2 Using the State Hook

Exercise 1: Reimplementing the useState function

In order to get a better understanding of how Hooks work internally, we are going to reimplement the `useState` Hook from scratch. However, we are not going to implement it as an actual React Hook, but as a simple JavaScript function—just to get an idea of what Hooks are actually doing.

> Please note that this reimplementation is not exactly how React Hooks work internally. The actual implementation is similar, and thus, it has similar constraints. However, the real implementation is much more complicated than what we will be implementing here.

Step 1: We are now going to start reimplementing the State Hook:

1. First, we copy the code from `chapter1_2`, where we are going to replace the current `useState` Hook with our own implementation.
2. Open `src/App.js` and remove the import of the Hook by removing the following line:

   ```
   import React, { useState } from 'react'
   ```

   Replace it with these lines of code:

   ```
   import React from 'react'
   import ReactDOM from 'react-dom'
   ```

   We are going to need `ReactDOM` in order to force rerendering of the component in our reimplementation of the `useState` Hook. If we used actual React Hooks, this would be dealt with internally.

3. Now, we define our own `useState` function. As we already know, the `useState` function takes the `initialState` as an argument:

   ```
   function useState (initialState) {
   ```

4. Then, we define a value, where we will store our state. At first, this value will be set to `initialState`, which is passed as an argument to the function:

   ```
       let value = initialState
   ```

5. Next, we define the `setState` function, where we will set the value to something different, and force the rerendering of our `MyName` component:

   ```
       function setState (nextValue) {
           value = nextValue
           ReactDOM.render(<MyName />,
               document.getElementById('root'))
       }
   ```

6. Finally, we return the `value` and the `setState` function as an array:

```
            return [ value, setState ]
        }
```

The reason why we use an array, and not an object, is that we usually want to rename the value and setState variables. Using an array makes it easy to rename the variables through destructuring:

```
const [ name, setName ] = useState('')
```

As we can see, Hooks are simple JavaScript functions that deal with side effects, such as setting a stateful value.

Our Hook function uses a closure to store the current value. The closure is an environment where variables exist and are stored. In our case, the function provides the closure, and the `value` variable is stored within that closure. The `setState` function is also defined within the same closure, which is why we can access the `value` variable within that function. Outside of the `useState` function, we cannot directly access the `value` variable unless we return it from the function.

Exercise 2: Problems with our simple Hook implementation

If we run our Hook implementation now, we are going to notice that when our component rerenders, the state gets reset, so we cannot enter any text in the field. This is due to the reinitialization of the `value` variable every time our component gets rendered, which happens because we call `useState` each time we render the component.

In the upcoming sections, we are going to solve this problem by using a global variable and then turn the simple value into an array, allowing us to define multiple Hooks.

Step 1: Using a global variable

As we have learned, the value is stored within the closure that is defined by the `useState` function. Every time the component rerenders, the closure is reinitialized, which means that our value will be reset. To solve this, we need to store the value in a global variable, outside of the function. That way, the `value` variable will be in the closure outside of the function, which means that when the function gets called again, the closure will not be reinitialized.

We can define a global variable as follows:

1. First, we add the following line (in bold) above the `useState` function definition:

```
let value

function useState (initialState) {
```

2. Then, we replace the first line in our function with the following code:

```
if (typeof value === 'undefined') value = initialState
```

Now, our `useState` function uses the global `value` variable, instead of defining the `value` variable within its closure, so it will not be reinitialized when the function gets called again.

Defining multiple Hooks

Our Hook function works! However, if we wanted to add another Hook, we would run into another problem: all the Hooks write to the same global `value` variable!

Let's take a closer look at this problem by adding a second Hook to our component.

Step 2: Adding multiple Hooks to our component

Let's say we want to create a second field for the last name of the user, as follows:

1. We start by creating a new Hook at the beginning of our function, after the current Hook:

```
const [ name, setName ] = useState('')
const [ lastName, setLastName ] = useState('')
```

2. Then, we define another `handleChange` function:

```
function handleLastNameChange (evt) {
    setLastName(evt.target.value)
}
```

3. Next, we place the `lastName` variable after the first name:

```
<h1>My name is: {name} {lastName}</h1>
```

4. Finally, we add another `input` field:

```
<input type="text" value={lastName}
     onChange={handleLastNameChange}
 />
```

When we try this out, we are going to notice that our reimplemented Hook function uses the same value for both states, so we are always changing both fields at once.

Step 3: Implementing multiple Hooks

In order to implement multiple Hooks, instead of having a single global variable, we should have an array of Hook values.

We are now going to refactor the `value` variable to a `values` array so that we can define multiple Hooks:

1. Remove the following line of code:

   ```
   let value
   ```

   Replace it with the following code snippet:

   ```
   let values = []
   let currentHook = 0
   ```
2. Then, edit the first line of the `useState` function where we now initialize the value at the `currentHook` index of the `values` array:

   ```
   if (typeof values[currentHook] === 'undefined')
      values[currentHook] = initialState
   ```
3. We also need to update the setter function, so that only the corresponding state value is updated. Here, we need to store the `currentHook` value in a separate `hookIndex` variable, because the `currentHook` value will change later. This ensures that a copy of the `currentHook` variable is created within the closure of the `useState` function. Otherwise, the `useState` function would access the `currentHook` variable from the outer closure, which gets modified with each call to `useState`:

   ```
   let hookIndex = currentHook
    function setState (nextValue) {
       values[hookIndex] = nextValue
       ReactDOM.render(<MyName />,
          document.getElementById('root'))
   }
   ```
4. Edit the final line of the `useState` function, as follows:

   ```
   return [ values[currentHook++], setState ]
   ```

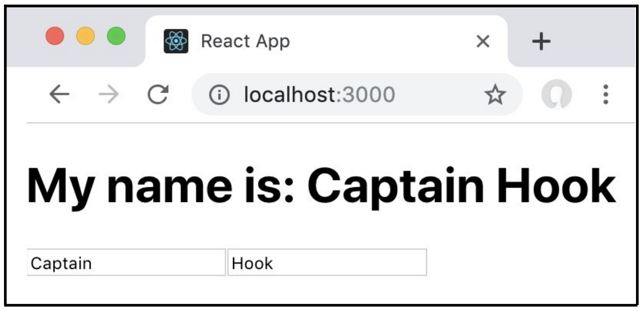
   Using `values[currentHook++]`, we pass the current value of `currentHook` as an index to the `values` array, and then increase `currentHook` by one. This means that `currentHook` will be increased after returning from the function.

   If we wanted to first increment a value and then use it, we could use the `arr[++indexToBeIncremented]` syntax, which first increments, and then passes the result to the array.

5. We still need to reset the `currentHook` counter when we start rendering our component. Add the following line (in bold) right after the component definition:

   ```
   function Name () {
       currentHook = 0
   ```

Finally, our simple reimplementation of the `useState` Hook works! The following screenshot highlights this:



My name is: Captain Hook

Captain Hook

Our custom Hook reimplementation works

As we can see, using a global array to store our Hook values solved the problems that we had when defining multiple Hooks.

Exercise 3: Can we define conditional Hooks? (chapter2_2)

What if we wanted to add a checkbox that toggles the use of the first name field?

Step 1: Let's find out by implementing such a checkbox:

1. First, we add a new Hook in order to store the state of our checkbox:

   ```
   const [ enableFirstName, setEnableFirstName ] = useState(false)
   ```

2. Then, we define a handler function:

   ```
   function handleEnableChange (evt) {
       setEnableFirstName(!enableFirstName)
   }
   ```

3. Next, we render a checkbox:

   ```
   <input type="checkbox" value={enableFirstName}
      onChange={handleEnableChange} />
   ```

4. We do not want to show the first name if it is disabled. Edit the following existing line in order to add a check for the `enableFirstName` variable:

   ```
   <h1>My name is: {enableFirstName ? name : ''}
      {lastName}</h1>
   ```

5. Could we put the Hook definition into an `if` condition, or a ternary expression, like we are in the following code snippet?

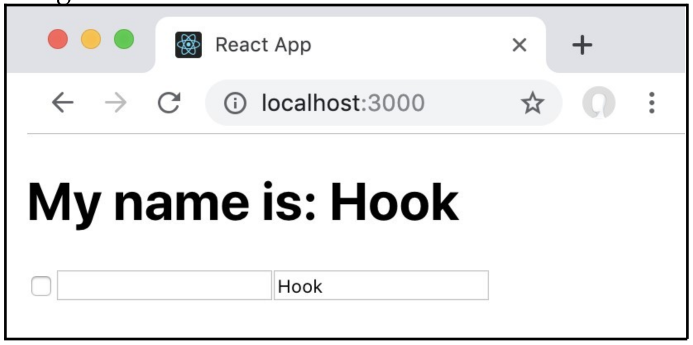
   ```
   const [ name, setName ] = enableFirstName
       ? useState('')
       : [ '', () => {} ]
   ```

6. The latest version of `react-scripts` actually throws an error when defining conditional Hooks, so we need to downgrade the library for this example, by running the following command:

   ```
   > npm install --save react-scripts@^2.1.8
   ```

Here, we either use the Hook, or if the first name is disabled, we return the initial state and an empty setter function so that editing the input field will not work.

Step 2: If we now try out this code, we are going to notice that editing the last name still works, but editing the first name does not work, which is what we wanted. As we can see in the following screenshot, only editing the last name works now:



My name is: Hook

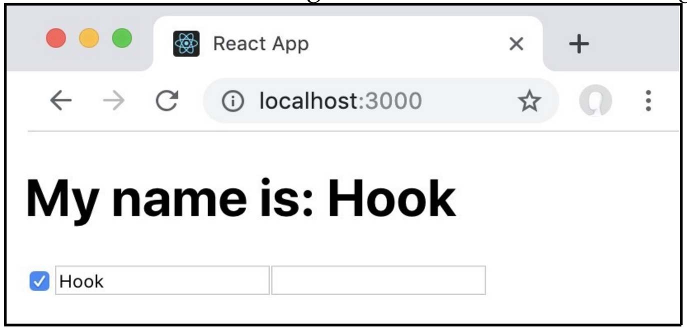
Hook

State of the app before checking the checkbox

When we click the checkbox, something strange happens:

- The checkbox is checked
- The first name input field is enabled
- The value of the last name field is now the value of the first name field

Step 3: We can see the result of clicking the checkbox in the following screenshot:



My name is: Hook

Hook

State of the app after checking the checkbox

We can see that the last name state is now in the first name field. The values have been swapped because the order of Hooks matters. As we know from our implementation, we use the `currentHook` index in order to know where the state of each Hook is stored. However, when we insert an additional Hook in-between two existing Hooks, the order gets messed up.

Before checking the checkbox, the `values` array was as follows:

- `[false, '']`
- Hook order: `enableFirstName`, `lastName`

Then, we entered some text in the `lastName` field:

- `[false, 'Hook']`
- Hook order: `enableFirstName`, `lastName`

Next, we toggled the checkbox, which activated our new Hook:

- `[true, 'Hook', '']`
- Hook order: `enableFirstName`, `name`, `lastName`

As we can see, inserting a new Hook in-between two existing Hooks makes the `name` Hook steal the state from the next Hook (`lastName`) because it now has the same index that the `lastName` Hook previously had. Now, the `lastName` Hook does not have a value, which causes it to set the initial value (an empty string). As a result, toggling the checkbox puts the value of the `lastName` field into the `name` field.

Solving common problems with Hooks

As we found out, implementing Hooks with the official API also has its own trade-offs and limitations. We are now going to learn how to overcome these common problems, which stem from the limitations of React Hooks.

We will take a look at solutions that can be used to overcome these two problems:

- Solving conditional Hooks
- Solving Hooks in loops

Exercise 4: Solving conditional Hooks

So, how do we implement conditional Hooks? Instead of making the Hook conditional, we can always define the Hook and use it whenever we need it. If this is not an option, we need to split up our components, which is usually better anyway!

Always defining the Hook

For simple cases, such as the first and last name example that we had previously, we can just always keep the Hook defined, as follows:

```
const [ name, setName ] = useState('')
```

Always defining the Hook is usually a good solution for simple cases.

Splitting up components

Another way to solve conditional Hooks is to split up one component into multiple components and then conditionally render the components. For example, let's say we want to fetch user information from a database after the user logs in.

We cannot do the following, as using an `if` conditional could change the order of the Hooks:

```
function UserInfo ({ username }) {
    if (username) {
        const info = useFetchUserInfo(username)
        return <div>{info}</div>
    }
    return <div>Not logged in</div>
}
```

Instead, we have to create a separate component for when the user is logged in, as follows:

```
function LoggedInUserInfo ({ username }) {
    const info = useFetchUserInfo(username)
    return <div>{info}</div>
}

function UserInfo ({ username }) {
    if (username) {
        return <LoggedInUserInfo username={username} />
    }
    return <div>Not logged in</div> }
```

Using two separate components for the non-logged in and logged in state makes sense anyway, because we want to stick to the principle of having one functionality per component. So, usually, not being able to have conditional Hooks is not much of a limitation if we stick to best practices.

Exercise 5: Solving Hooks in loops

As for Hooks in loops, we can either use a single State Hook containing an array, or we can split up our components. For example, let's say we want to display all the users that are online.

Using an array

We could simply use an array that contains all `users`, as follows:

```
function OnlineUsers ({ users }) {
    const [ userInfos, setUserInfos ] = useState([])
    // ... fetch & keep userInfos up to date ...
    return (
        <div>
          {users.map(username => {
            const user = userInfos.find(u => u.username === username)
                return <UserInfo {...user} />
            })}
        </div>
    )
}
```

However, this might not always make sense. For example, we might not want to update the `user` state through the `OnlineUsers` component because we would have to select the correct `user` state from the array, and then modify the array. This might work, but it is quite tedious.

Splitting up components

A better solution would be to use the Hook in the `UserInfo` component instead. That way, we can keep the state for each user up to date, without having to deal with array logic:

```
function OnlineUsers ({ users }) {
    return (
        <div>
            {users.map(username => <UserInfo username={username} />)}
        </div>
    )
}

function UserInfo ({ username }) {
    const info = useFetchUserInfo(username)
    // ... keep user info up to date ...
    return <div>{info}</div>
}
```

As we can see, using one component for each functionality keeps our code simple and concise, and also avoids the limitations of React Hooks.

Exercise 6: Solving problems with conditional Hooks(chapter2_3)

Now that we have learned about the different alternatives to conditional Hooks, we are going to solve the problem that we had in our small example project earlier. The simplest solution to this problem would be to always define the Hook, instead of conditionally defining it. In a simple project like this one, always defining the Hook makes the most sense.

Edit `src/App.js` and remove the following conditional Hook:

```
    const [ name, setName ] = enableFirstName
        ? useState('')
        : [ '', () => {} ]
```

Replace it with a normal Hook, such as the following:

```
    const [ name, setName ] = useState('')
```

Now, our example works fine! In more complex cases, it might not be feasible to always define the Hook. In that case, we would need to create a new component, define the Hook there, and then conditionally render the component.