

Handling Props with the Same Property Name

In this lesson, we'll see how two prop properties with the same name can be invoked at once.

WE'LL COVER THE FOLLOWING ^

- Overriding the User's `onClick` Handler
- Invoking Both User's and Internal `onClick`
- How `callFunctionsInSequence` Works
- Quick Quiz!

We allowed passing custom props into the `getToggler` props getter. Now, what if the user wanted to pass in a custom `onClick` prop?

Here's what I mean:

```
.Expandable-panel {  
  margin: 0;  
  padding: 1em 1.5em;  
  border: 1px solid hsl(216, 94%, 94%);  
  min-height: 150px;  
}
```

If the user does this, it completely overrides the `onClick` handler within the `useExpanded` custom hook and breaks the functionality of the app as you can see in the output.

The button no longer expands the element, but the user's custom click handler is invoked whenever you click the button.

Why is this the case?

This is because we have an `onClick` property in the returned object, but it is overridden by the user.

```
// useExpandable.js
...
const getTogglerProps = useCallback(
  ({ ...customProps }) => ({
    onClick: toggle, // this is overridden
    'aria-expanded': expanded,
    // customProps overrides the onClick above
    ...customProps
  }),
  [toggle, expanded]
)
```

Overriding the User's `onClick` Handler

If you move the position of the internal `onClick` handler, you can override the user.

```
const getTogglerProps = useCallback(
  ({ onClick, ...props } = {}) => ({
    'aria-expanded': expanded,
    onClick: callFunctionsInSequence(toggle, onClick),
    ...props
  }),
  [toggle, expanded]
)
```

This works; however, we want a component that's as flexible as possible. The goal is to give the user the ability to invoke their own custom `onClick` handler.

How can we achieve this?

Invoking Both User's and Internal `onClick`

Well, we could have the internal `onClick` invoke the user's click handler as well.

Here's how:

```
.Expandable-panel {
  margin: 0;
  padding: 1em 1.5em;
  border: 1px solid hsl(216, 94%, 94%);
  min-height: 150px;
}
```

Now instead of returning an object with the `onClick` set to a single function, i.e. our internal `toggle` function or the user's `onClick` we could set the

`onClick` property to a function, `callFunctionsInSequence`, that invokes both functions - our `toggle` function and the user's `onClick`!

```
...  
onClick: callFunctionsInSequence(toggle, onClick)
```

I've called this function `callFunctionsInSequence` so that it's easy to understand what happens when called.

Also, note how we destructure the user's `onClick` handler and use the rest parameter to handle the other props passed in. To avoid getting an error if no object is passed in by the user, we set a default parameter value of `{}`.

```
...  
// note destructuring and default parameter value of {}  
({ onClick, ...props } = {}) => ({  
  
})
```

Now back to the elephant in the room; how do we write a function that takes one or two functions and invokes all of them without neglecting the passed arguments?

This function cannot break if any of the function arguments are `undefined` either.

Here's one solution to this problem:

```
const callFunctionsInSequence = (...fns) => (...args) =>  
  fns.forEach(fn => fn && fn(...args))
```

When we do this:

```
...  
onClick: callFunctionsInSequence(toggle, onClick) // this invokes callFunctionsInSequence
```

The return value saved in the `onClick` object property is the returned function from invoking `callFunctionsInSequence`:

```
(...args) => fns.forEach(fn && fn(...args))
```



If you remember from basic React, we always attach click handlers like this:

```
...  
render() {  
  return <button onClick={this.handleClick} />  
}
```



We attach the click handler `handleClick`, but in the `handleClick` declaration, we expect an `evt` argument to be passed in at invocation time.

```
// see event object (evt)  
handleClick = evt => {  
  console.log(evt.target.value)  
}  
...
```



How `callFunctionsInSequence` Works

Now back to the returned function from `callFunctionsInSequence`:

```
(...args) => fns.forEach(fn && fn(...args))
```

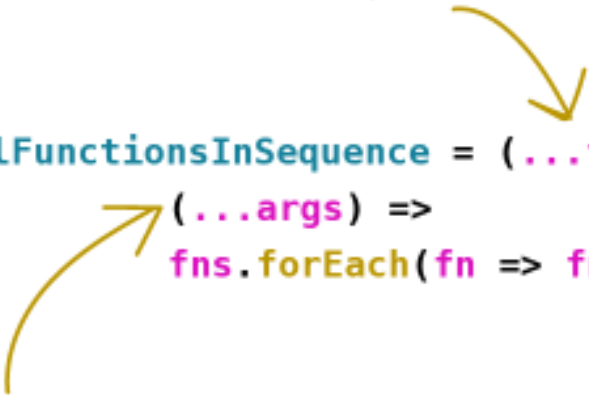


The function receives whatever arguments are passed into the function, `(...args)`, and invokes all function parameters with these arguments if the function is not false. `fn && fn(...args)`.

In this case, the argument received by the function will be the event object and it will be passed down to both `toggle` and `onClick` as arguments.

first invocation takes in the
function parameters

```
const callFunctionsInSequence = (...fns) =>  
  (...args) =>  
    fns.forEach(fn => fn && fn(...args))
```



second invocation takes in whatever arguments are passed in,
loops over the function list and invokes the functions
with the arguments.

The callFunctionsInSequence Function

The full implementation of the custom hook now looks like this:

```
.Expandable-panel {  
  margin: 0;  
  padding: 1em 1.5em;  
  border: 1px solid hsl(216, 94%, 94%);  
  min-height: 150px;  
}
```

Quick Quiz!

Q

What is the suggested solution to the issue of props with the same property name?

COMPLETED 0%

1 of 1



With this addition, our internal click handler, `toggle`, and the user's custom handler, `onClick`, are both invoked when the toggle button is clicked!

How amazing! We wouldn't be able to cater for this case with just a prop collection object. Thanks to prop getters, we have more power to provide reusable components.

From the next chapter on, we're going to be looking at a new pattern: **State Initializers**. Catch you in the next lesson!