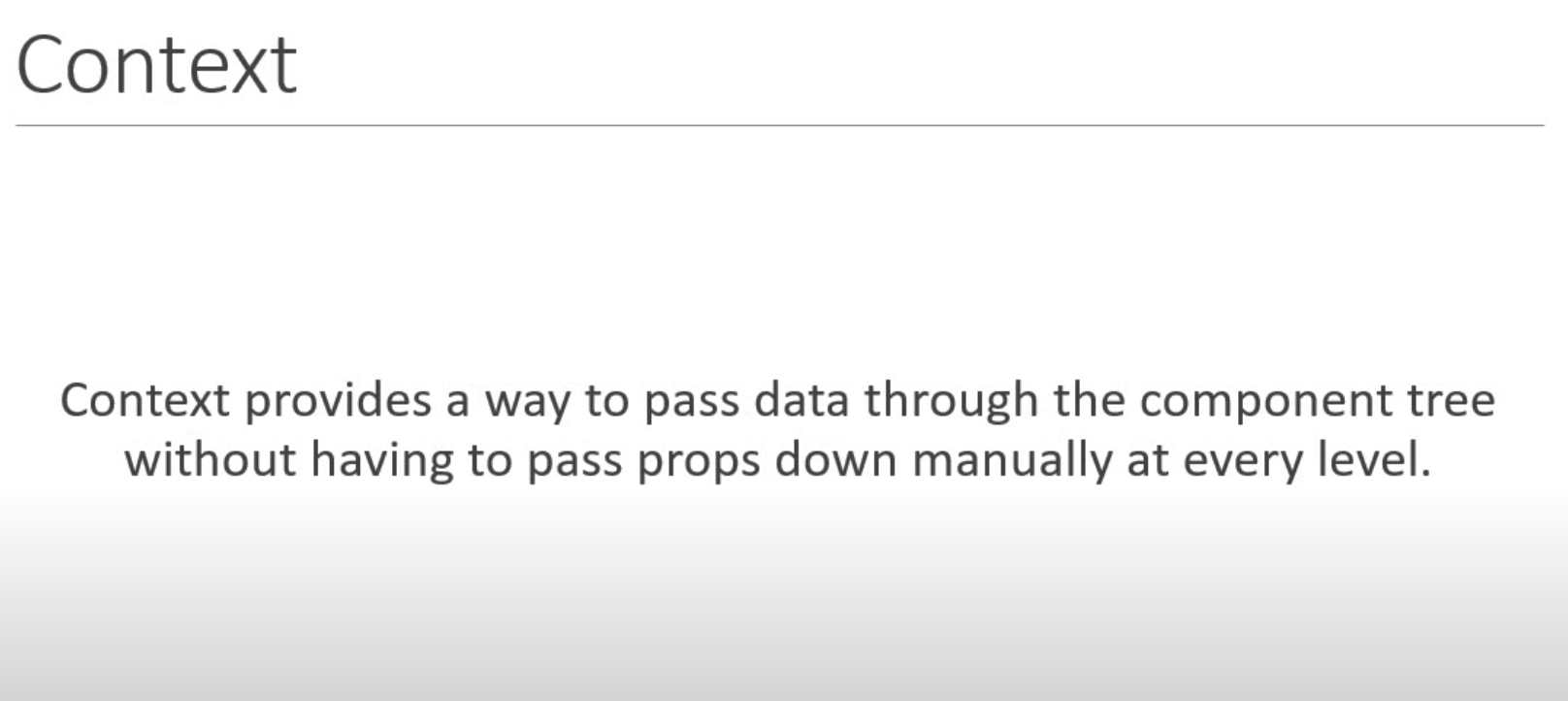
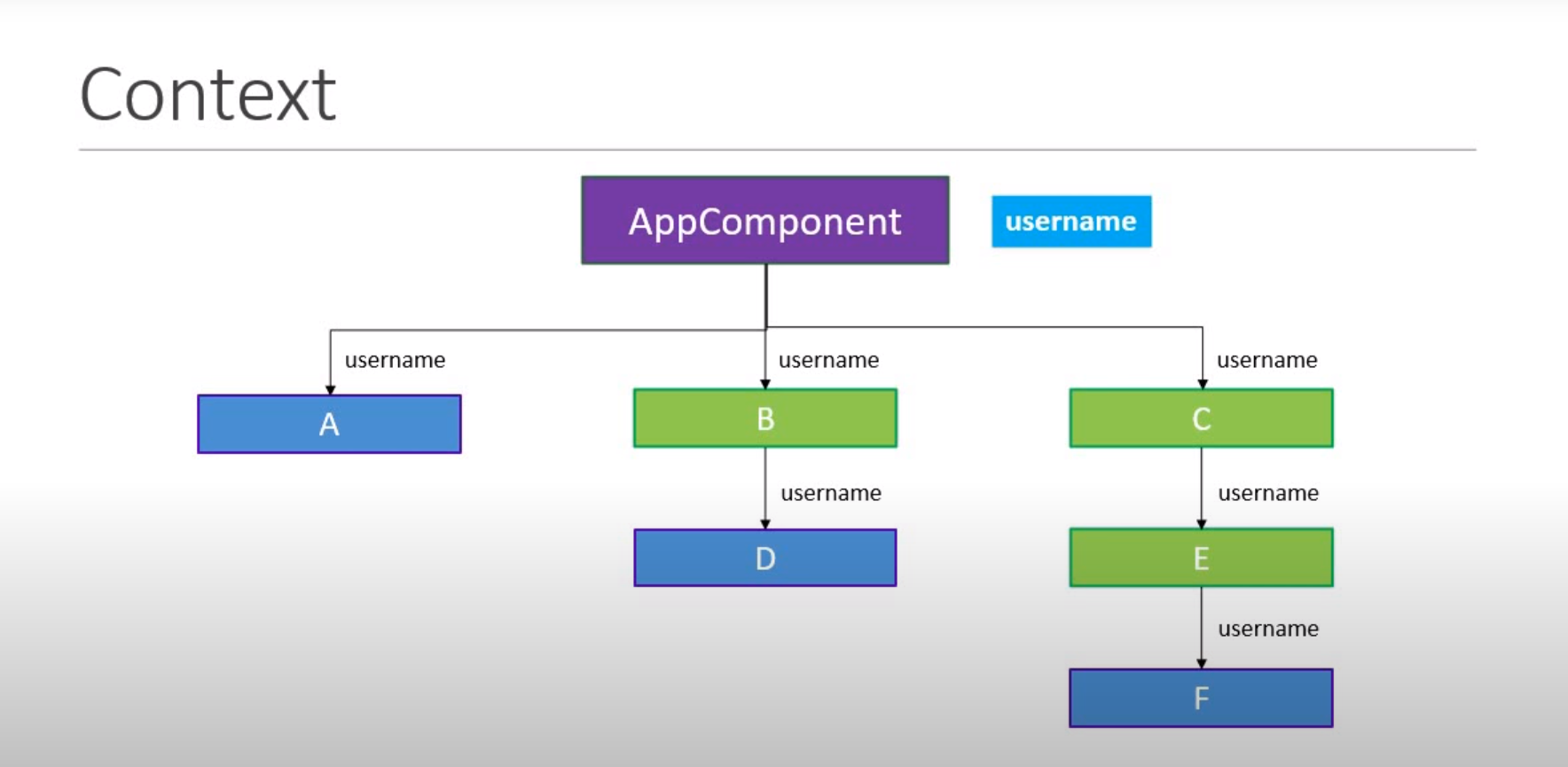
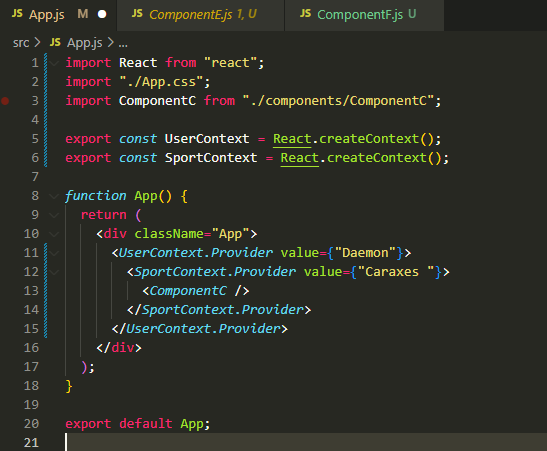
**useContext Hook**



In this example, we want to pass *username* from *AppComponent* to Component *F* without needing to pass it through Component *C* & Component *E.*



*App.js:*



1) We create a context using the *createcontext()* method.

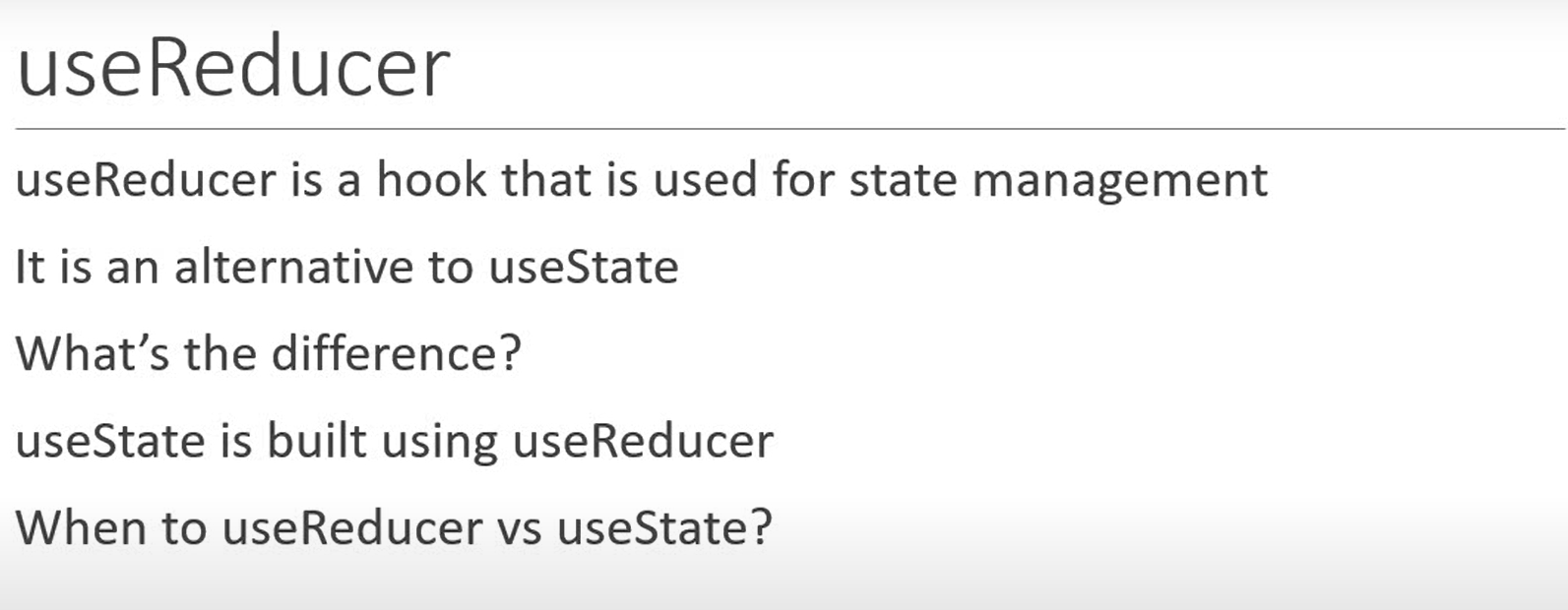
2) We provide the context value at a high level (*<ComponentC/>*) in the component tree.

3) We consume the context value using the render props pattern.

the *useContext()* hook only makes the consumption of the context value simpler.

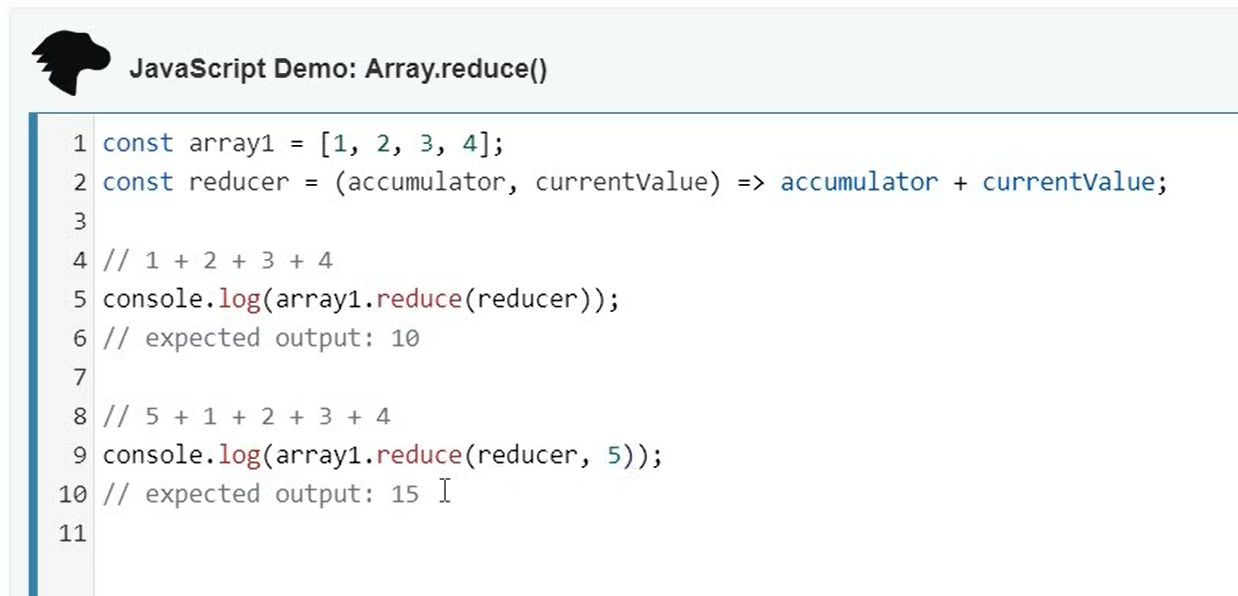


**useReducer Hook**

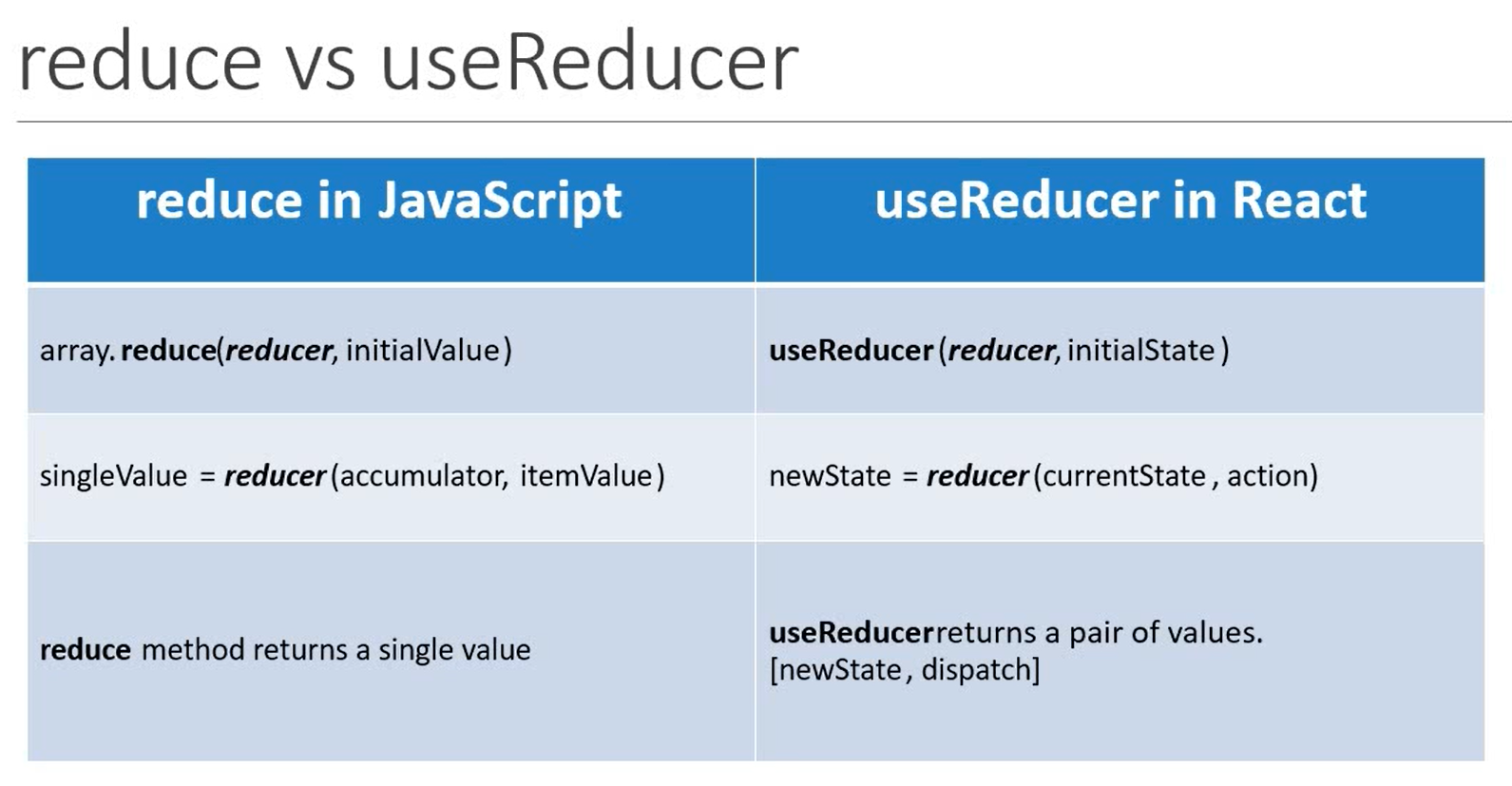


*useReducer* hook is similar to the *reduce()* method in JavaScript.

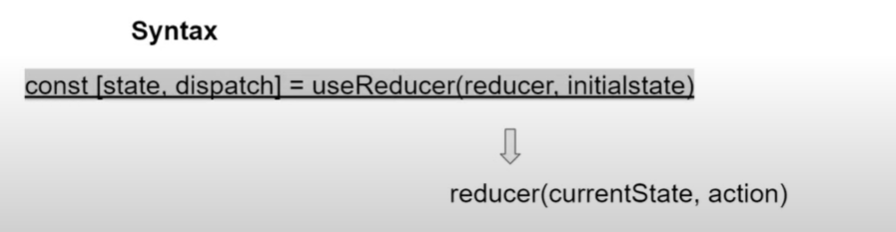
The *reduce()* method has two forms.



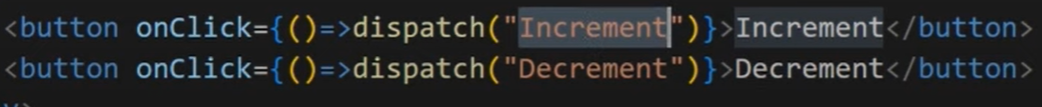
*reduce()* vs *useReducer* hook:



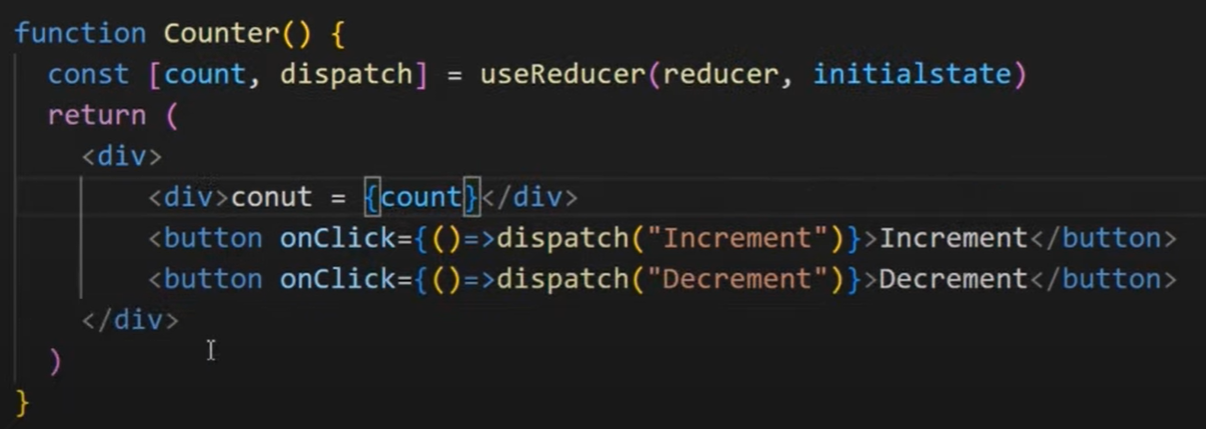
The syntax for *useReducer* is:

**

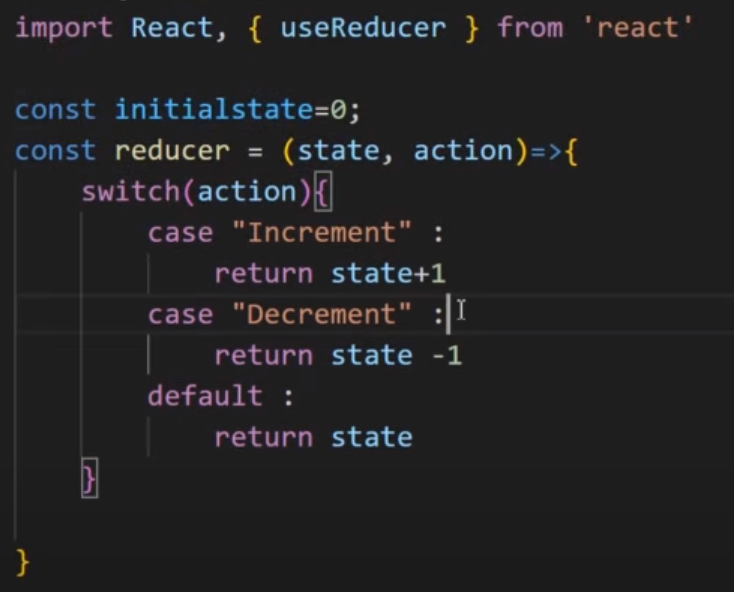
In *Counter.js* define two buttons:



The *Counter* function will look something like:

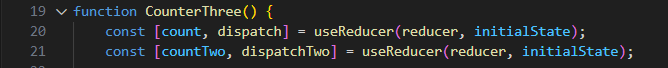


We need to declare the initial state variable (*initialstate*) & the *reducer* function.



We can use multiple *useReducer()* hooks to increase efficiency. In this example, we will make two counter buttons.

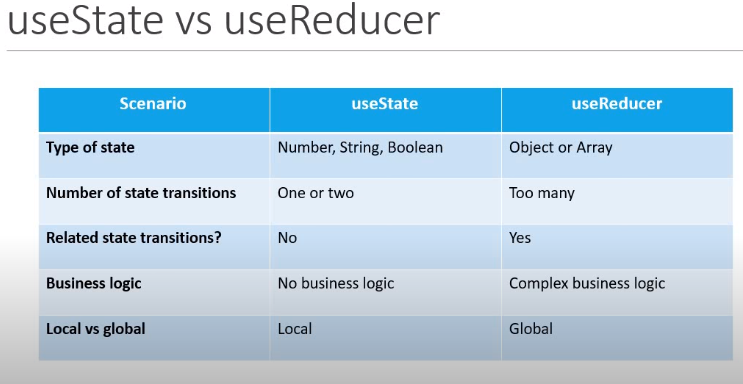
The *reducer* functioncan be reused again although to maintain their states separately, we need two separate *useReducer()* hooks.



In addition to the first button, we add the JSX for the second



**useState vs useReducer**



**useCallback hook**

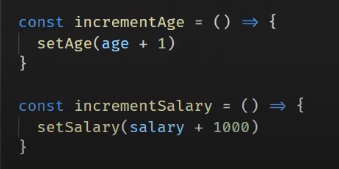
*useCallback* is a hook that will return a memorized version of the callback function that only changes if one of the dependencies has changed.

It is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders.

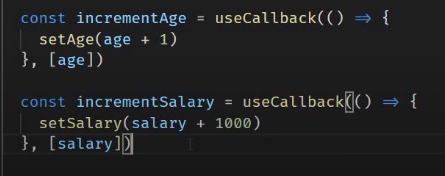
The syntax of *useCallback* hook is:

*const cachedFunc = useCallback (function, dependencies)*

without using *useCallback:*



using *useCallback:*

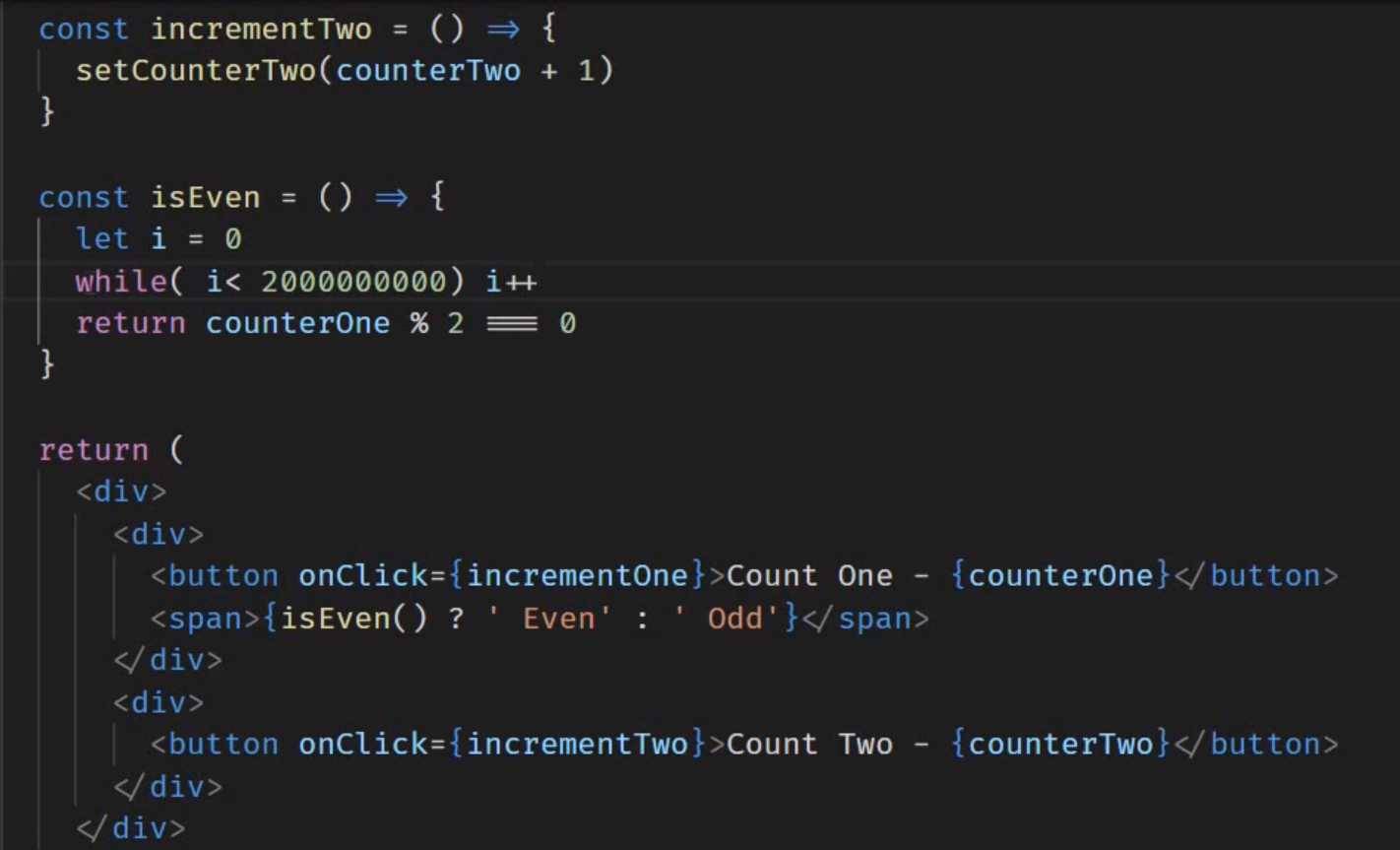


**useMemo hook**

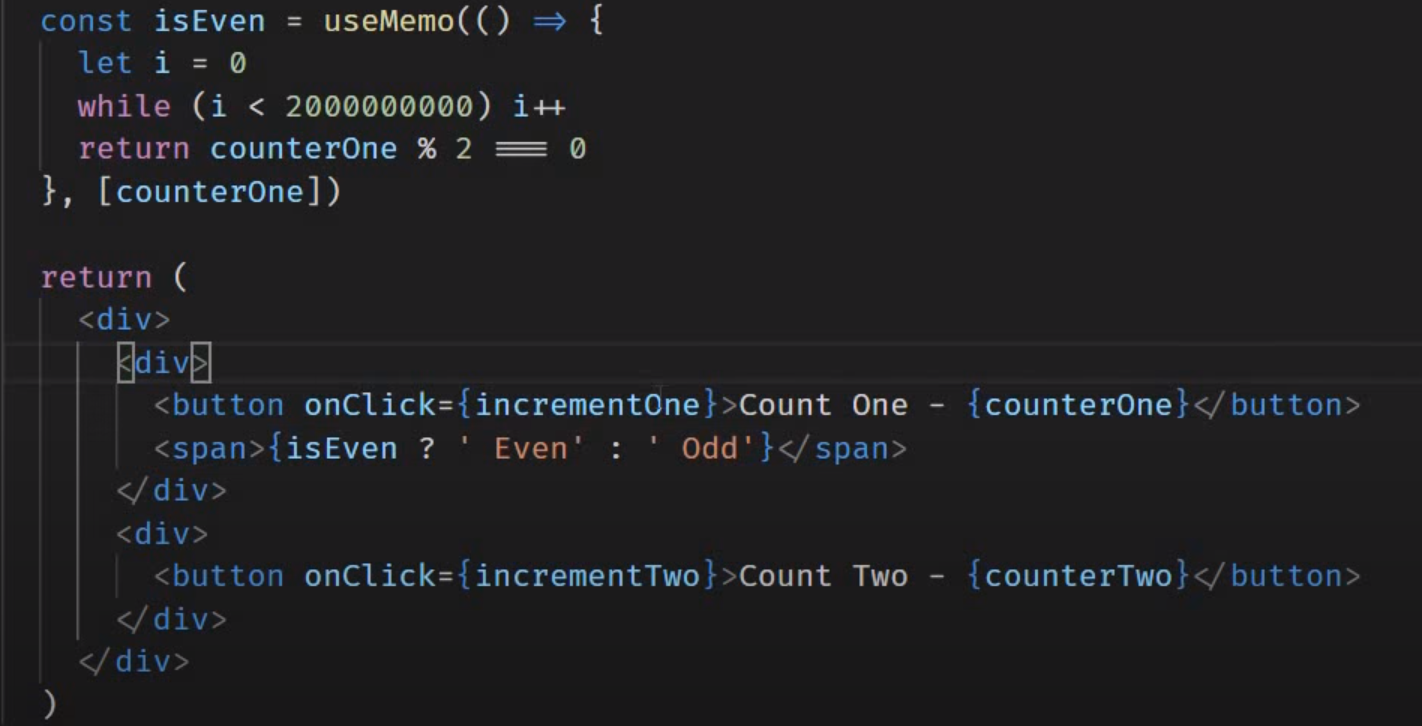
The *useMemo* hook returns a memorized value. Think of memorization as caching a value so that it does not need to be recalculated.

The useMemo hook only runs when one of its dependencies update.

without using *useMemo:*



using *useCallback:*



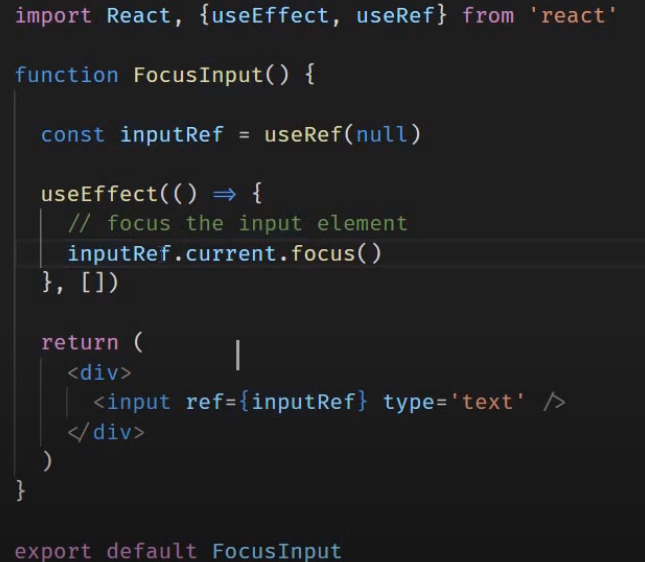
The *useMemo* hook in *incrementOne* buttonprevents re-rendering of *incrementTwo button.*

The *useMemo* and *useCallback* Hooks are similar. The main difference is that *useMemo* returns a memorized value and *useCallback* returns a memorized function.

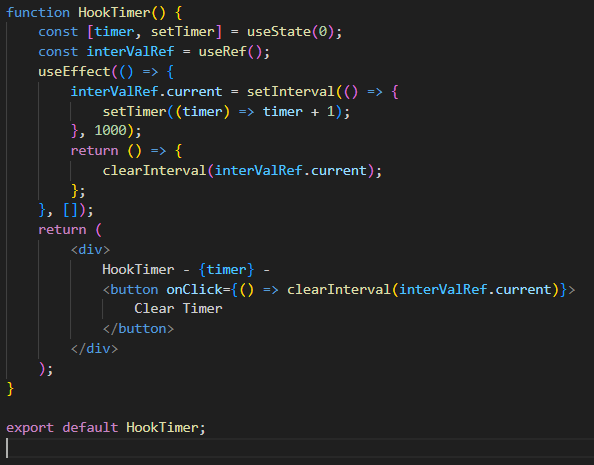
**useRef hook**

*useRef* hook can be used to access a DOM element directly.

In this example, we want to automatically want the *input* field to be focused when the page renders. *useRef hook* only returns one item. It returns an object called current.



In this example, we want to pass the *clearInterval()* as a cleanup function& also pass it in a *onClick().* It allows you to persist values between renders. It can be used to store a mutable value that does not cause a re-render when data inside it is updated. It also remembers the stored data even after other state variables cause a re-render.

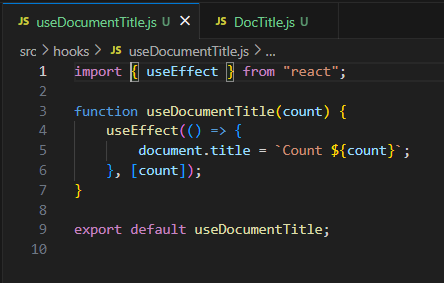


**Custom hook**

Building our own Hooks lets us extract component logic into reusable functions.

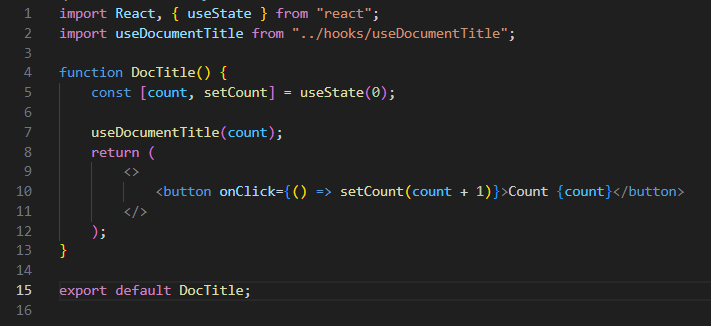
In this example, we make a hook that updates the title of the page. Create a file with naming convention ‘use-‘ to denote a hook.

*useDocumentTitle.js :*



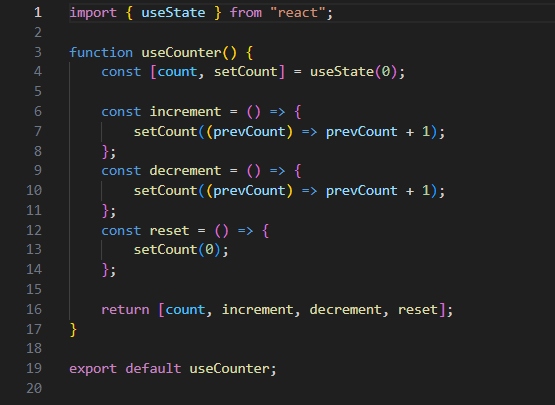
Import this hook in the file you desire.

*DocTitle.js:*

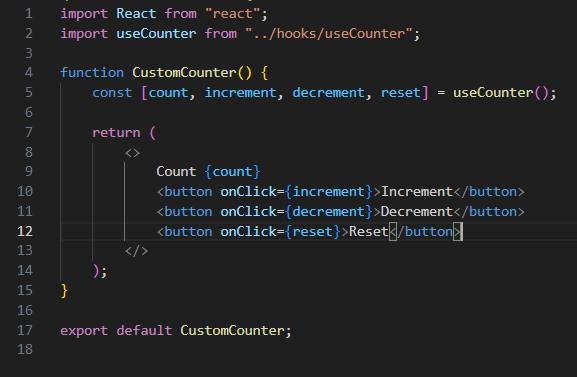


In another example, we create a hook for a counter.

*useCounter.js :*



*CustomCounter.js:*



We can further customize the hooks.

