Common Mistakes with Hooks:

6 Common mistakes with Hooks:

Type 1: Forgetting to spread the previous state value into the new one when updating arrays with useState

Common Mistake	Solution

```
const updateNums = () => {
  setNums([1])
}
```
	```
	const updateNums = () => {
	  setNums([...nums, 1])
	}
	```

Common Mistake:
The entire array is replaced with **only** the number provided, since useState does not merge state updates like setState does.

Solution:
In general, when you are updating an array in the state, remember to use the spread operator to spread the existing values into the updated value.

Type 2: Forgetting to spread the previous state value into the new one when updating objects with useState

Common Mistake	Solution

```
onChange={e => setName({
  firstName: e.target.value
})}

onChange={e => setName({
  lastName: e.target.value
```
	```
	onChange={e => setName({
	  ...name,
	  firstName: e.target.value
	})}

	onChange={e => setName({
	  ...name,
	```

`})}`	`  lastName: e.target.value` `})}`

Common Mistake:
Similar to Mistake 1: When updating one form field, the opposite state property is removed from the state e.g. firstName vs lastName.

Solution:
Best practice is to just use the spread operator to copy the entire object, and only update what you need to.

Type 3: Forgetting to specify the dependency array - the second parameter - to useEffect.

Common Mistake	Solution
`useEffect(() => {` `  console.log('count 1 effect')` `  document.title = count;` `})`	`useEffect(() => {` `  console.log('count 1 effect')` `  document.title = count;` `}, [count])`

Common Mistake:
The first common mistake developers usually make with useEffect is completely forgetting to specify the dependency array - that is - the second parameter which specifies the variables the component should be watching for changes.

Solution:
Make sure to add a second parameter to your useEffect Hook. Otherwise, your Hook will run after any change on the page.

Type 4: Specifying the dependency array to useEffect incorrectly or accidentally leaving required dependencies out of it

Common Mistake	Solution
`function updateTitle() {`	`function updateTitle() {`
`  document.title = count;`	`  document.title = count;`
`}`	`}`
`useEffect(() => {`	`useEffect(() => {`
`  console.log('count 1 effect');`	`  console.log('count 1 effect');`
`  updateTitle();`	`  updateTitle();`
`}, [])`	`}, [count])`

Common Mistake:
Similar to above, instead of leaving out the second parameter, another common mistake is to add the incorrect dependency as the parameter.

Solution:
Make sure to think about what part of the Component Lifecycle you are trying to replicate with the useEffect Hook:

1. componentDidMount: [ ] - empty array
2. componentDidUpdate: [count] - array containing dependency

Type 5: Forgetting to specify or incorrectly specifying a cleanup function when using useEffect

Common Mistake	Solution
`useEffect(() => {`	`useEffect(() => {`
`  console.log('Creating timer')`	`  console.log('Creating timer')`
`  const interval = setInterval(()`	`  const interval = setInterval(()`
`=> {`	`=> {`
`    console.log('Interval`	`    console.log('Interval`
`executed')`	`executed')`

```
    setTime(time => time + 1)
  }, 1000)

}, [])
```

```
    setTime(time => time + 1)
  }, 1000)
  return () => {
    console.log('cleaning up!')
    clearInterval(interval);
  }
}, [])
```

Common Mistake:
Often when your components have side effects, cleanup will be required when the component unmounts.

Solution:
To execute a function when your component unmounts, remember that all you need to do is return an arrow function inside useEffect and call your function there.