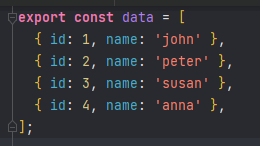
* Hooks starts with use.
* Components must be uppercase
* Hook must be inside the component body, or function
* We can’t call hook conditionally (for example, inside if block).
* **UseState:**

It returns an array. first value of array is the initial value and second one is the handler function.

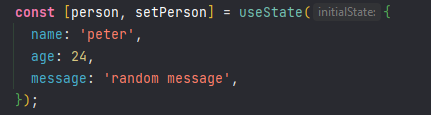


To change the value of text, we need to use setText function. It will trigger a re-render.

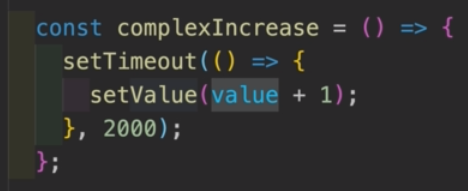
Array:

Object:

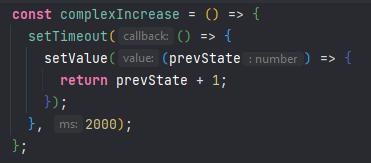


Async Example:

In this case, there is a delay between execution. So if we keep clicking the button that have this handler, It will take value = 0 for 2 sec. So, the value won’t be updated.

In this case, we can pass a function instead of a value inside the **setValue** function. The advantage here, that it gets the previous state value as parameter. Thus the issue can be solved.



* **UseEffect:**

It is used for side effect (means any work outside of the component). For example, changing document title, signing up for subscription, fetching data, setting up an event listener etc.

1. **Conditionally running logic:**

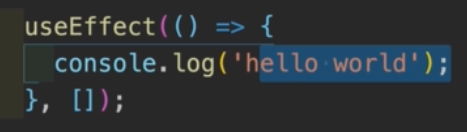
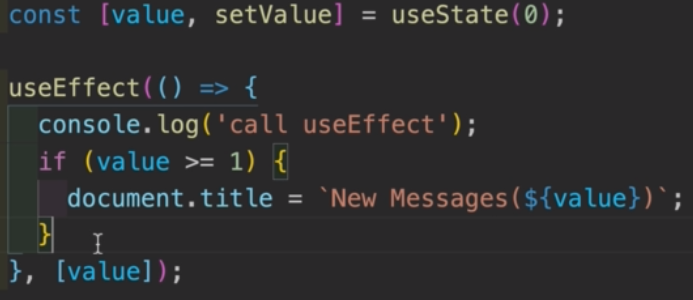
By default, it run once after every re-render. In many cases we might want to call useEffect conditionally. But in general, we can not call hooks conditionally in react.

However, just like useState, we can pass a function in side the useEffect hook. Inside that function we can use condition as much as we want.

1. **Second Parameter/Dependency List:**

Apart from the callback function, we can pass an array as second argument to useEffect. It is a list of dependencies. If we leave this array blank, it will only run on the initial render. If we put a value in there, useEffect will only run when the value updates. We can have multiple useEffect.

1. **Cleanup function:**

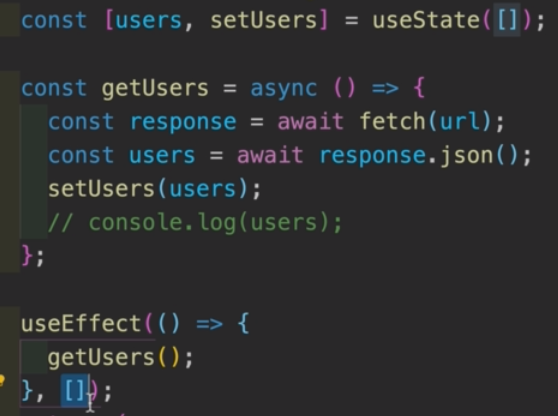
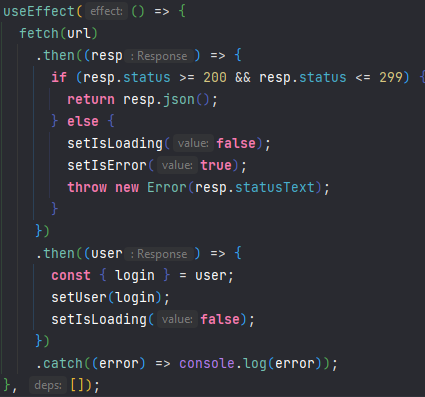
Event listeners can cause memory leaks if not cleaned up properly. In this case, we are adding an event listener everytime window is resized. But the problem is we are doing this in setSize function, which is causing a re-render. In this case a new event listener is added and it will keep growing.

To solve this, we can return a function from the first argument of useEffect. This will be invoked once we exit. Here we can do the cleanup work.

1. **Fetch Data:**

**Important**: we can not use async await to the callback function of the useEffect (first parameter). We can do this inside the callback function, or setup the function separately and call it inside.

Here we are calling the async function inside the useEffect callback function. It is important to setup the dependency list as an empty array in this case. Otherwise it will cause an infinite loop of re-render (useEffect runs on every re-render and it calls getUser() which in turn calls setUser which causes a re-render)

Second example is for conditional rendering while fetching a data. Here we have logic to throw error in case server returns error response.

* **UseRef:**

This hook is used for uncontrolled element. Most popular use of this hook is to target a dom element to have uncontrolled input (similar to javascript).