2/7/21

* Pure - Component Issues →

→ So, lets suppose I have an array and on the click of the button I want the array to get updated with one more item which I will be pushing.

→ So, when we go ahead and do that we will see that re-rendering won't be occurring fo even if the array is updated.

→ So, it is only advised to use Pure

Components with simple variables and not with arrays, lists or objects of any sort.

→ Why is it happening because what Pure Components does is shallow comparison.

[So, when we try to change a value inside the array the reference to that array remains the same]

{ Which means it thinks that it is the same and there is no change happened so, it doesn't renders. }

* React - Hooks → So, as we learned earlier that Class Components are stateful components whereas functional components are stateless components.

→ [So, with the help of hooks, functional Components are no longer stateless as we can go ahead and use all the properties of state in functional component using "Hooks".]

* Definition → Hooks are functions that let you "hook into" React state and lifecycle features from functional components.

→ Hooks doesn't work inside classes — they let you use React without classes.

"[Hooks were introduced in "React-16.8"]"

→ So, in order to use hooks we have two methods →

(1) for State
(2) for lifecycles.

(1) For state we have a method called 'useState' we just have go ahead and import it.

So, this "useState()" ~~fvas~~ when consoled gives us an array with two values.

at [0] index the value that we need to change,

and at [1] index the function that is supposed to change the value of the variable.

Ex → import { useState } from 'react'

```
Const App = () => {
    let Count = useState(1);
Console.log(Count);
return (
  <h1>
      { 'React Hooks ${Count[0]}' }
  </h1>
)
}

Export default App;
```

(1) → initial value that I want to pass and the [0]th index will be having 1 as the value]

→ [So, whenever I want to access the value I will have to use [0]. and to access the method [1].]

So, their is a better way to do that and is known as "Array Destructuring"

Syntax →

```
let [Count, setCount] = useState
```

Count → value getting modified

setCount → function responsible for modifying the value.

```
return (
  <h1>
    { 'React Hooks ${Count}' }
  </h1>
)
```

* So, now what if I want that on the click of that button value keeps on increasing →

Ex →

```
const App = () =>
{
  let [count, setCount] = useState(1)
  console.log(count);
  return (
    <>
    <h1> {`React Hook ${count}`} </h1>
    <button onClick={() => {setCount(count+1)}}
      > Update </button>
    </>
  )
}
export default App;
```

* More on UseState →

→ In order to have multiple states we can go ahead and use it by having different variables set for them.

→ While using objects as the parameter inside our useState() method we must keep one thing in mind that if our objects is having multiple values then the function with setState which will be containing the object with updated values will be considered as a whole new object and so the properties defined previously in the previous object will just disappear, so all properties must be included in the setState.

for explanation refer to App.js