React Hooks

History

- Stateless functional components

```
class HelloWorld extends React.Component {
  render() {
    return <div>Hello, World!</div>
  }
}

// <HelloWorld />
```

stateless

- Lacks `this.state` and `this.setState`

Code ... Code everywhere ...

- A component can also be a `function` that returns `JSX`

```
function HelloWorld() {
  return <div>Hello, World!</div>
}

// <HelloWorld />
```

Easier to read and understand

- Less "ceremony"

Yet, how do we access `props` **if there is no** `this` **for** `this.props`**?**

Function receives props as an argument

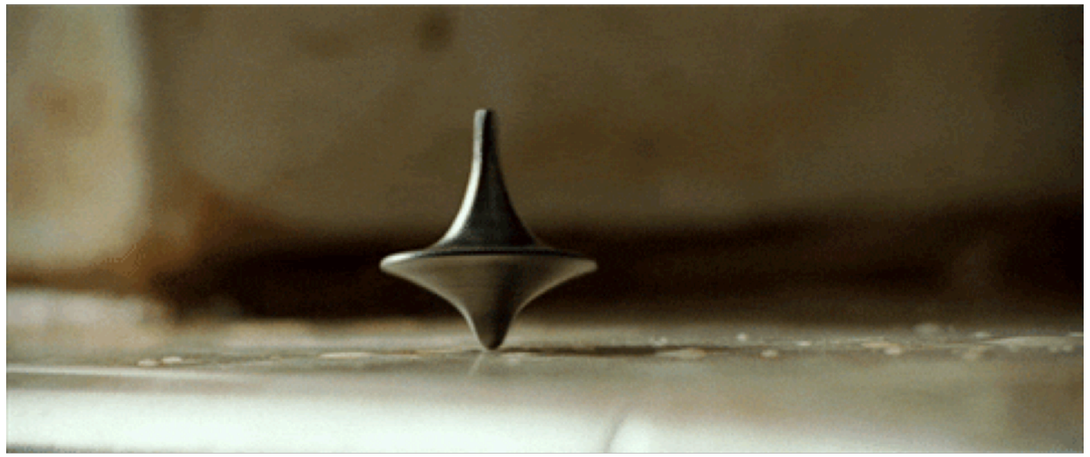
```
function HelloWorld(props) {
  return <div>Hello, {props.name}!</div>
}

// <HelloWorld name="Sandy" />
```

Ok, ok. But what about event handlers?

```
function handleClickOnDiv(event) {
  console.log('You clicked on the div!')
}

function HelloWorld(props) {
  return <div onClick={handleClickOnDiv}>Hello, {props.name}!</div>
}

// <HelloWorld name="Sandy" />
```

We can also put functions inside functions

```
function HelloWorld(props) {
  function handleClickOnDiv(event) {
    console.log('You clicked on the div!')
  }

  return <div onClick={handleClickOnDiv}>Hello, {props.name}!</div>
}

// <HelloWorld name="Sandy" />
```



Or use arrow functions ...

```
function HelloWorld(props) {
  const handleClickOnDiv = event => {
    console.log('You clicked on the div!')
  }

  return <div onClick={handleClickOnDiv}>Hello, {props.name}!</div>
}

// <HelloWorld name="Sandy" />
```

Ok, how do these help?

- Separate concerns.
  - Stateful classes for managing state
  - Functional components for rendering

```
class ToDoListContainer extends React.Component {
  state = {
    list: [],
  }

  addItem = item => {
    this.setState({ list: [...this.state.list, item] })
  }

  // code to remove items, sort items, mark complete, etc.

  render() {
    return (
      <ToDoList
        list={list}
        addItem={this.addItem}
        deleteItem={this.deleteItem}
      />
    )
  }
}
```

```
function ToDoList(props) {
  return (
    <ul>
      <li onClick={props.addItem}>Add item!</li>
      {list.map(item => (
        <li key={item.id} onClick={props.deleteItem}>
          {item}
        </li>
      ))}
    </ul>
  )
}
```

ToDoList is still functional and Stateless

The `ToDoListContainer` does not handle any rendering.
Its purpose is to manage state.

Great, but two different styles?

- Use `classes` for state
- Use `function` for stateless

React 16.8.0
Solve these challenges

- Hard to reuse stateful logic between components
- Complex components are hard to understand. (e.g. componentDidMount, componentWillMount, componentWillReceiveProps)
- See React Lifecycle
- Classes are new and didn't *really* fit the JavaScript style
- `function` is easier to write

Enter Hooks

Hooks

Allow React developers to do everything a traditional `class` *based component could do, but with only using* `function` *style definitions.*

The first hook

`useState`

- `useState` is a method provided by `React`
- Meant to manage related state data (sometimes just a single number, sometimes an array or object)
- Called with the value the state should be the *first* time the component renders

Returning to our counter

```
const counterValueAndSetMethod = useState(0)
```

`useState` rules and behavior

1. The value in parenthesis is the initial value *only*
2. Returns an array of two values (we will see what these are in a moment)
3. Does a *full* replacement of the state. Unlike `this.setState` that can do *partial* updates

What is in `counterValueAndSetMethod`?

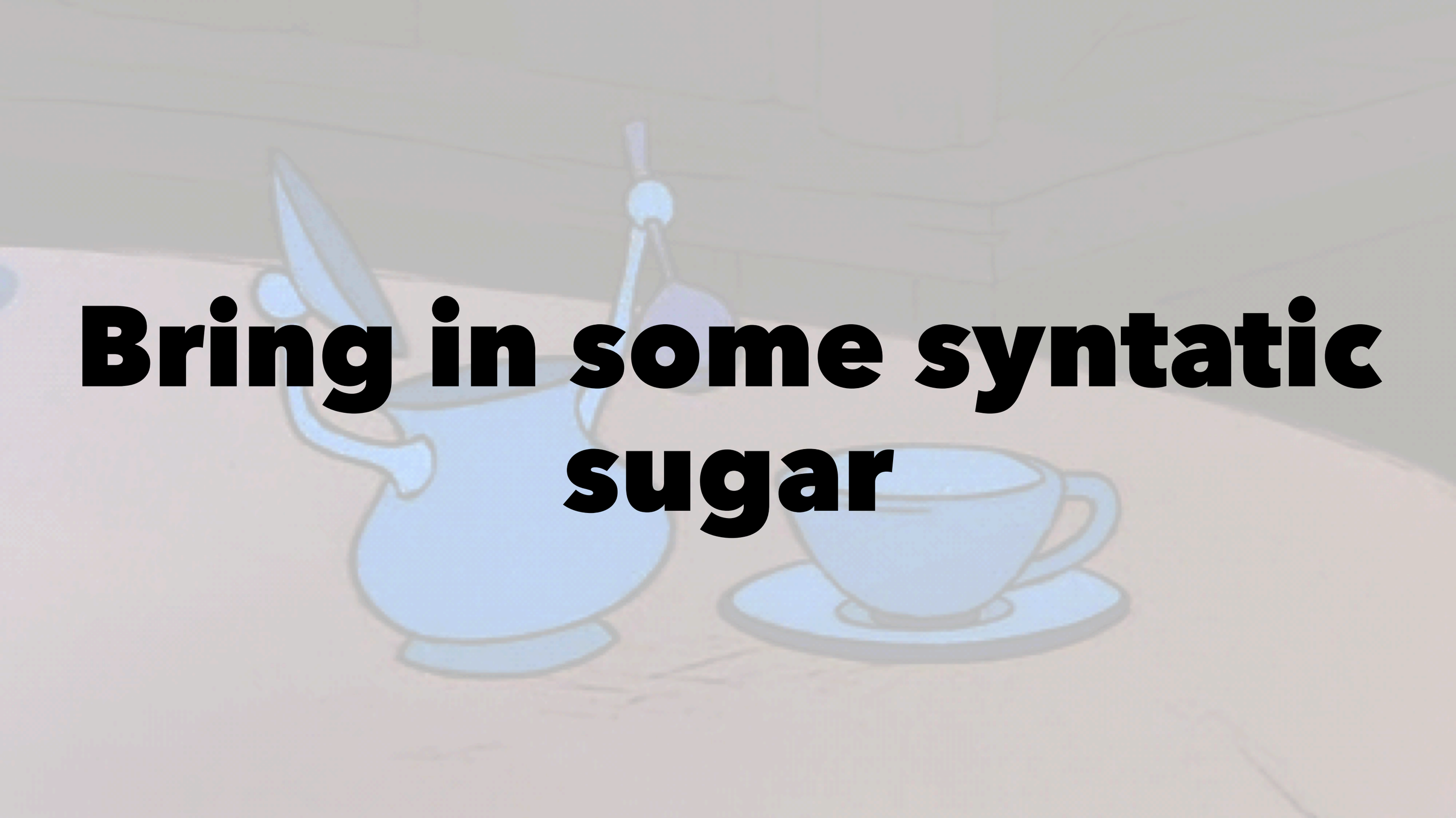
It is an array with two entries.

- The first is the current value of the state
- The second is the *function* that can change the state value

```
const counterValueAndSetMethod = useState(0)

const counter = counterValueAndSetMethod[0]
const setCounter = counterValueAndSetMethod[1]
```

```
const counter = counterValueAndSetMethod[0]
const setCounter = counterValueAndSetMethod[1]
```

Make two local variables to store the **current value** of our state, which we call **counter** and the **method that updates the counter** as **setCounter**



Bring in some syntatic sugar

Destructuring arrays!

```
const names = ['Susan', 'Bob']

const first = names[0]
const other = names[1]
```

Better:

```
const names = ['Susan', 'Bob']

const [first, other] = names
```

Apply to `useState`

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

See this article for more details on how and why this syntax works.

Apply this to our Counter

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

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

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

What about a second bit of state?

- Keep track of a person's name on the counter.
- Traditionally we would define a state like this:

```
class CounterWithName extends React.Component {
  state = {
    counter: 0,
    name: '',
  }
}
```

Hooks allows us to have *multiple* independent states

Separating these pieces of state has a few benefits:

1. It is easier to remove one part of the state since it has its own variable and state changing function.
2. We can more easily tell where in the code a piece of state or a state changing function is used.
3. We don't have to worry about using `this.state.name` or `this.state.counter`, just `counter` and `name`.

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

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

  function onChangeInput(event) {
    setName(event.target.value)
  }

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

Not everything is perfect...

- Components can end up with *many* useState
- Have to keep track of multiple variables
- Also, how do we handle the *when I first mount/render please fetch some data*

React comes with other hooks "out of the box"

- We'll look at some of these next.
  - `useEffect`
  - `useReducer`
  - `useContext`

The React team has a nice example of hooks in their guide to hooks