React State and Events Code-along

 (https://github.com/learn-co-curriculum/react-hooks-state-and-events-codealong)  (https://github.com/learn-co-curriculum/react-hooks-state-and-events-codealong/issues/new)

Learning Goals

- Use the `useState` hook to create state within a component
- Update state based on events

Introduction

In this code-along, we'll get some hands-on practice writing components with state and setting state based on different kinds of events.

Adding State to a Component

To get some practice adding state to a component, code along with this lesson. There's some starter code in the `src/components` folder. We'll be adding state to some existing components and building out some functionality that depends on state.

As a quick recap:

> State is data that changes over time in your component. State must be initialized in a component by calling `useState`. Updating state by calling `setState` will cause our components to re-render automatically.

To code along, run `npm install && npm start` to install the dependencies and get the demo app running.

Determining When To Add State

For our first component, let's work on a toggle button. It should do the following:

- The button should say "OFF" when it is first displayed

- When the button is clicked, it should say "ON"
- When the button is clicked again, it should say "OFF"
- etc

Let's talk through the steps and how we might think about building a feature like this as a React developer.

First, let's decide: do we actually need state for this feature? We need to determine if the data for this feature is *static* (doesn't change) or *dynamic* (does change). If it's dynamic, we'll definitely need state! We should also ask if this feature could be made using **props** instead of **state**.

Here are some questions from **Thinking in React (https://reactjs.org/docs/thinking-in-react.html#step-3-identify-the-minimal-but-complete-representation-of-ui-state)** that will help us decide if we need state:

- Is it passed in from a parent via props? If so, it probably isn't state.
- Can you compute it based on any other state or props in your component? If so, it isn't state.
- Does it remain unchanged over time? If so, it probably isn't state.

Since this component isn't being passed any props that will let us display some new button text, and the button's text is *dynamic* (it changes), we definitely need to add state!

Our full checklist looks like this:

- 🚫 Is it passed as a prop?
- 🚫 Can you compute it based on any other state or props in your component?
- 🚫 Does it remain unchanged over time?

So it's time to add state! There's some starter code in the `Toggle.js` file. If you're feeling good about what you learned in the last lesson, give it a shot now! We'll walk through the steps below.

...

...

...

...

Ok, hope you were able to get that going! Here's our process for adding state to build out this feature.

1. Import the `useState` hook

Any time we need state in a component, we need to use the `useState` hook from React. We can import it like so:

```
import React, { useState } from "react";
```

2. Set up the initial state

To create a state variable in our component, we need to call `useState` and provide an initial value:

```
function Toggle() {
  const [isOn, setIsOn] = useState(false);
  // ... the rest of Toggle component
}
```

Whenever you're using a React hook, it **must** be within a React component. We're setting the initial state here as `false`, because the button should be "OFF" when the component first renders.

3. Use the state variable in the component

Now that we have this new variable, it's time to use it! We can use the `isOn` variable to determine what text to display in the button:

```
<button>{isOn ? "ON" : "OFF"}</button>
```

Here, we're doing some **conditional rendering (https://reactjs.org/docs/conditional-rendering.html)** to dynamically determine the button's text *based on our state variable*.

You should now be able to change the initial state in the `useState` function and see if your button's text displays what you expect. Setting an initial state of `true` should display "ON", and `false` should display "OFF".

4. Call the setter function to update state

Any time we want to *update* state, we need to use the *setter function* returned by calling `useState`. We also need to determine what triggers that update. In our case it's the button being clicked.

Let's start by adding an `onClick` handler to the button:

```
<button onClick={handleClick}>{isOn ? "ON" : "OFF"}</button>
```

Next, let's set up the `handleClick` callback function, and update state. Here, we must call the *setter function* to update our state variable. Trying to update the variable directly won't have any effect (even if we changed our variable declaration to `let` instead of `const`):

```
let [isOn, setIsOn] = useState(false);
function handleClick() {
  // updating state directly is a no-no!
  isOn = !isOn;
}
```

So the way we should update state looks like this:

```
function handleClick() {
  setIsOn((isOn) => !isOn);
}
```

All together, here's our updated component:

```
function Toggle() {
  const [isOn, setIsOn] = useState(false);

  function handleClick() {
    setIsOn((isOn) => !isOn);
  }
```

```
  return <button onClick={handleClick}>{isOn ? "ON" : "OFF"}</button>;
}
```

Adding More Features

With this state variable in place, let's add another feature to our button. When the button is ON, let's make the background red, like this:

```
<button style={{ background: "red" }}>
```

When it's OFF, it should have a white background.

Let's go through those same questions to determine if we need to add state for this feature.

- 🚫 Is it passed as a prop?
- ✅ Can you compute it based on any other state or props in your component?
- 🚫 Does it remain unchanged over time?

In this case, we **can** compute it based on other state in our component, so we don't need to add any *new* state for this feature.

We can use that same `isOn` state variable, along with some conditional rendering, to get the button to display the correct border color:

```
function Toggle() {
  const [isOn, setIsOn] = useState(false);

  function handleClick() {
    setIsOn((isOn) => !isOn);
  }

  const color = isOn ? "red" : "white";

  return (
    <button style={{ background: color }} onClick={handleClick}>
      {isOn ? "ON" : "OFF"}
    </button>
```

```
    );
  }
```

Conclusion

Thinking like a React developer involves making a lot of decisions about how to structure your components, particularly when it comes to **props** and **state**. Now that you've seen the process and some common patterns for working with state, it's up to you to apply these decisions to your own components moving forward.

Resources

- **The useState hook (https://reactjs.org/docs/hooks-state.html)**
- **Props vs. State (https://github.com/uberVU/react-guide/blob/master/props-vs-state.md)**
- **Thinking in React (https://reactjs.org/docs/thinking-in-react.html#step-3-identify-the-minimal-but-complete-representation-of-ui-state)**