The State Hook

Let's learn about the useState hook and how to use it. You'll create your first React component with hooks.

WE'LL COVER THE FOLLOWING

As stated earlier, hooks are functions. Officially, there are **10** of them. *10* new functions that exist to make writing and sharing functionalities in your components a lot more expressive.

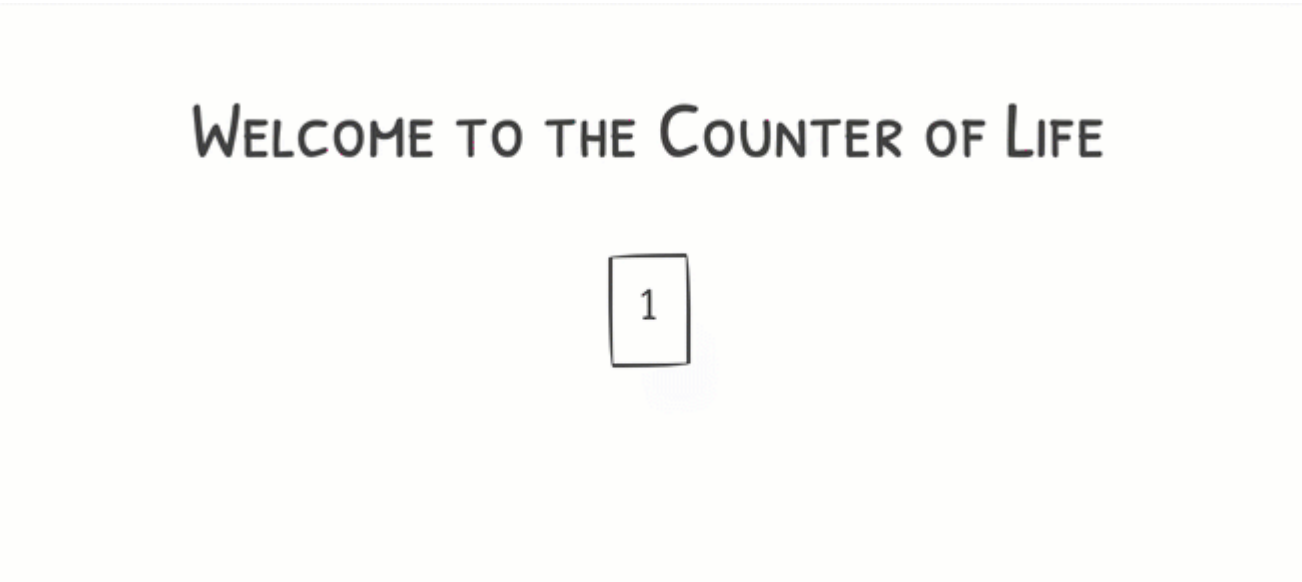
`useState` Hook

The first hook we'll take a look at is called `useState`.

For a long time, you couldn't use the local state in a functional component. Not until hooks.

With `useState`, your functional component can use and update local state. How interesting!

Consider the following counter application:



With the `Counter` component shown below:

```
import React, { Component } from 'react';
class Counter extends Component {
 state = {
    count: 1
 }

  handleClick = () => {
    this.setState(prevState => ({count: prevState.count + 1}))
  }
  render() {
    const { count } = this.state;
    return (
      <div>
        <h3 className="center">
            Welcome to the Counter of Life
        </h3>
        <button className="center-block" onClick={this.handleClick}>
            {count}
        </button>
      </div>
    );
  }
}
```

Simple, huh? Let me ask you this; why exactly do we have this component as a `class` component?

The answer is simple: we need to keep track of some local state within the component.

Here's the same component refactored to a functional component with access to the state via the `useState` hooks.

```
function CounterHooks() {
  const [count, setCount] = useState(0);
  const handleClick = () => {
    setCount(count + 1);
  }
  return (
      <div>
        <h3 className="center">
          Welcome to the Counter of Life
        </h3>
        <button
           className="center-block"
           onClick={handleClick}>
         {count}
        </button>
      </div>
```

```
    );
}
```

What's different? I'll walk you through it step by step.

A functional component doesn't have all the `class extend ...` syntax.

```
function CounterHooks() {
}
```

It also doesn't require a render method.

```
function CounterHooks() {
  return (
      <div>
        <h3 className="center">Welcome to the Counter of Life </h3>
        <button
          className="center-block"
          onClick={this.handleClick}> {count} </button>
      </div>
    );
}
```

There are two concerns with the code above.

- You're not supposed to use the `this` keyword in function components.
- The `count` state variable hasn't been defined.

Extract `handleClick` to a separate function within the functional component:

```
function CounterHooks() {
  const handleClick = () => {
  }
  return (
      <div>
        <h3 className="center">Welcome to the Counter of Life </h3>
        <button
          className="center-block"
          onClick={handleClick}> {count} </button>
      </div>
    );
}
```

Calling `useState`

Before the refactor, the `count` variable came from the class component's state object.

In functional components, and with hooks, that comes from invoking the `useState` function or hook.
`useState` is called with one argument, the initial state value, e.g., `useState(0)`, where **0** represents the initial state value to be kept track of.

Invoking this function returns an array with two values.

```
// returns an array with 2 values.
useState(0)
```

The first value is the current `state` value being tracked, and the second, a function to update the `state` value.

Think of this as some `state` and `setState` replica - however, they aren't quite the same.

With this new knowledge, here's `useState` in action.

```
function CounterHooks() {
  // 
  const [count, setCount] = useState(0);
  const handleClick = () => {
    setCount(count + 1)
  }
  return (
      <div>
        <h3 className="center">Welcome to the Counter of Life </h3>
        <button
          className="center-block"
          onClick={handleClick}> {count} </button>
      </div>
    );
}
```

There are a few things to note here, apart from the obvious simplicity of the code!

One, since invoking `useState` returns an array of values, the values could be easily destructed into separate values as shown below:

```
const [count, setCount] = useState(0);
```

Also, note how the `handleClick` function in the refactored code doesn't need any reference to `prevState` or anything like that.

It just calls `setCount` with the new value, `count + 1`.

As simple as it sounds, you've built your very first component using hooks. I know it's a contrived example, but that's a good start!

It's also possible to pass a function to the state updater function. This is usually recommended as with `setState` when a state update depends on a previous value of state, e.g., `setCount(prevCount => prevCount + 1)`

In the next lesson, we'll learn how to use multiple `useState` calls in a single component.