React State

State vs. Props

We have already seen properties.

```
<NewsArticle title="SDG Announces New Cohort" body="..." />

<NewsArticle title="React Version 17 is Released" body="..." />
```

Props

- Passed from parent to child
- Cannot modify inside the component
- Accessible via `this.props`

What if we want to modify data?

STATE

- Can be *modified*
- React knows to re-render the UI when the state is changed

Hooks

In a functional component we use a system called hooks to implement features such as tracking `state`-ful information.

The name hook comes from the idea that we are hooking into React's processing.

We will start with the simplest hook **in React,** `useState`**.**

`useState` is a React function that allows us to create a variable in our component that can change over time.

It comes from the standard React library.

Rules of hooks

1. Hooks should all begin with the word use and follow `camelCase` names.
2. Hooks must be called in the same order each time a component renders. The easiest way to guarantee this is to not place a use`XXXX` hook inside of a conditional, or have any "guard clauses" **before** the use of a hook method.

State changes lead to re-rendering

This is a key aspect of `state` in React.

Each time we change the `state` (using the method we are about to introduce) the `React` system detects this change and then **re-renders** our component with the new information.

Demo time!

Click Counter

The "Hello World" of interactive web applications!

A step-by-step approach to building a dynamic UI

1. Static Implementation
2. Make a state object containing data
3. Try manually changing the value in the state
4. Connect actions (later on, we'll add API interaction here)
5. Update state

Step 1 - Static implementation

- Render a static (hardcoded) version of what you want

```
export function Counter() {
  return (
    <div>
      <p>The count is 0</p>
      <button>Increment</button>
    </div>
  )
}
```

Step 2 - Introduce State

Add our first hook, known as `useState`.

Here is the code to create the state variables and display their value. We'll then break down this code line-by-line

```
export function Counter() {
  const valueAndSetMethod /* <- array */ = useState(0 /* initial state */)

  const counter = valueAndSetMethod[0]
  const setCounter = valueAndSetMethod[1]

  return (
    <div>
      <p>The counter is {counter}</p>
      <button>Count!</button>
    </div>
  )
}
```

```
export function Counter() {
  const valueAndSetMethod /* <- array */ = useState(0 /* initial state */)

  const counter = valueAndSetMethod[0]
  const setCounter = valueAndSetMethod[1]

  return (
    <div>
      <p>The counter is {counter}</p>
      <button>Count!</button>
    </div>
  )
}
```

- Declares we are going to use some state (e.g. useState)
- Sets initial value (e.g. 0)
- useState always returns an array with two entries

```
export function Counter() {
  const valueAndSetMethod /* <- array */ = useState(0 /* initial state */)

  const counter = valueAndSetMethod[0]
  const setCounter = valueAndSetMethod[1]

  return (
    <div>
      <p>The counter is {counter}</p>
      <button>Count!</button>
    </div>
  )
}
```

- The first value of the array is the current value
- The second value is a function used to change the value

```
export function Counter() {
  const valueAndSetMethod /* <- array */ = useState(0 /* initial state */)

  const counter = valueAndSetMethod[0]
  const setCounter = valueAndSetMethod[1]

  return (
    <div>
      <p>The counter is {counter}</p>
      <button>Count!</button>
    </div>
  )
}
```

- We can use the value to show the current State
- Later we'll see how to *change* the state

`useState` **rules**

`useState` has a few particular *rules* that we need to remember:

1. Value given to `useState` in parenthesis is used as the initial value only the first time the component's instance is rendered. Even if the component is rendered again due to a state change, the state's value isn't reset to the initial value.

2. `useState` always returns an *array* with exactly *two* elements. The **first** element is the *current value of the state* and the **second** element is *a function that can change the value of this state*

Simplify (using Destructuring Assignment)

```
export function Counter() {
  const [counter, setCounter] = useState(0)

  return (
    <div>
      <p>The counter is {counter}</p>
      <button>Count!</button>
    </div>
  )
}
```

Ah, so much more room for activities…

Step 3 - Try manually changing the value in the state.

- See that the UI changes when the state is modified

```
export function Counter() {
  const [counter, setCounter] = useState(42)

  return (
    <div>
      <p>The counter is {counter}</p>
      <button>Count!</button>
    </div>
  )
}
```

Step 4 - Connect actions

- Create a `handleXXXX` function to handle events.
- We will define this *INSIDE* our function. Whoa! Nested functions!

```
export function Counter() {
  const [counter, setCounter] = useState(42)

  function handleClickButton(event: React.MouseEvent) {
    event.preventDefault()

    console.log('Clicked!')
  }

  return (
    <div>
      <p>The counter is {counter}</p>
      <button>Count!</button>
    </div>
  )
}
```

Event handlers still receive event object

- Except now we **must** provide a specific type.
- Type depends on what *kind* of handler this is.

```
function handleClickButton(event: MouseEvent) {
  event.preventDefault()

  console.log('Clicked!')
}
```

NOTE: We are showing how to prevent the default behavior, not typically needed outside of links and form submits.

Connect the event

`Goodbye addEventListener`

```
<button onClick={handleClickButton}>Increment</button>
```

- We are associating the event, `onClick` with the function `handleClickButton`.
- The `onClick` is actually a property of the DOM element.
- We assign that property to the function itself.

Naming conventions

- `onXXXXX` or `handleXXXXX` named methods (e.g. `onClick`, onChange, `handleClick`, etc.)
- `_buttonClick` -- because the _ looks like a *"handle"* attached to the word `buttonClick`

Declaring handling functions *inline*

```
<button
  onClick={function (event) {
    event.preventDefault()

    console.log('Clicked!')
  }}
>
  Increment
</button>
```

Benefits

Don't need to declare a type! TypeScript will automatically make `event` a type of `React.MouseEvent<HTMLButtonElement, MouseEvent>`, an even more specific type than we used!

Downsides

Code that is more than one or two lines really clutters up the JSX

Can use arrow functions for very nice `one liners`

- We'll see these in a moment

Can connect a method to an event!

Time to update state (finally...)

Step 5 - Update state

- For our button, we want to:
  - Get the current counter
  - Increment it
  - Update the state

```
export function Counter() {
  const [counter, setCounter] = useState(42)

  function handleClickButton(event: MouseEvent) {
    event.preventDefault()

    // Increment
    const newCounter = counter + 1

    // Tell React there is a new value for the count
    setCount(newCounter)
  }

  return (
    <div>
      <p>The counter is {counter}</p>
      <button>Count!</button>
    </div>
  )
}
```

Warning! Warning!

NOTE: After `setCounter` *does not change* `counter` *right away. The value isn't changed until React gets a chance to update state AFTER our function is done.*
This often confuses new React developers. We'll see this again when we use more complex state

Simplify the code

```
function CounterWithName() {
  const [counter, setCounter] = useState(0)

  function handleButtonClick() {
    setCounter(counter + 1)
  }

  return (
    <div>
      <button onClick={handleButtonClick}>Count!</button>
    </div>
  )
}
```

- We are associating the event, `onClick` with the function `handleClickButton`.
- The `onClick` is actually a property of the DOM element.
- We assign that property to the function itself.

Inline function

```
function CounterWithName() {
  const [counter, setCounter] = useState(0)

  function handleButtonClick() {
    setCounter(counter + 1)
  }

  return (
    <div>
      <button
        onClick={function () {
          setCounter(counter + 1)
        }}
      >
        Count!
      </button>
    </div>
  )
}
```

Arrow function

```
function CounterWithName() {
  const [counter, setCounter] = useState(0)

  function handleButtonClick() {
    setCounter(counter + 1)
  }

  return (
    <div>
      <button onClick={() => setCounter(counter + 1)}>Count!</button>
    </div>
  )
}
```

Adding more state

What if we also wanted to keep track of a person's name on the counter?

With hooks, we will make two **independent** states that each track a single piece of information.

```
function CounterWithName() {
  const [counter, setCounter] = useState(0)
  const [name, setName] = useState('Susan')

  function handleChangeInput(event: React.ChangeEvent<HTMLInputElement>) {
    setName(event.target.value)
  }

  return (
    <div>
      <p>
        Hi there {name} The counter is {counter}
      </p>
      <button onClick={() => setCounter(counter + 1)}>Count!</button>
      <p>
        <input type="text" value={name} onChange={handleChangeInput} />
      </p>
    </div>
  )
}
```

handleChangeInput

- `event` is a `React.ChangeEvent` on an `HTMLInputElement` element
- For our button, we want to:
    - Get the current count
    - Increment it
    - Update the state

Inline function

```
function CounterWithName() {
  const [counter, setCounter] = useState(0)
  const [name, setName] = useState('Susan')

  return (
    <div>
      <p>
        Hi there {name} The counter is {counter}
      </p>
      <button onClick={() => setCounter(counter + 1)}>Count!</button>
      <p>
        <input
          type="text"
          value={name}
          onChange={function (event) {
            setName(event.target.value)
          }}
        />
      </p>
    </div>
  )
}
```

Arrow function

```
function CounterWithName() {
  const [counter, setCounter] = useState(0)
  const [name, setName] = useState('Susan')

  return (
    <div>
      <p>
        Hi there {name} The counter is {counter}
      </p>
      <button onClick={() => setCounter(counter + 1)}>Count!</button>
      <p>
        <input
          type="text"
          value={name}
          onChange={event => setName(event.target.value)}
        />
      </p>
    </div>
  )
}
```

A note on types

You may have noticed that when declaring these variables we did **not** have to specify a type:

```
const [counter, setCounter] = useState(0)
```

If we did not provide an initial state, React would **not** be able to infer the type. Here is an example of that type of `useState`

```
const [price, setPrice] = useState()
```

- TypeScript will set a type of `undefined` to `price`.
- When we try to `setPrice(42)` (or any other number) we'll receive a TypeScript error that we cannot assign number to `undefined`.

In the case where we do **not** provide an initial value to `useState` we *should* provide a type.

```
const [price, setPrice] = useState<number>()
```

`price` has a type of `undefined | number`.

Always set default state value

This is the reason that we **strongly** recommend always using an initial value for all of your `useState` hooks.

If you *cannot* set an initial value you must consider the impact that allowing an `undefined` value in a state variable will have.

Steps:

- Step 1 - Static implementation
- Step 2 - Make a state object containing data
- Step 3 - Try manually changing the value in the state.
- Step 4 - Connect actions
- Step 5 - Update state