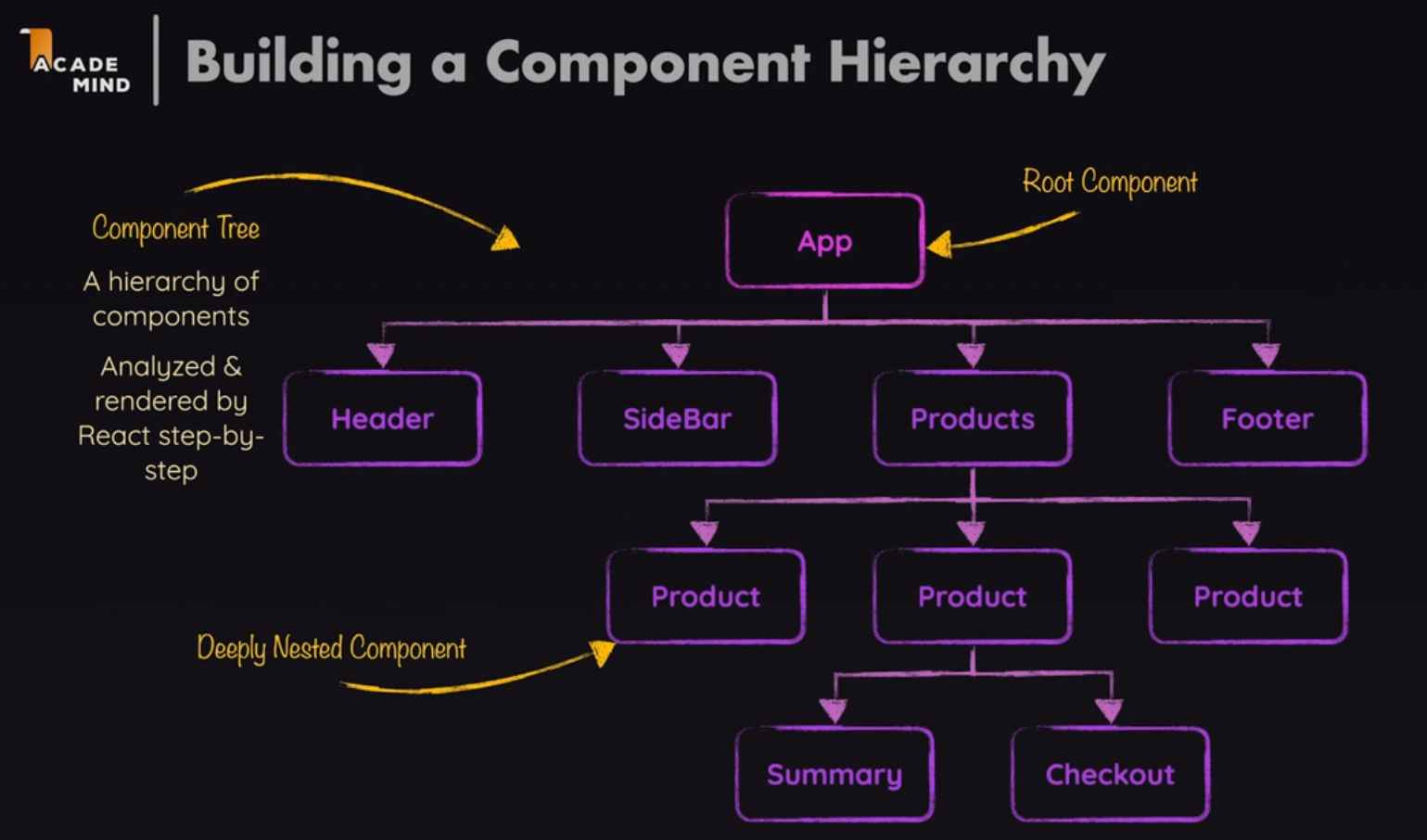
Graphical user interface, diagram

Description automatically generated

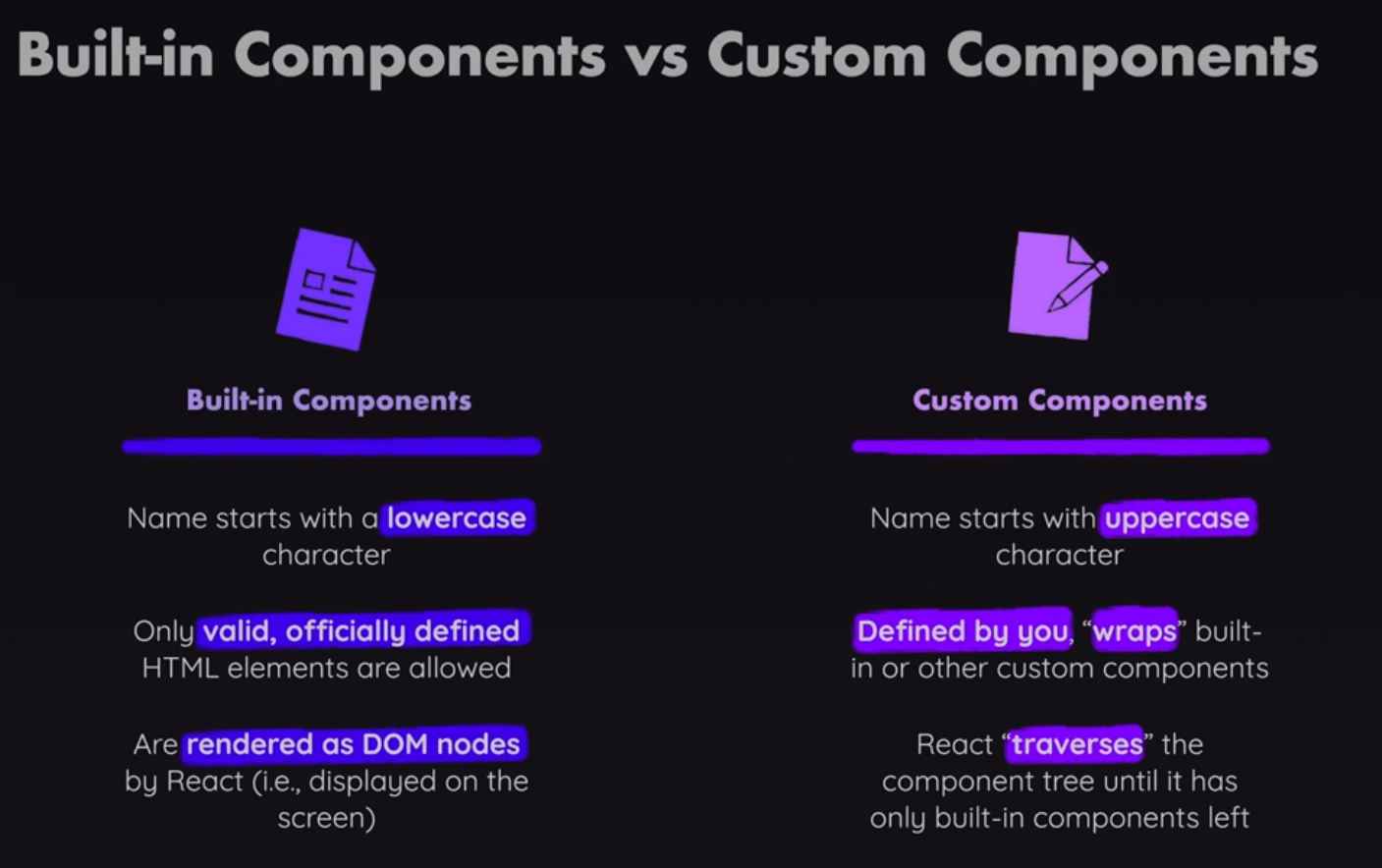
**createRoot & render function:** responsible for rendering a single root component which then in turn may contain as many nested components as needed.



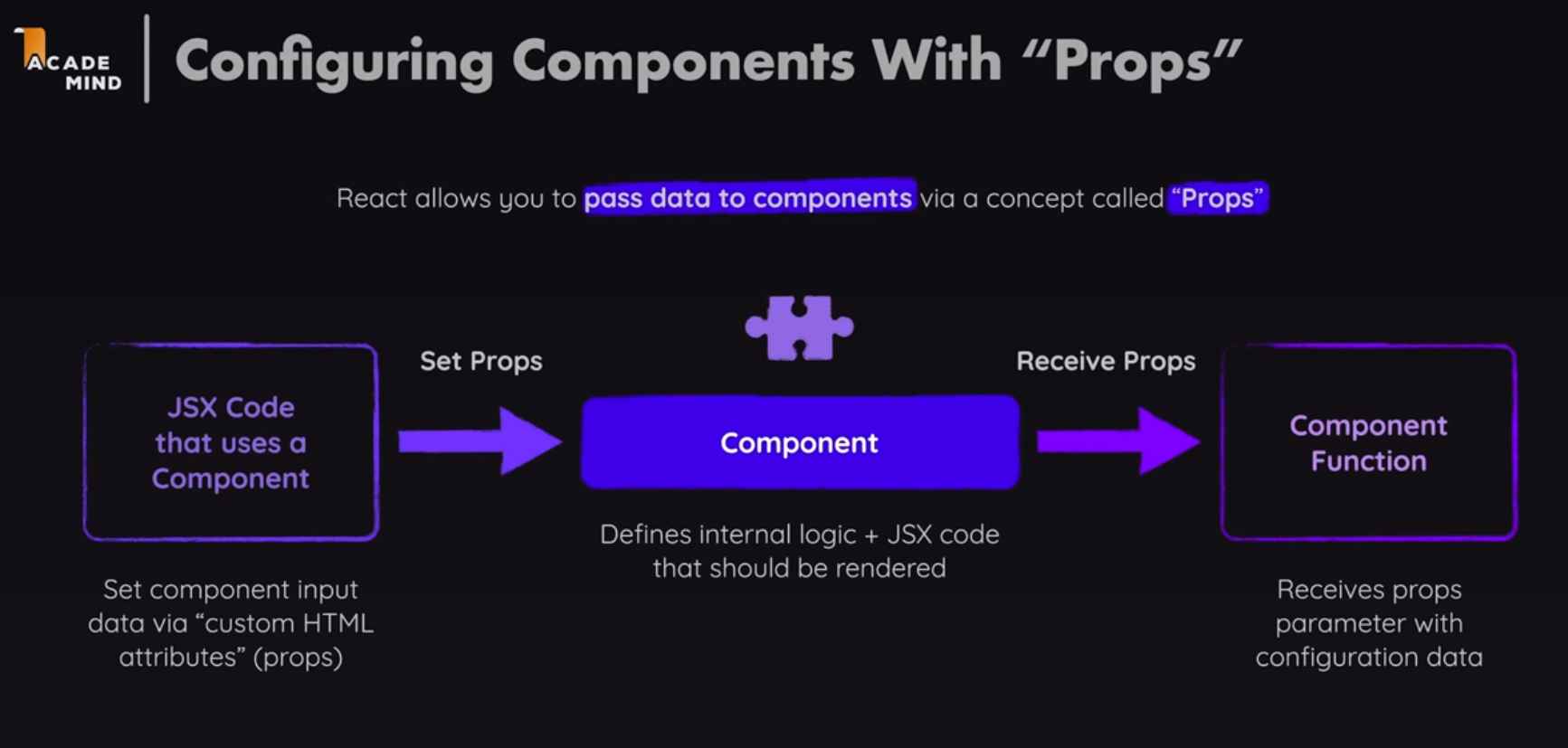
Ultimately, we end up with a component hierarchy



Built in components like header, image, div are rendered as DOM nodes in React. On the other hand, custom components are just functions and executed as functions by React.

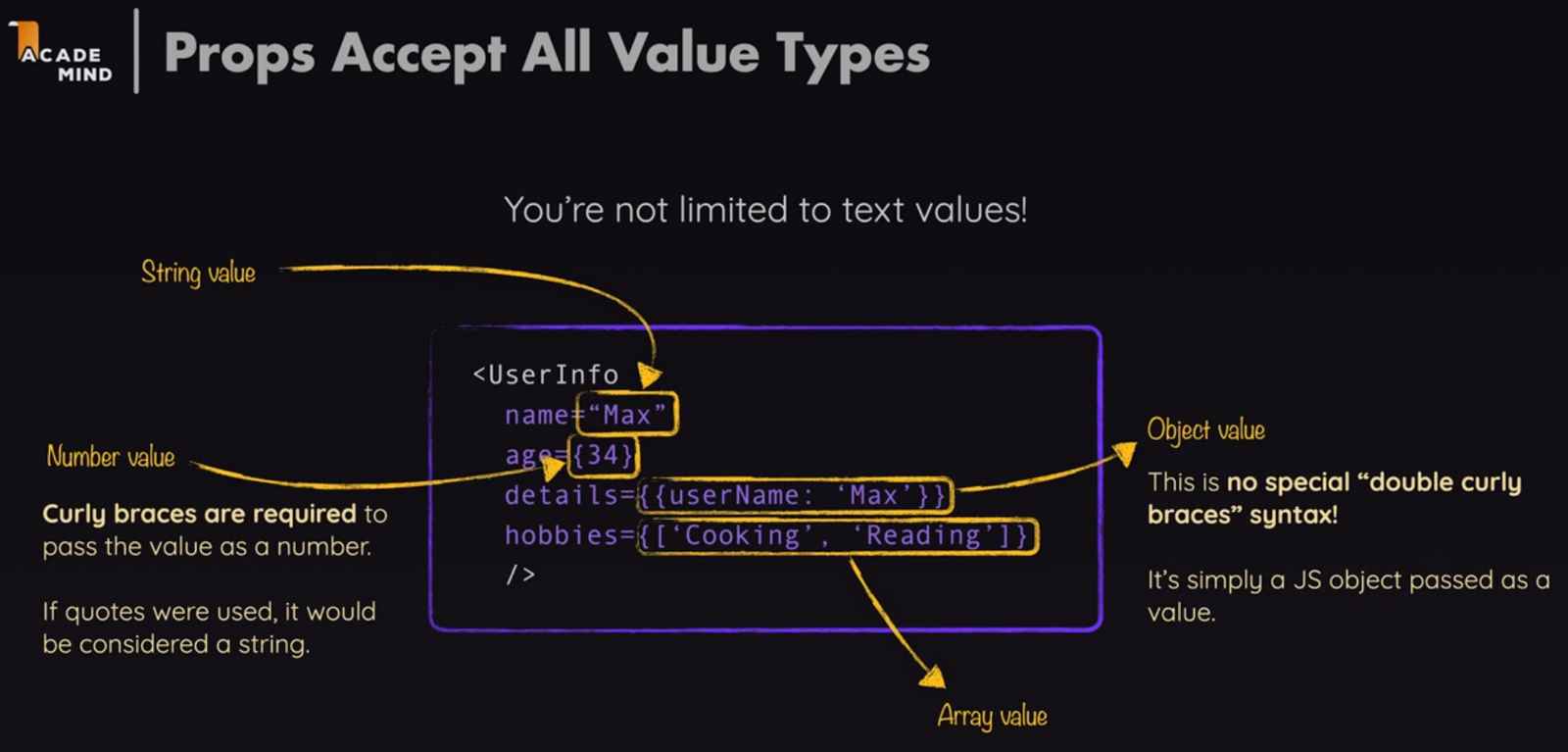


**Props in React:** let’s say we have custom component which should output a list of items where the goalItem should be output dynamically. We can pass data to custom component by adding a attribute. And inside that component, we can then get access to all these attributes.





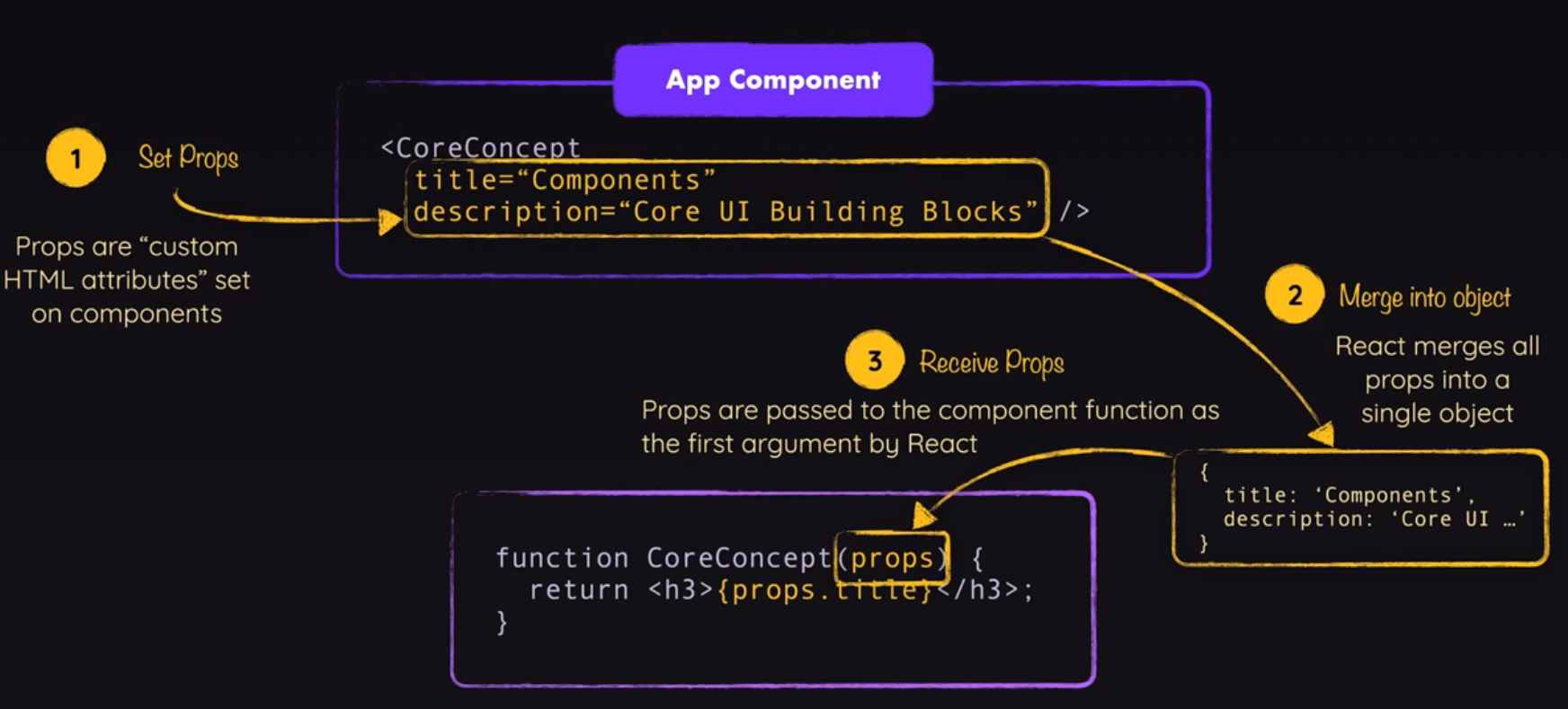
You can pass all kinds of values for your props



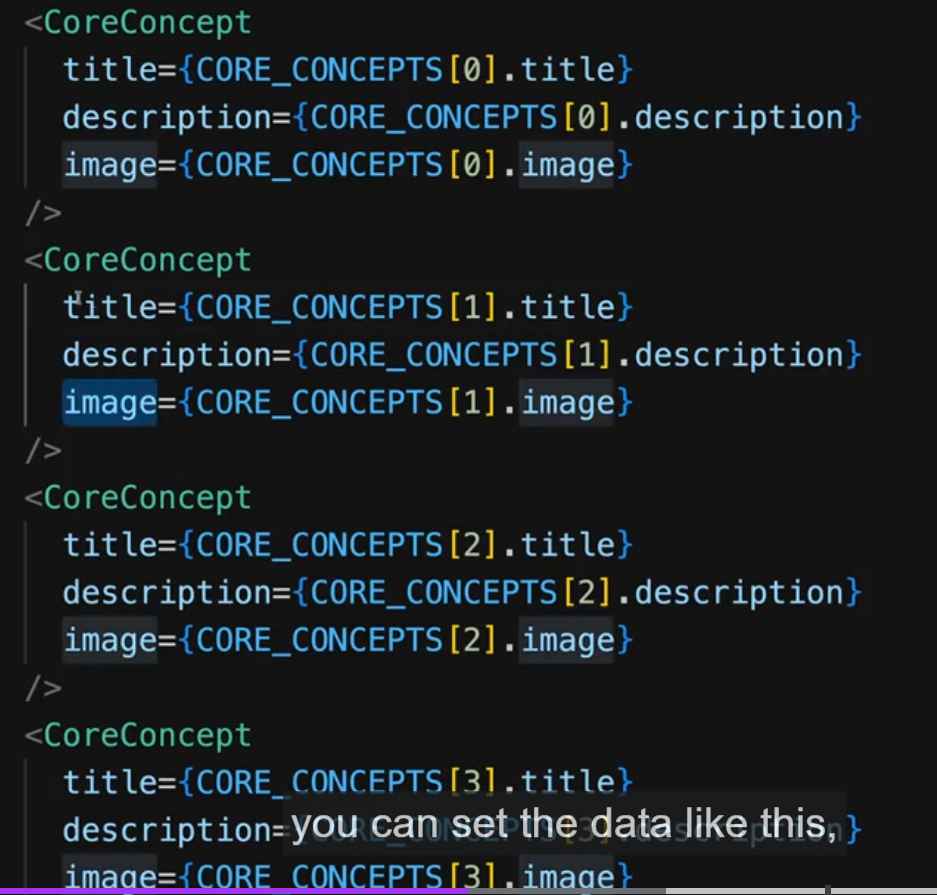
However, when pass parameter to the function, it only allows 1 parameter as an object



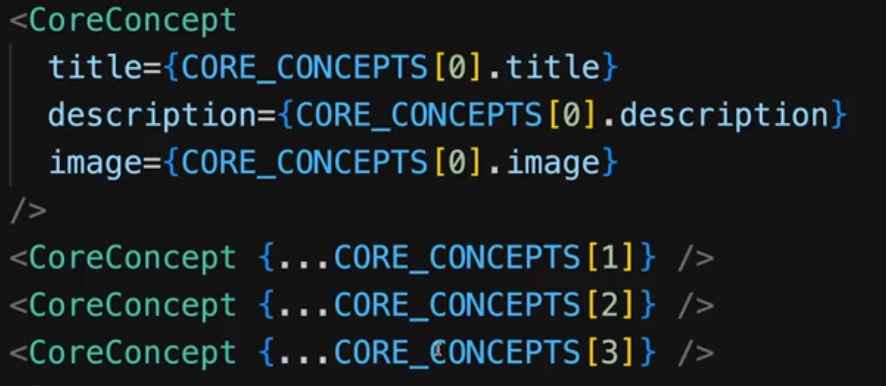
That “props” parameter is an object.



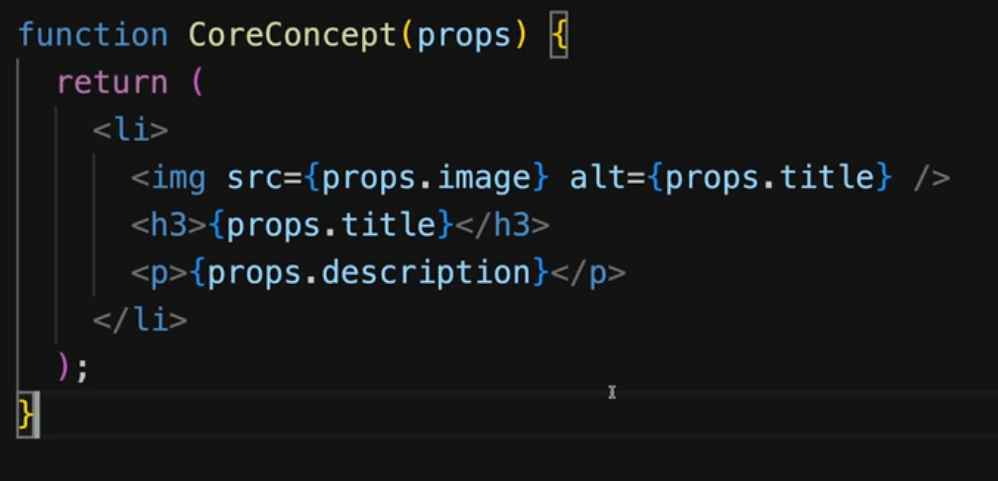
Example of how props are used with CORE\_CONCEPTS imported externally



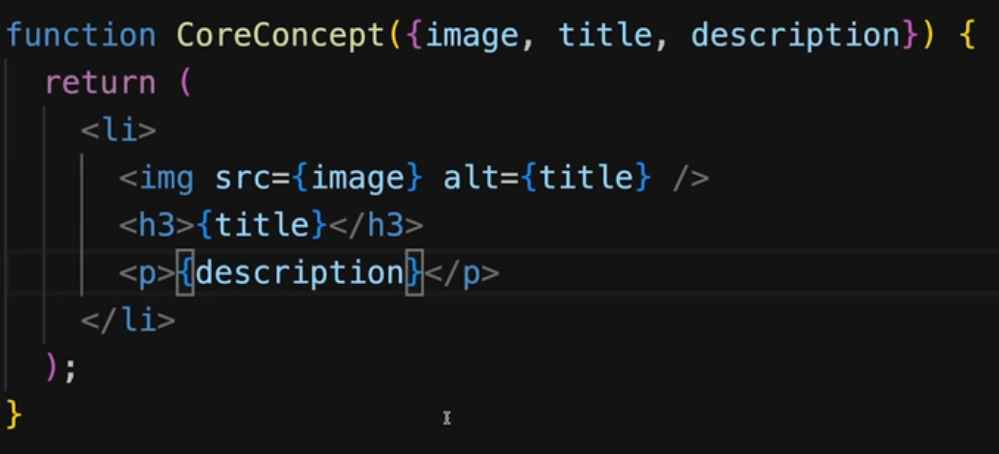
However, it could also be written like this



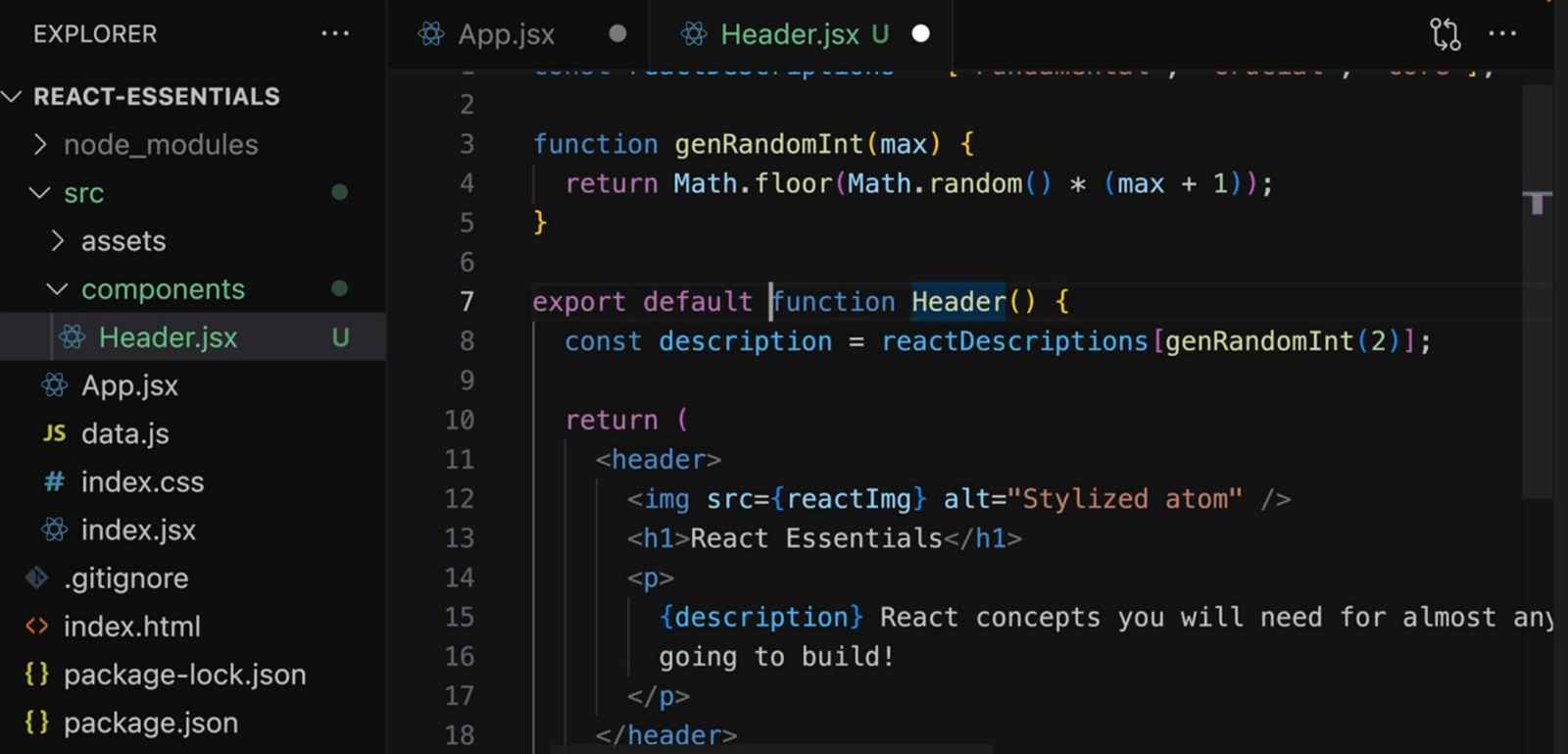
Also, instead of defining “props” parameter, we could also deconstruct the “props” object



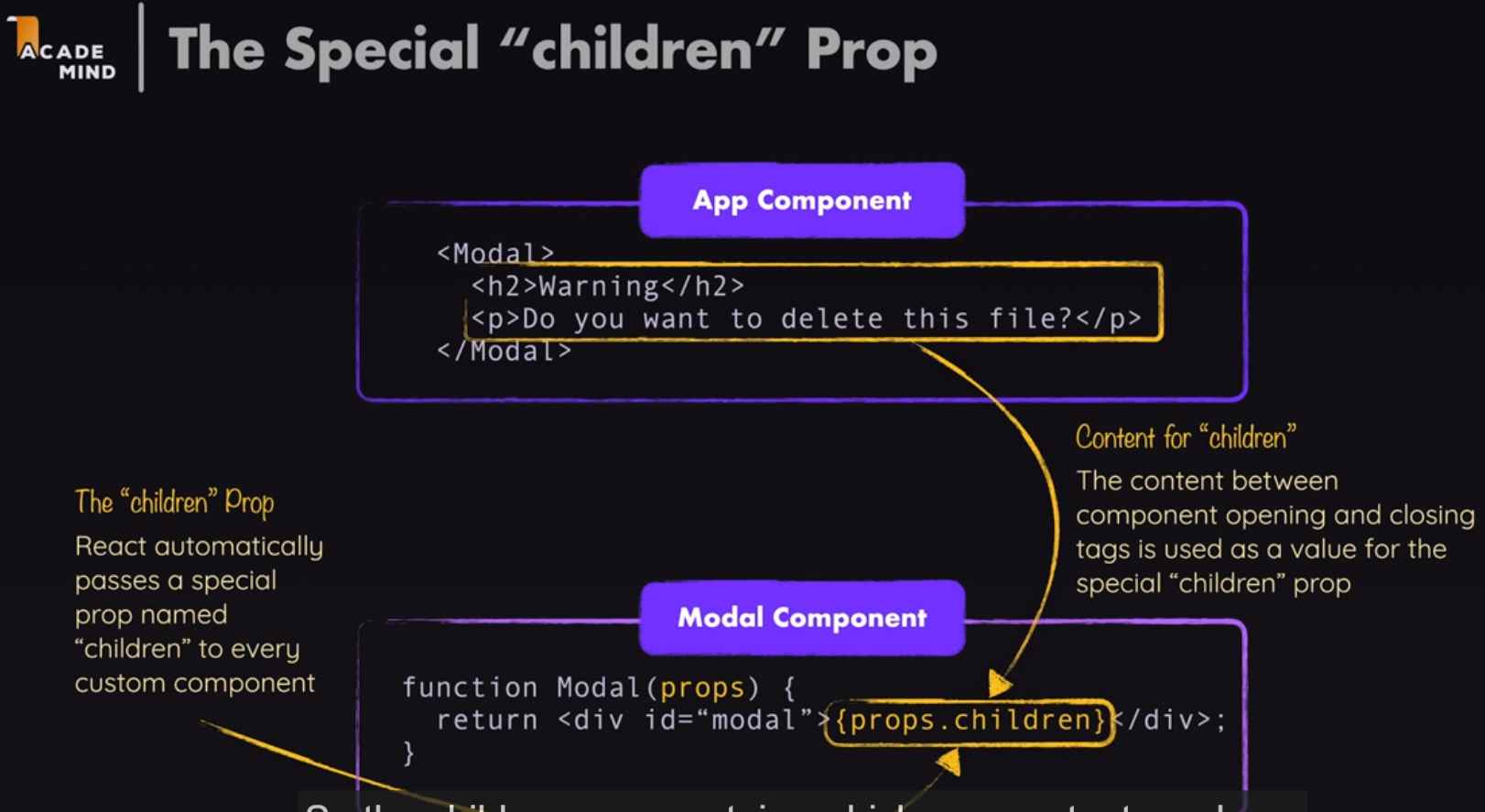
Turn it into this

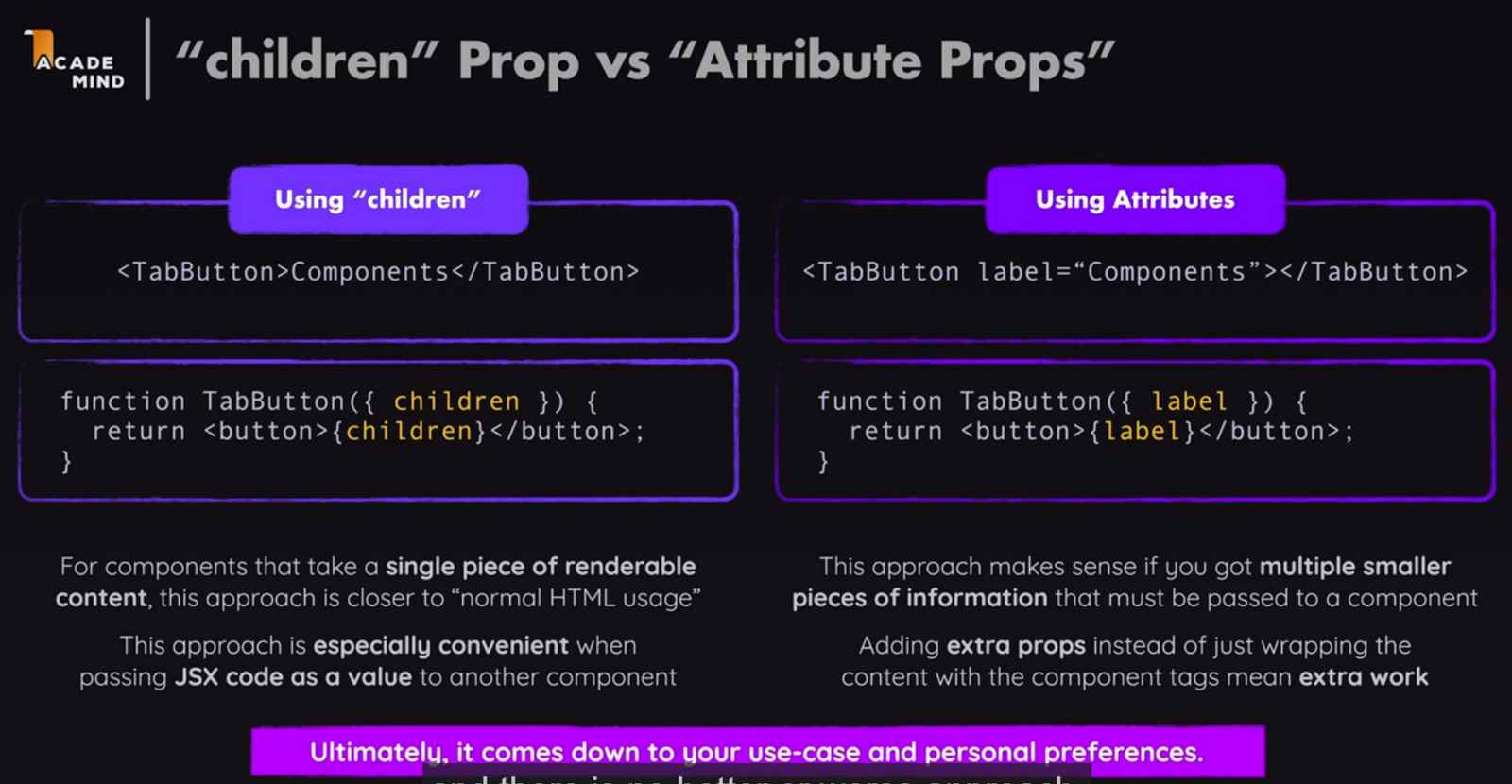


Best practice is to minimize the size of components and move to the same file name JSX with the component name.



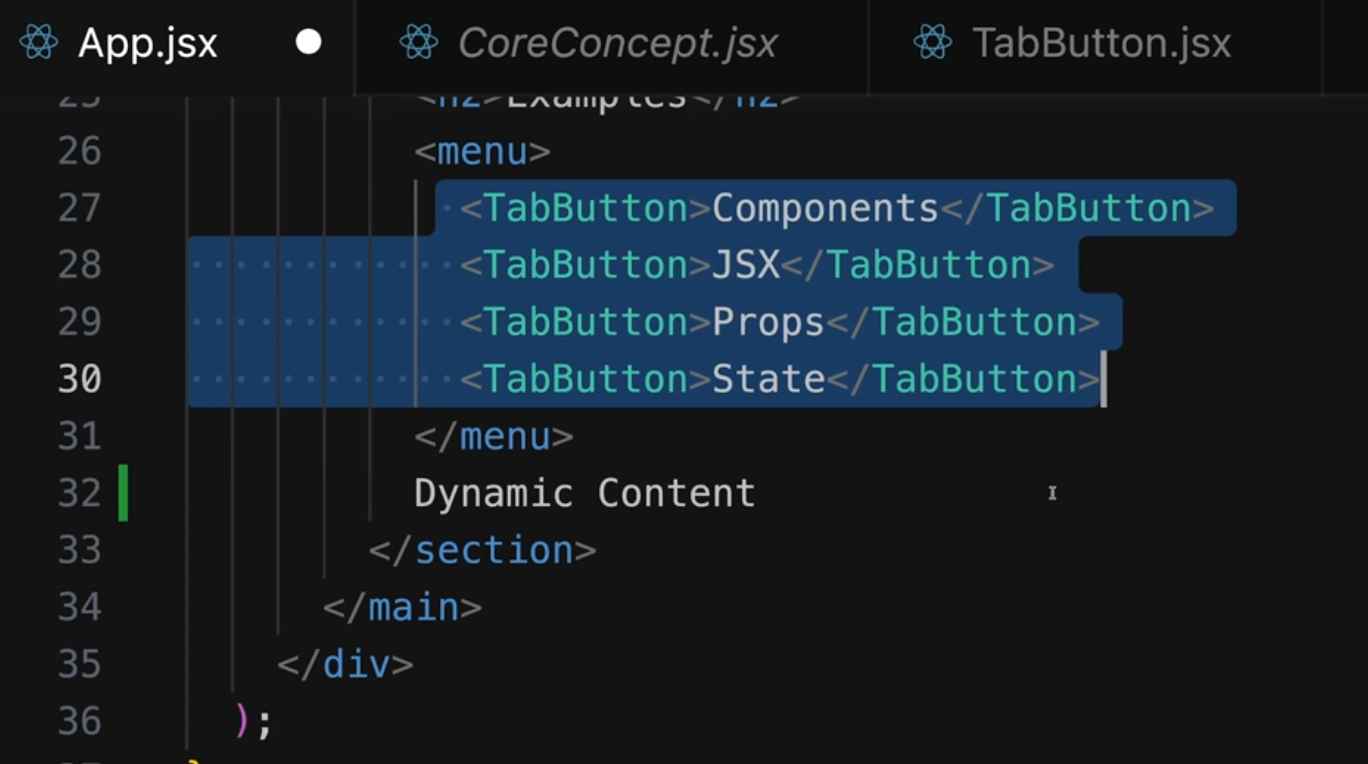
The children props contain whichever content you have between your component text, it doesn’t appear with the help of “children” attribute (we don’t need to have something like <Modal children =””> for it to appear in the function), it could be a text or any complicated elements.





In order to create event in React, we use special attribute, for example: “onClick” and the value would be a function. The function must NOT be executed or must not add paranthesis (not handleClick() but only handleClick). We are using function as value.



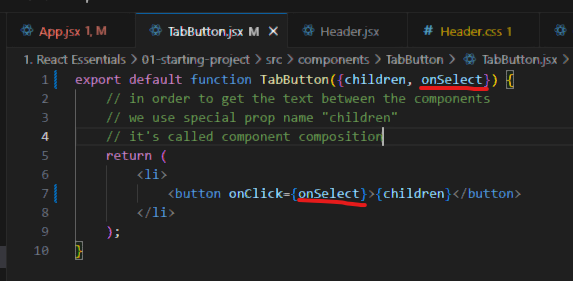


Here we want to output the dynamic content based on which button is pressed, therefore in order to set and update the Dynamic Content, we need to listen to clicks on our custom button.

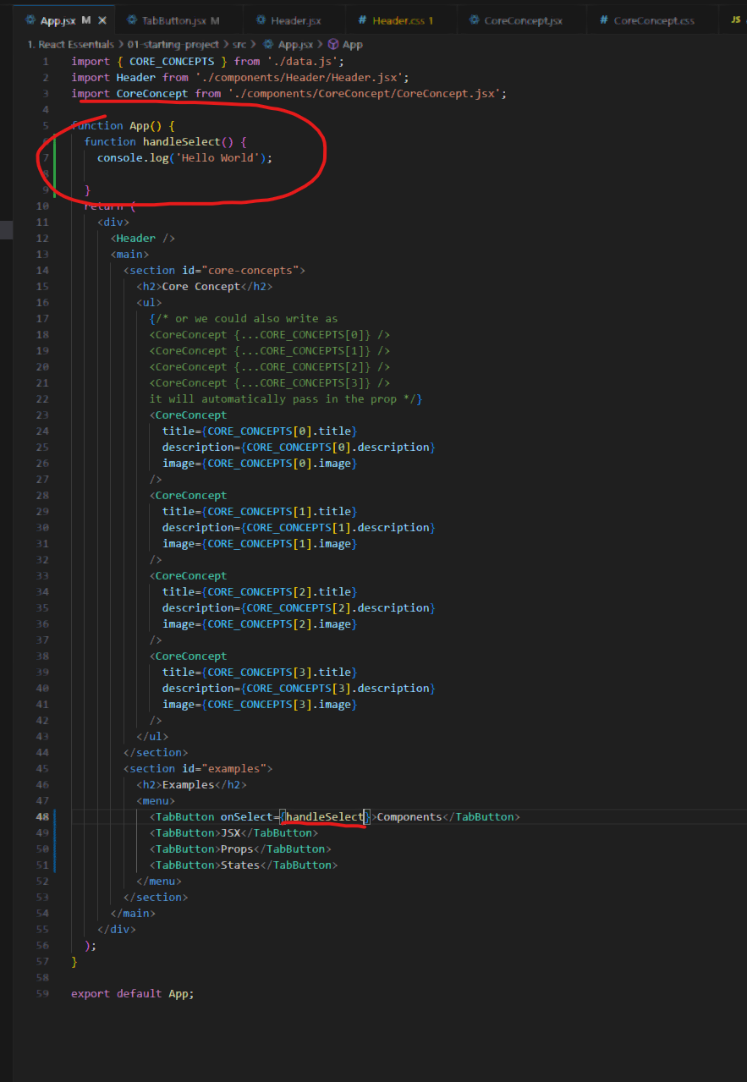
So it would be nice if we have the onClick attribute on those custom button as well and it would then be forwarded to the built-in element, which does have the real “onClick” prop that’s built-in and where React will make sure the function you provide as a value will be triggered.



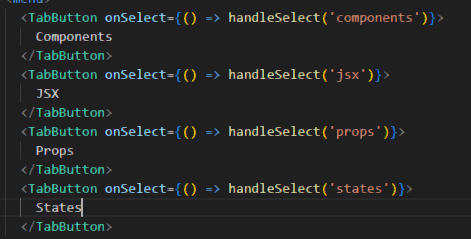
So to achieve it, we can simply add a prop name “onSelect” and pass that function as a value to the “onClick” built-in attribute:



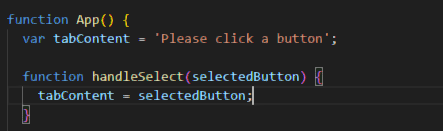
Then after that, we go to the place where the custom TabButton component then set the onSelect prop and add the handleSelect function.



So now we want to pass a parameter to the handleSelect so that when a button is pressed, it will show only the dynamic content for that button, so instead of using a value for onSelect, we pass an arrow function to the onSelect



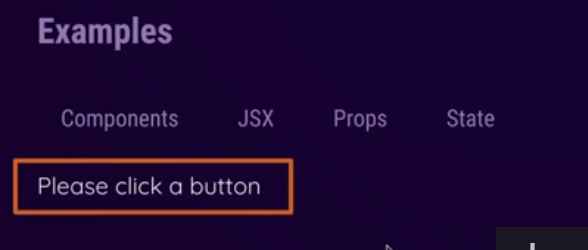
With that being said, let’s create a variable and update that variable value whenever a button is pressed:



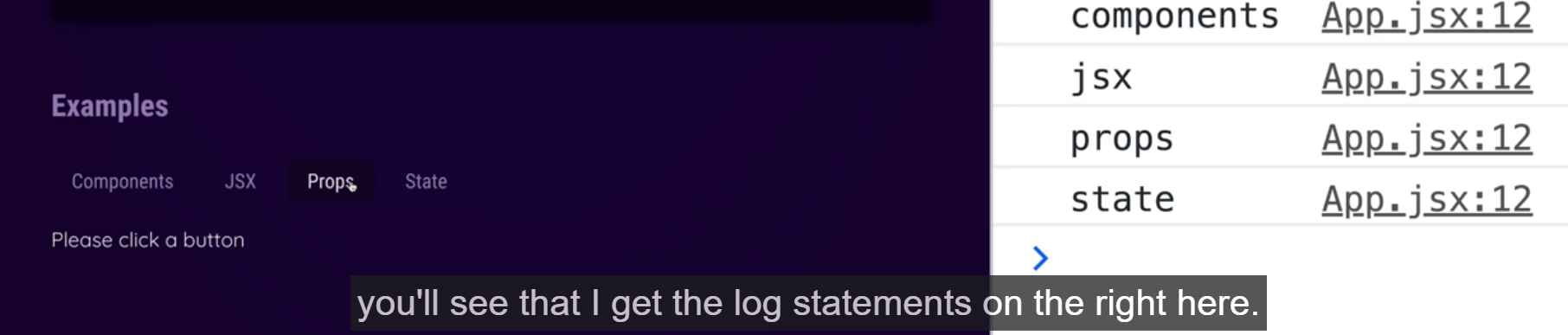
Then we try loading that variable down here



Then we do see “Please click a button” here:



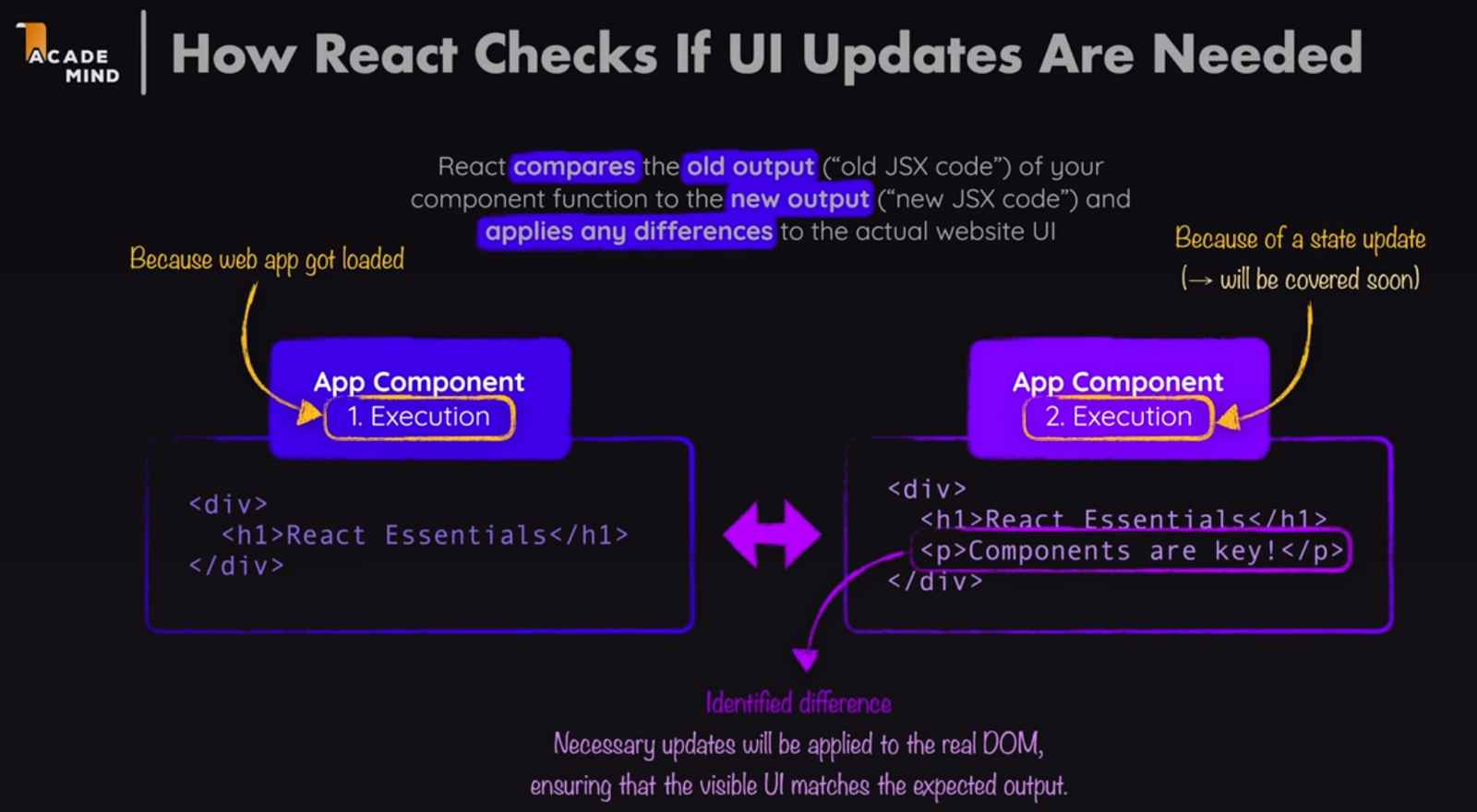
However, as we notice when we press those buttons, the content is not updated.



So the function is being executed and we update the variable but the UI is not updating. It is because the App component function is not executed again

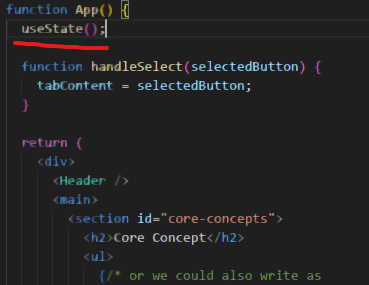


Since the App component is not executing again, the entire JSX code is not getting reevaluated, so it still only knows the initial tabContent value.

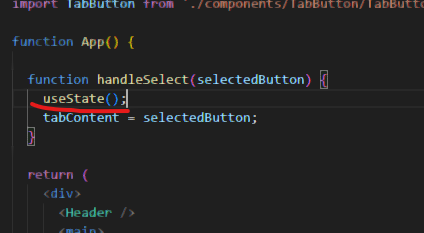


React in the end will take a look at the JSX code and it compares it to the currently rendered UI and it detects any differences, it will update the UI accordingly. Therefore, we need another React thing, called state. All function starts with ‘use’ in React are ‘hook’.

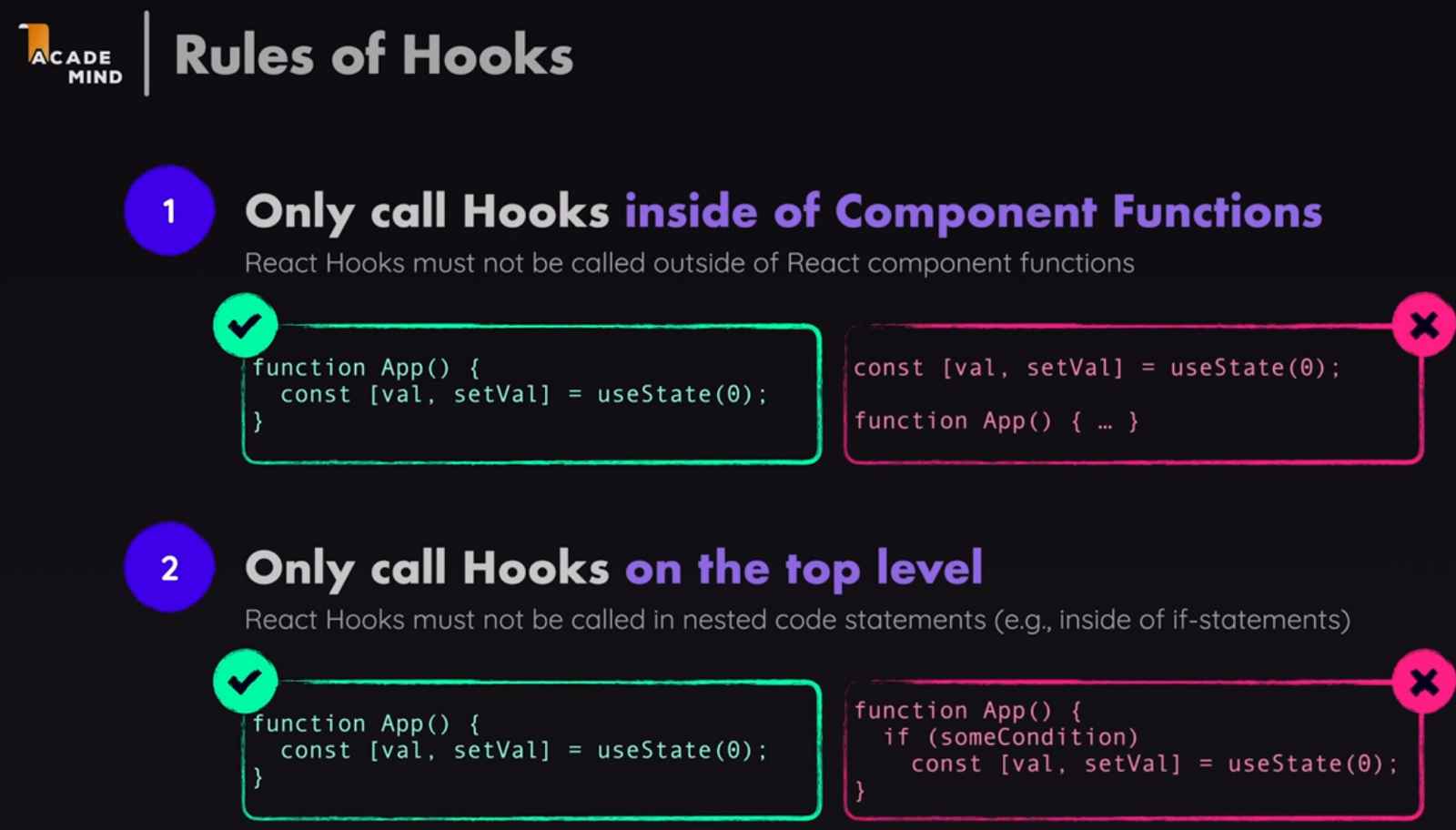


React hooks are technically regular functions but they must only be called inside of React component functions or inside of other React hooks. And we should call it like this, directly inside of the component function

Not nested inside of some other code, for example, not nested inside of the inner function, like this



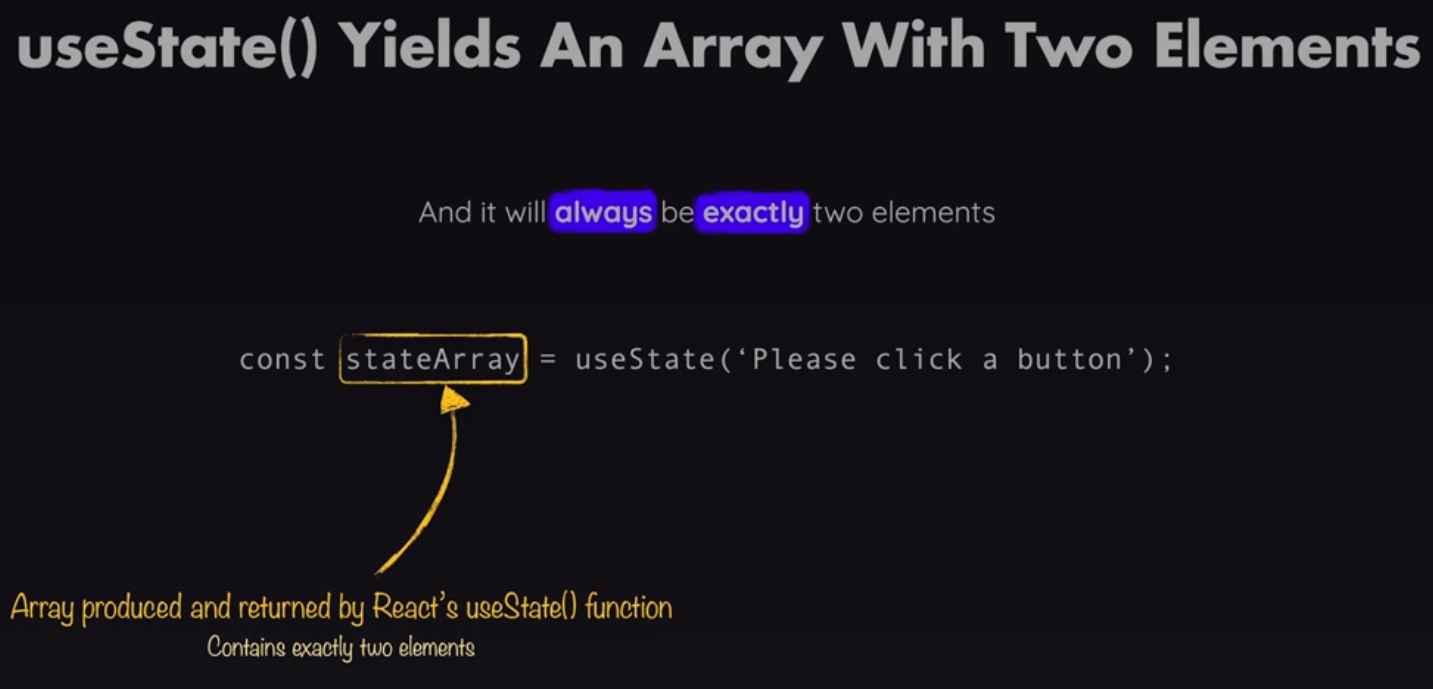
Important rules of hooks:

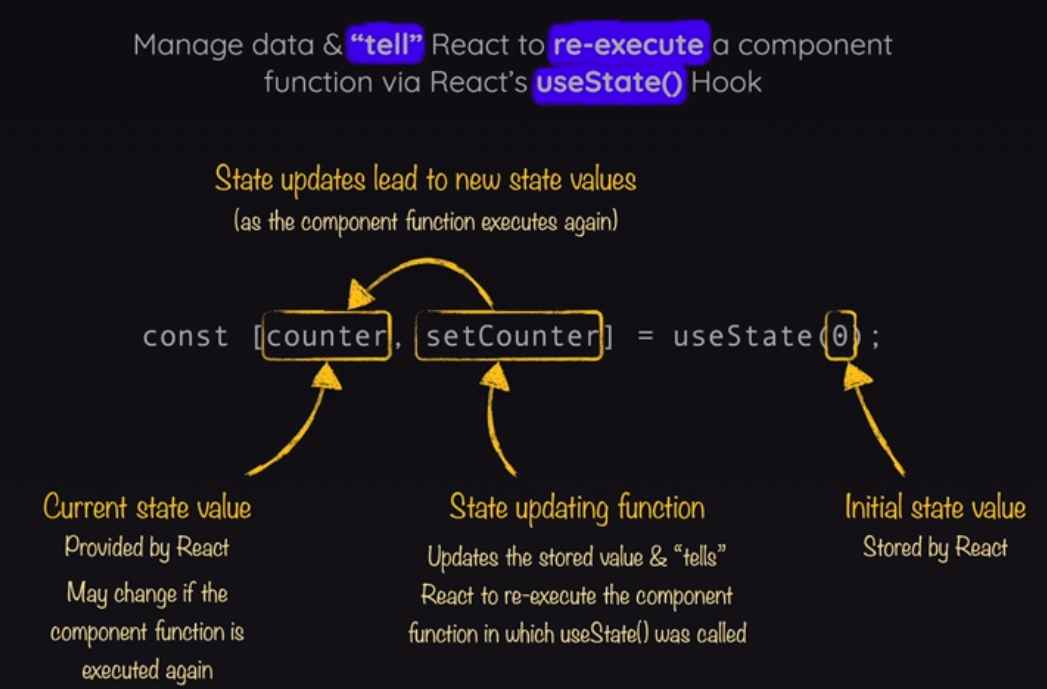


useState() does accept an argument which is basically the default value you want React to store when the component first rendered

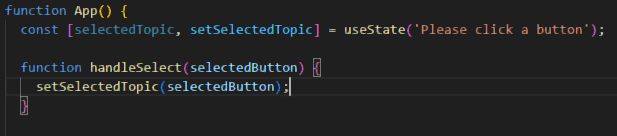


And then useState actually return a value, and that value we get back is actually an array

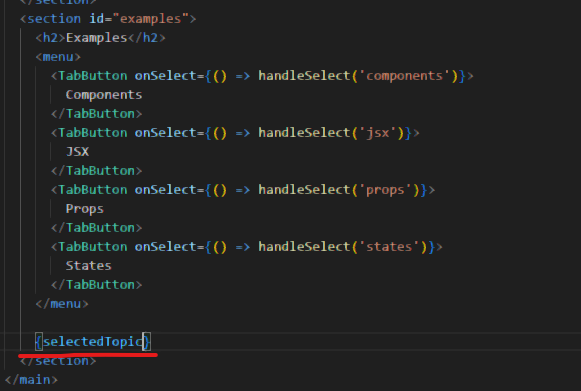




And therefore, now here in handleSelect, we can call setSelectedTopic here to be our selectedButton value.

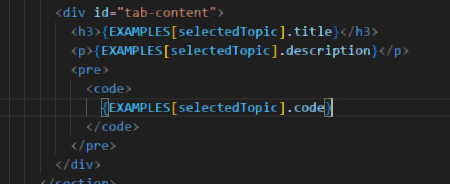


And down here, we can output selectedTopic

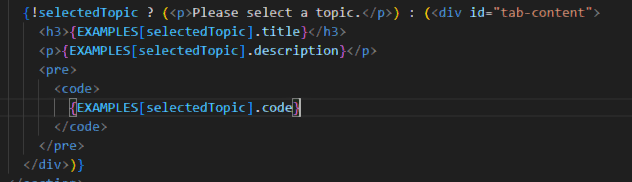


Also, when you call setSelectedTopic, React in the end schedules, this state update and then it re-executes this component function. So therefore, the updated value will only be available after this app component function executed again. Only then, the new value is available.

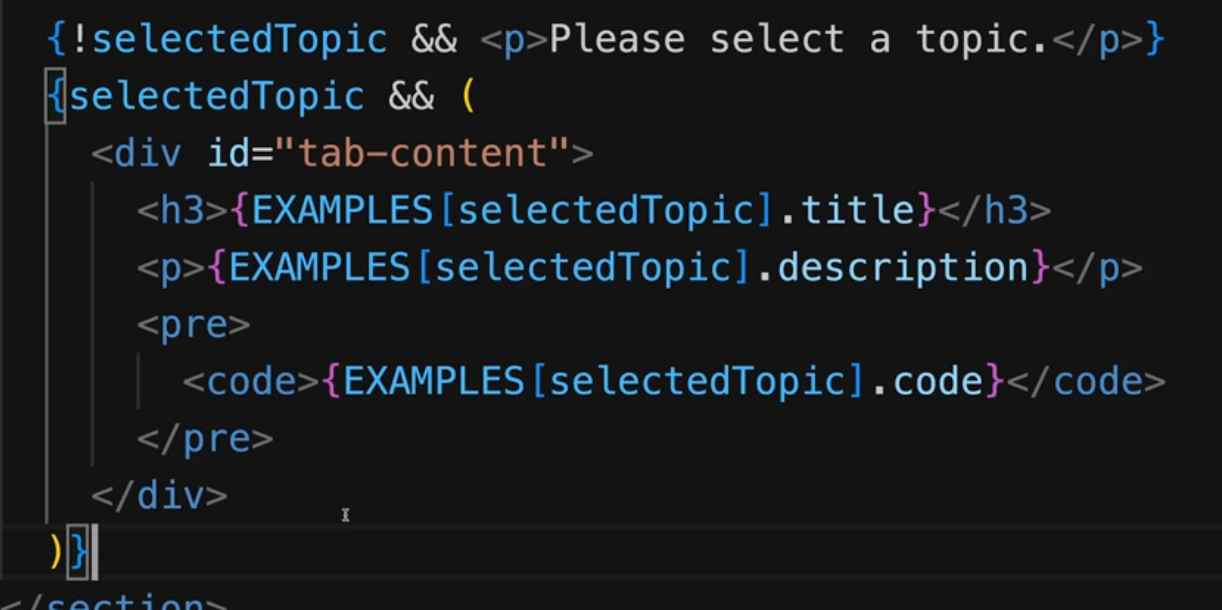
With that being said, we could use the function to access data like this:



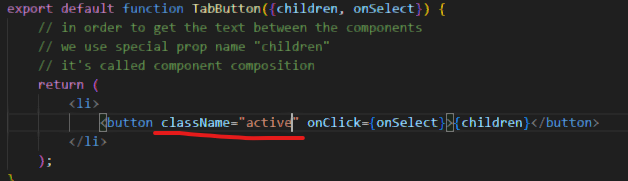
So now if we only want to show an initial info (not really showing anything selected, we could use conditional content like this):



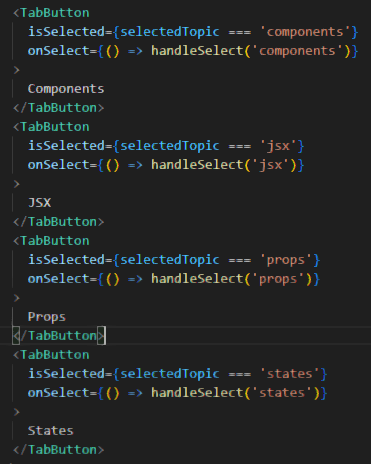
Alternatively, this also works



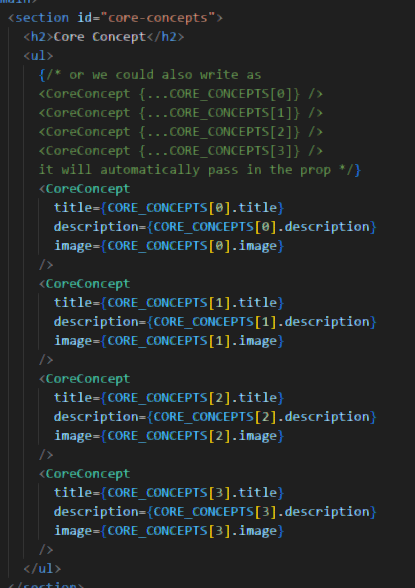
Add className attribute to add a class to an element in JSX



So to determine if the button is selected or not, we add a isSelected prop and compare if the selectedTopic is actually the one



This approach has the disadvantage of potentially breaking if the data source gets changed.



So it would be better if the number of the CORE\_CONCEPTS component would be derived dynamically based on the number of items from the data source, so it could simply output one core concept component per entry in that array automatically.

So we could do:

