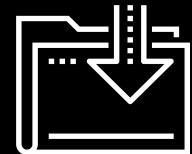


# Introduction to React Hooks

Skills Bootcamp in Front-End Web Development

Lesson 14.1





# WELCOME

# Learning Objectives

---

By the end of class, you will be able to:



Articulate the term “effect” in the broader sense of programming.



Utilize React’s most common built-in Hooks: `useState` and `useEffect`.



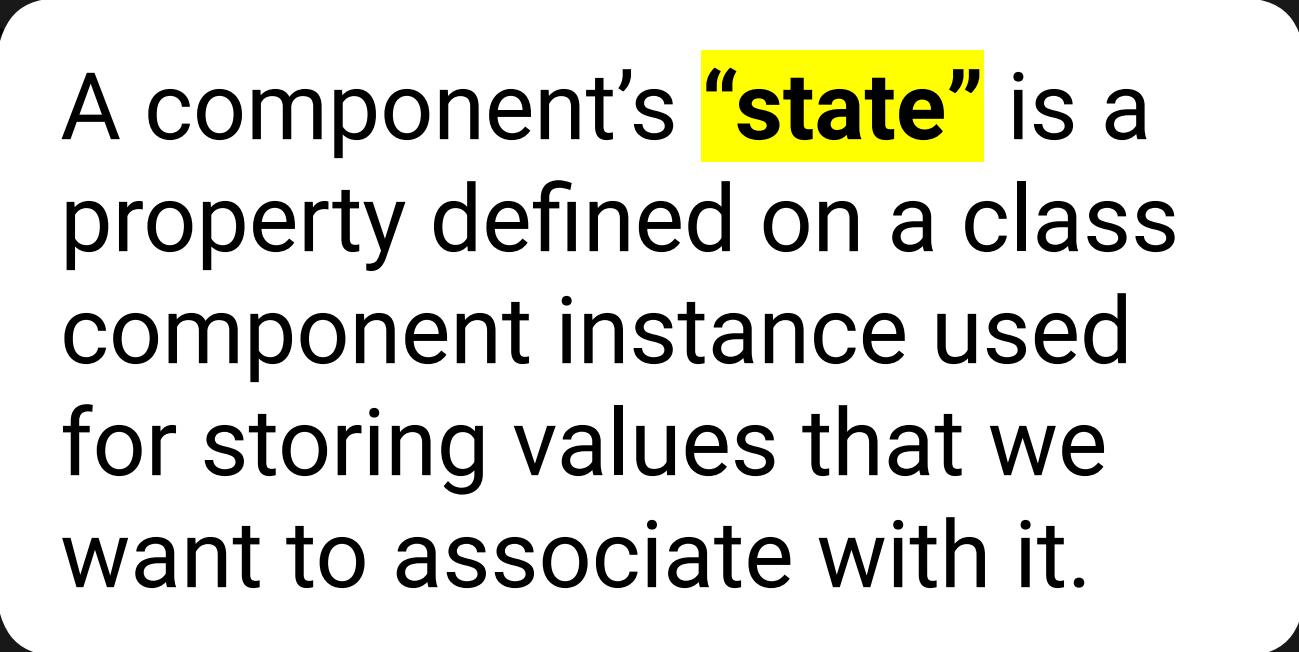
Transform a React application that manages state with a class component into an application that uses functional components with React Hooks.



Create a custom reusable Hook that follows the two rules of Hooks.

# State and Hooks

# React State Management



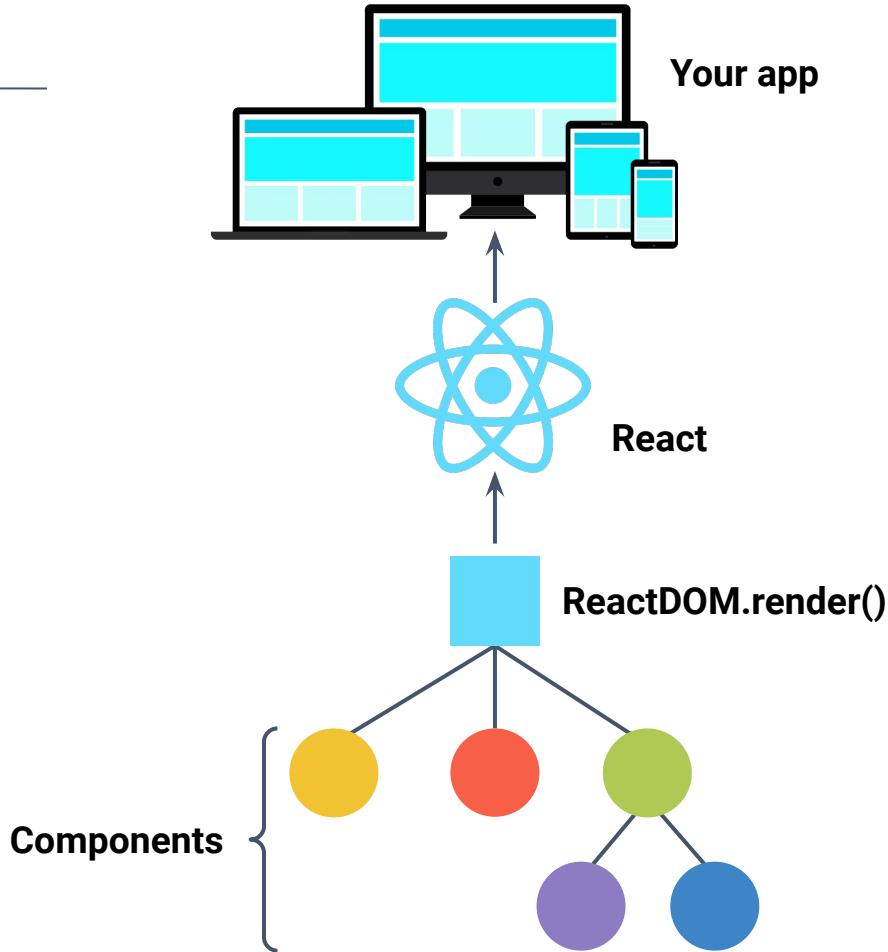
A component's “**state**” is a property defined on a class component instance used for storing values that we want to associate with it.

# A Component's “State”

This property is recognized by React and can be used to embed data inside of a component's UI, which we want to update over time.

Whenever a component's state is updated, its render method is fired along with the render methods of all of its children.

This updates the application's UI to display the new data without having to refresh the browser.



# React State Management

---



State is the data that are currently in memory—including numbers, strings, objects, etc.

At rest, state doesn't change.

State changes through events—such as UI interactions, application starts and stops, and background requests.

# React State Management

---

Most React bugs are the result of bad state. More accurately, they're the result of bad assumptions about state. We assume that some bit of data could never happen or that some sequence of state changes is impossible.

**Then we write code with those assumptions baked in.**

Even if our code is perfect, troubles can arise.

If our code isn't the only code managing state, another process or step can modify state into something that our code didn't anticipate.

**That's a bug.**



# React State Management

---

React state management protects us from a lot of that.

Science hasn't figured out how to create truly bug-free code, but React state management gets us a little closer.



# React's State Flow

At its core, React is conceptually simple.



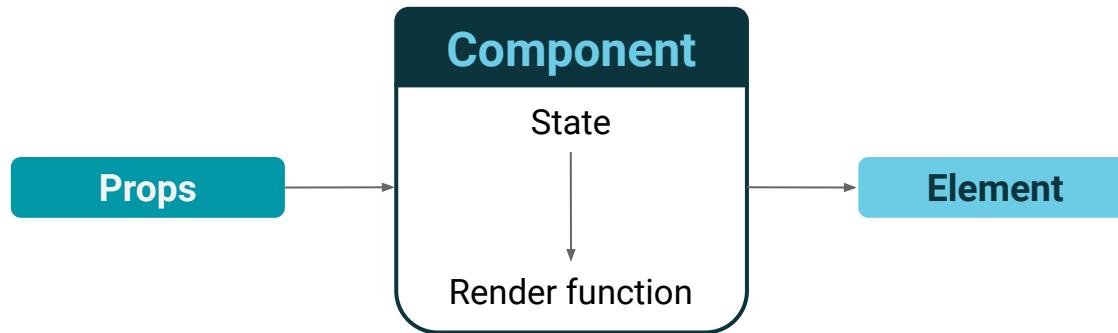
Component instances track their own private state.



Components may cautiously share their state with a child via props.



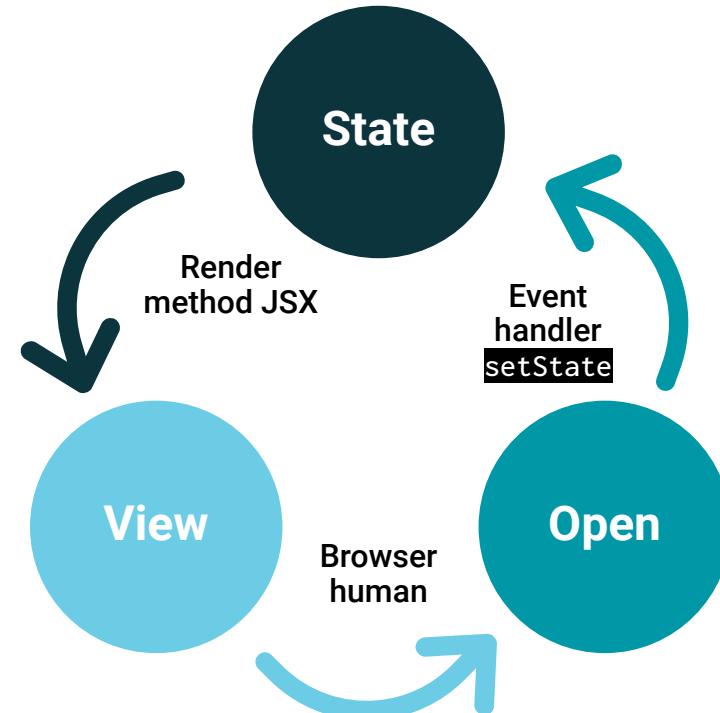
Any change to state (and by proxy, props) triggers a re-render.



# React's State Flow

This happens over and over in the following process:

- 01 State drives UI rendering.  
React reacts to state.
- 02 Events modify state.
- 03 Return to Step 1.  
But the devil is in the details.





Managing state can be difficult  
because there is no  
one-size-fits-all solution.  
But there is another way.

# Comparing Ways to Manage State

---

	Class Components with <code>setState()</code>	Functional Components with <code>useState()</code>
Advantages 	Component and children will re-render with up-to-date data.	<ul style="list-style-type: none"><li>It's easier to read and debug, and there's no need to use <code>this</code>.</li><li>You have access to Hooks.</li></ul>
Disadvantages 	<ul style="list-style-type: none"><li>Updating state from nested components can be difficult.</li><li>Since state only flows one way, all components that need access to the state must be children of the same stateful component.</li></ul>	<ul style="list-style-type: none"><li>Needs to use other Hooks to manage complex levels of state.</li><li>Not supported by older codebases, which will still need to use class components for state.</li></ul>

# Sharing State



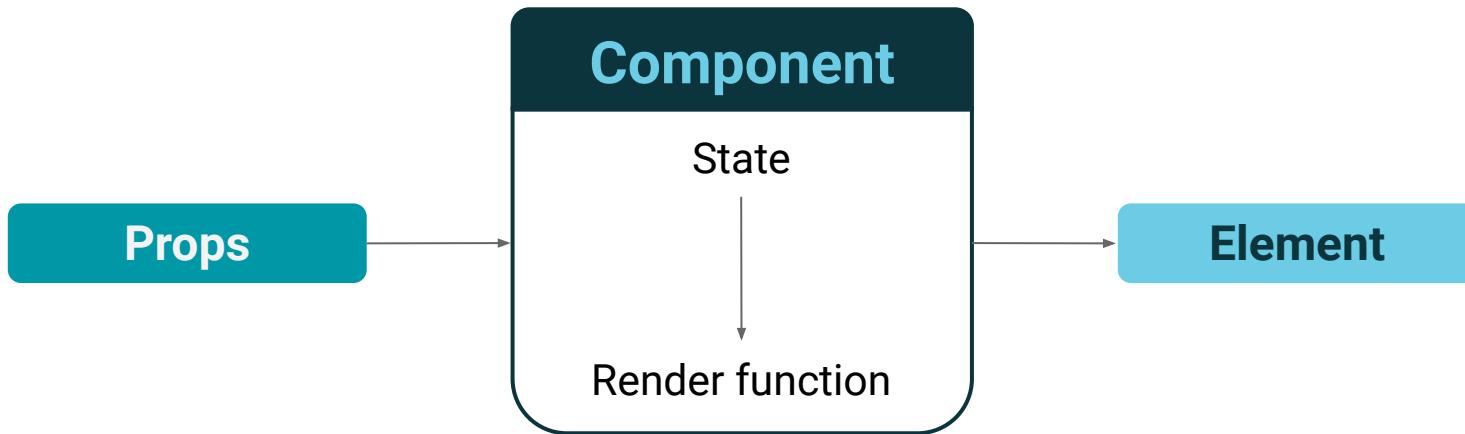
Component design and class  
design are the same thing but in a  
different layer of the application.

# Sharing State

---

Like Java classes, components need a way to share messages, which is just a different way of saying that components need a way to share changes in state.

We've already seen an example of parent-to-child state sharing through props.



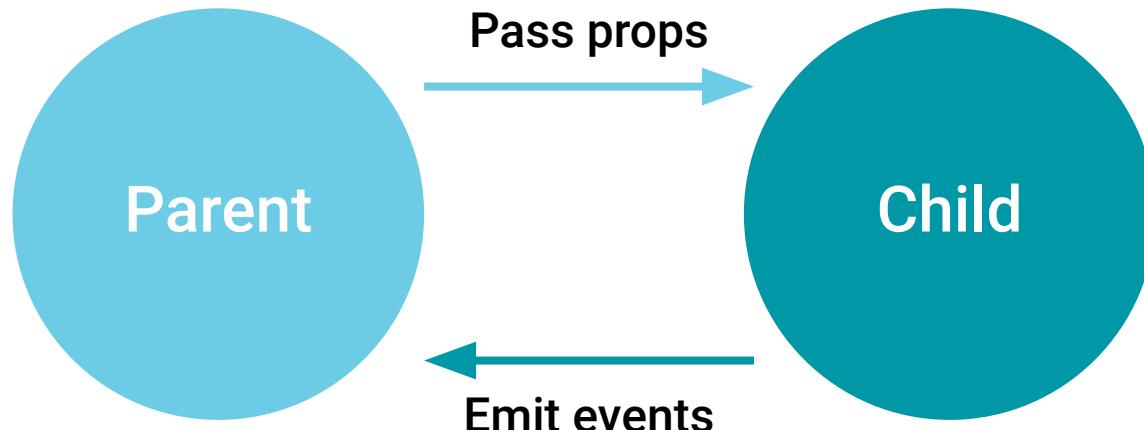


Children communicate with  
their parent via **events**.

# Sharing State

---

When components communicate in both directions, they can solve interesting problems.





How have you managed  
React state so far?



What is one of the biggest issues  
of sharing state between class  
and functional components?

# **Class Components vs. Functional Components**

# Class Components vs. Functional Components

---

Class components paired with the `setState` method is the most common way to manage state.



## Benefits

- Setting state will force the component to re-render.
- This ensures that the component and all of its children will be up to date with the latest state.



## Drawbacks

- Since state only flows in one direction, all components that need to use the state must be children of the stateful component.
- Updating state from deep within the component hierarchy is often difficult.
- You are forced to use the keyword `this` and use a class component.

# Class Components vs Functional Components

---

The `useState` Hook is another tool that can be used to manage state through functional components.



## Benefits

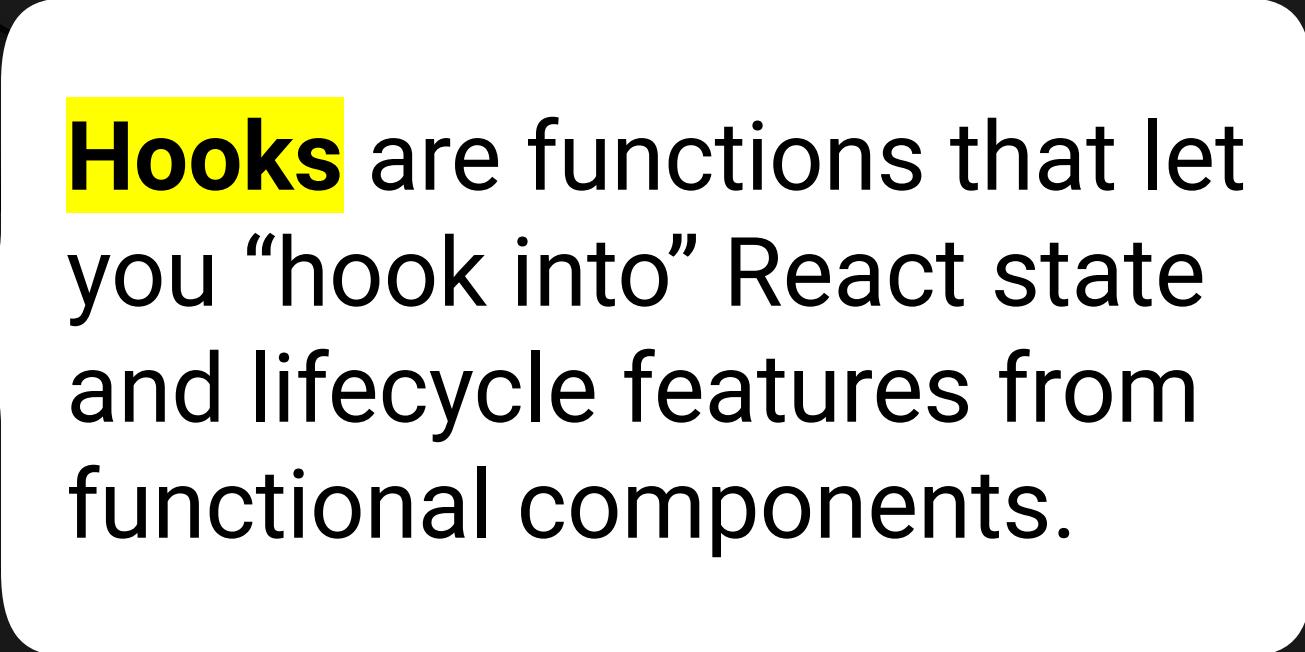
- There's no need to receive state from props as a child component.
  - It's easier to read and debug due to less code.
  - There is no need to use `this`.
  - As of React 16.8, Facebook recommends using functional components whenever possible.
- 



## Drawbacks

- Will need to use other *Hooks* to manage complex levels of state.

# React Hooks



**Hooks** are functions that let you “hook into” React state and lifecycle features from functional components.



React has no plans to deprecate class components, but Hooks can do everything a class component can do and more.

# React Hooks

---

In this lesson, we will cover two Hooks:

useState

Allows you to use state in a functional component.

useEffect

Replaces lifecycle methods like `componentDidMount` and `componentDidUpdate`.

An **effect**, also commonly referred to as a *side effect*, is a term used to describe the result of affecting the “outside world.”

# React Hooks

---

This includes, but is not limited to:



Data  
fetching

Subscribing  
to events

Changes to  
the DOM

# React Hooks

---



Functional components use **Hooks** for state management.



A Hook is a function.



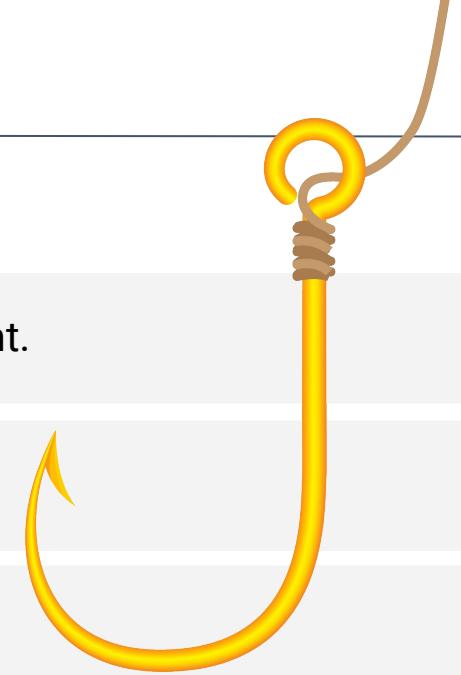
Hook names always start with the `use` prefix.



Hooks allow us to tap into larger React plumbing. We “hook into” a React process.



The `useState` Hook manages state.



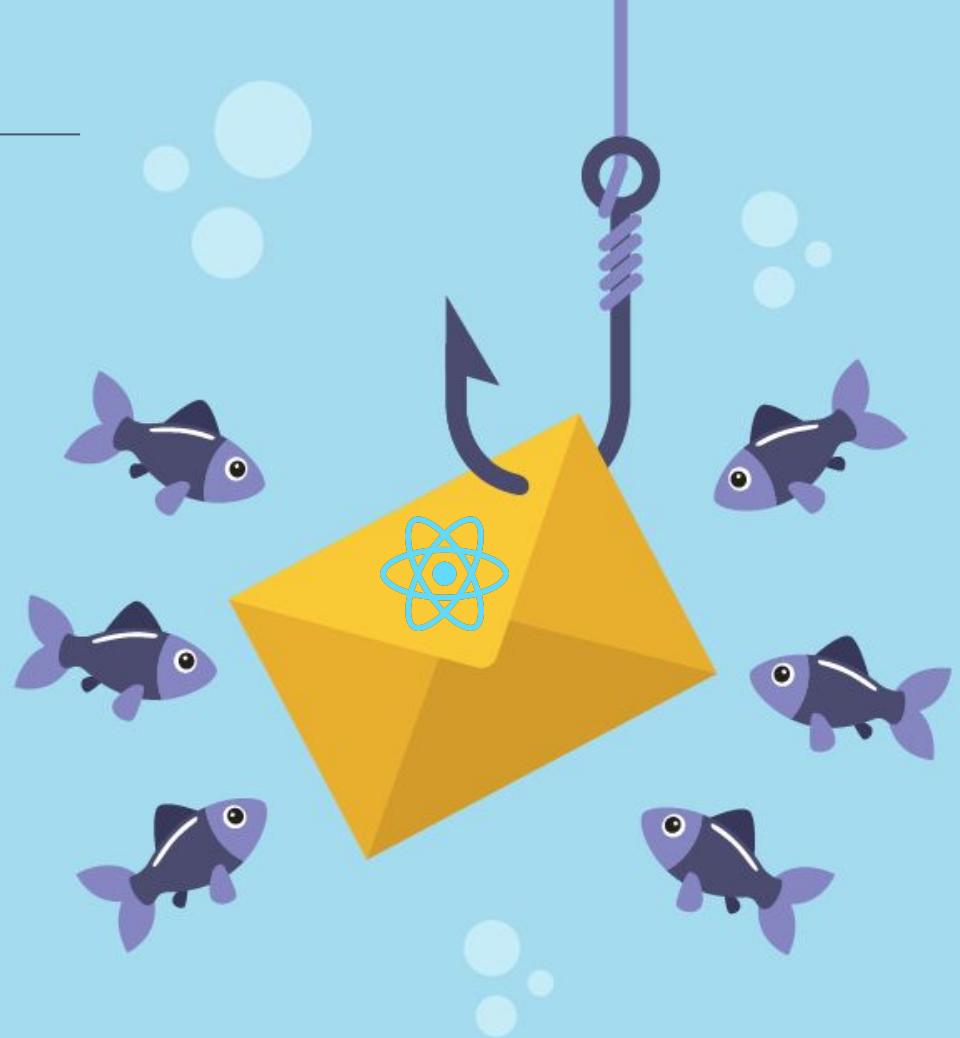
# React Hooks

---

We will be covering the built-in Hooks `useState` and `useEffect`, but there are many more Hooks.

The strong developer community around React has created a plethora of different [custom Hooks](#) that you can plug into your applications.

This makes it so that all stateful logic is easy to find for the developer.



# React Hook Rules

---

There are two rules of Hooks that must be complied with:

01

## **Only call Hooks from top-level components.**

- This means that you should never call Hooks from within loops, conditionals, or nested functions.
- It is important that Hooks are always called in the same order, like component lifecycle methods.
- It is also what makes it possible for React to store the state of Hooks when using `useState` or `useEffect`.

02

## **Hooks may only be called from React components.**

- Never call a Hook from a regular JavaScript function.
- This makes it so that all stateful logic is easy to find for the developer.



If we don't follow these rules, our components will not work properly.  
These rules apply to all Hooks,  
not just to `useState`.



# Instructor Demonstration

---

## useState



## Activity: useState

In this activity, you will practice using the `useState` Hook in React.

Suggested Time:

---



Time's Up! Let's Review.



Why might using `value` with  
`onChange` be a bad idea?

# Review: useState Activity

It all comes down to controlled input vs. uncontrolled input.

01

## Controlled input

Accepts its current value as a prop and has a callback that allows you to change its value.

Whenever `onChange` updates its value, it's essentially the input controlling itself.



The combination of the two would result in the component going from uncontrolled to controlled, which is considered **bad practice**.

02

## Uncontrolled input

Is an input that gets its value from somewhere else.

In our case, if `value={username}`, then the input would be getting its value from the state.



# Instructor Demonstration

---

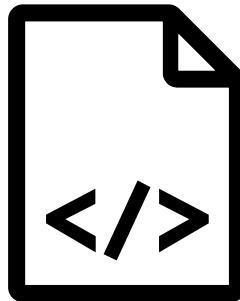
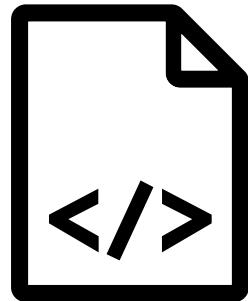
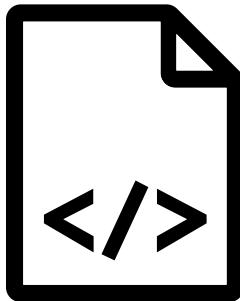
## useEffect

# useEffect Demo

---

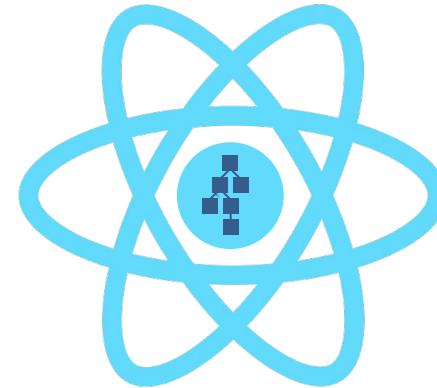
In the “outside world”:

Effects, often referred to as side effects, are bits of code responsible for the modification of state.



In React:

Effects are most commonly used for data fetching and manually changing the DOM.



# useEffect Demo

---

**useEffect** is a method that takes two arguments.

01

First argument

A function that you want to run after the component mounts.

02

Second argument

An array of dependencies, commonly referred to as “deps.”

Every time the component re-renders, the **useEffect** Hook will check to see if any of the values in its dependency array have changed.

- If they have, then the function that you supplied as the first argument of **useEffect** will run again.
- If not, React will skip the effect for that particular render.



## Activity: **useEffect**

In this activity, you will practice using the `useState` and `useEffect` Hooks in React by transforming a stateful class component into a functional component with React Hooks.

Suggested Time:

---



Time's Up! Let's Review.



What was the point of this?



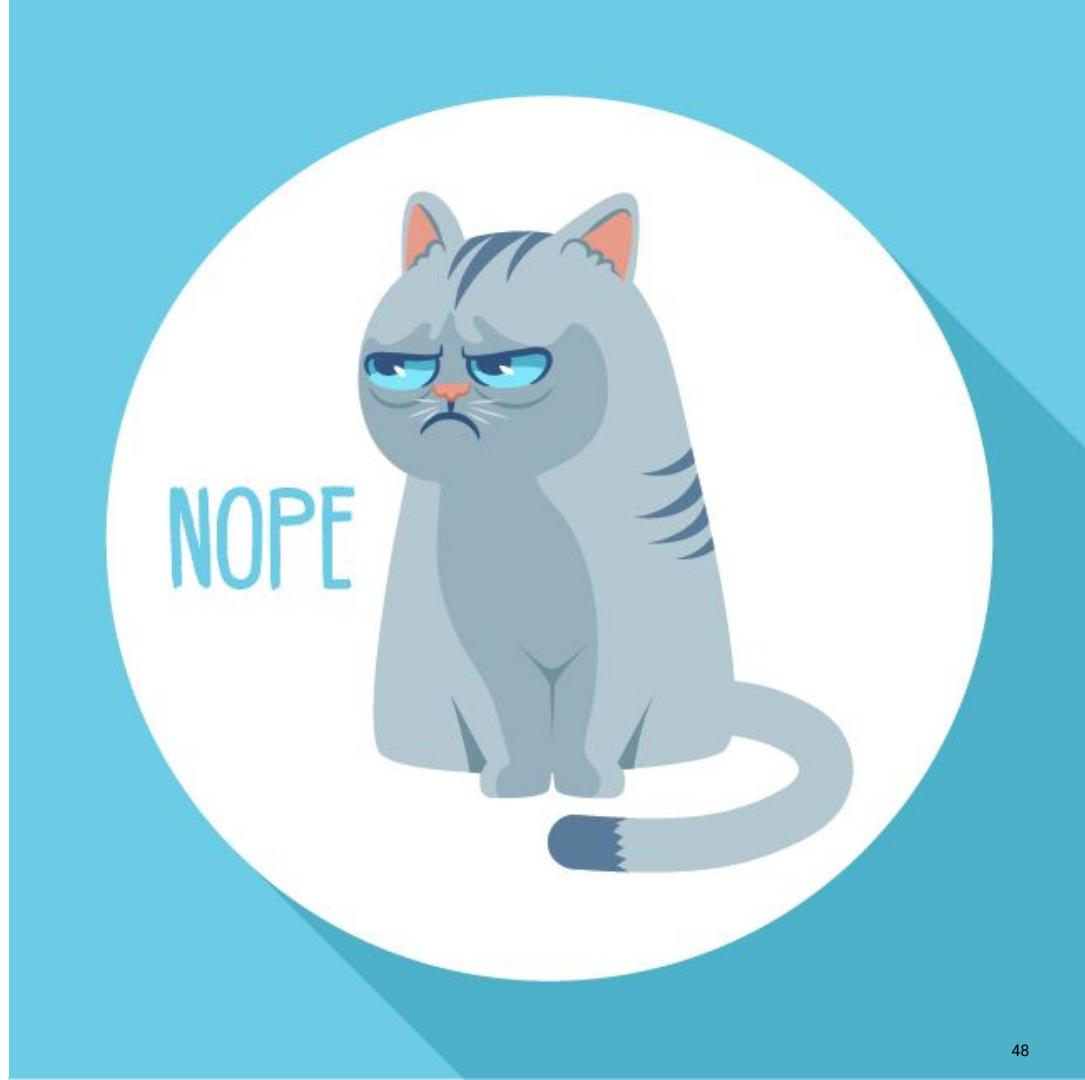
## Writing functional components is much cleaner.

Using Hooks allows you to write fewer lines of code and manage your state in a less complex way. (No `this` keyword required!)



If we use Hooks, can state be used by other components?

The state used within  
a single component  
cannot be used by  
different components.





Can you think of a concept that would allow us to share state across components?



For now, we can use props,  
but in the future, you will  
learn a better way.





# Instructor Demonstration

---

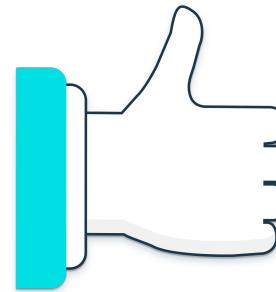
## Custom Hooks

# Custom Hooks

---



**Only** call Hooks from top-level components.



**Only** call Hooks from React components.



**Never** call Hooks from within loops, conditionals, or nested functions.

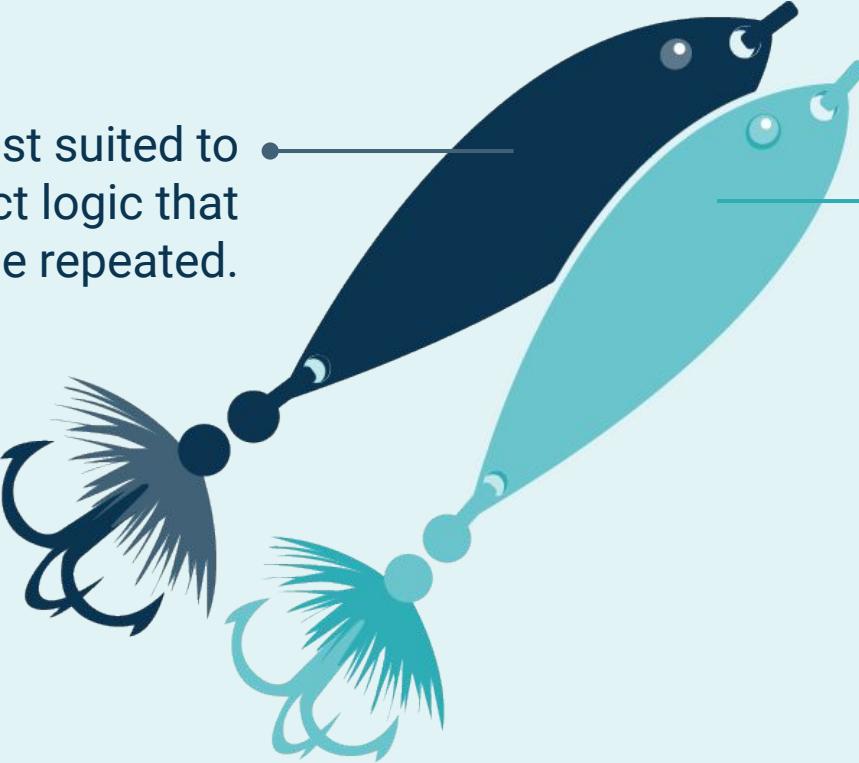


**Never** call a Hook from a regular JavaScript function.

# Custom Hooks Can Be Practically Anything!

---

Custom Hooks are:



Best suited to extract logic that may be repeated.

A great way to keep your React functions pure.

# Custom Hooks

---

As with the two  
rules of Hooks...

Custom Hooks must start with the word **use** so that React can ensure that your code is adhering to the two rules of Hooks.

As with **useState**  
and **useEffect**,

Different components that use the same custom Hook **do not** share the same state.



## Activity: Custom Hooks

In this activity, you will practice using custom Hooks by creating a `useDebounce` Hook that will delay the invoking of a function for a given number of milliseconds.

Suggested Time:

---



Time's Up! Let's Review.



## Activity: Third-Party Hooks

In this activity, you will practice using third-party Hooks. Specifically, you will be creating a survey form using the **react-hanger** package on npm.

Suggested Time:

---

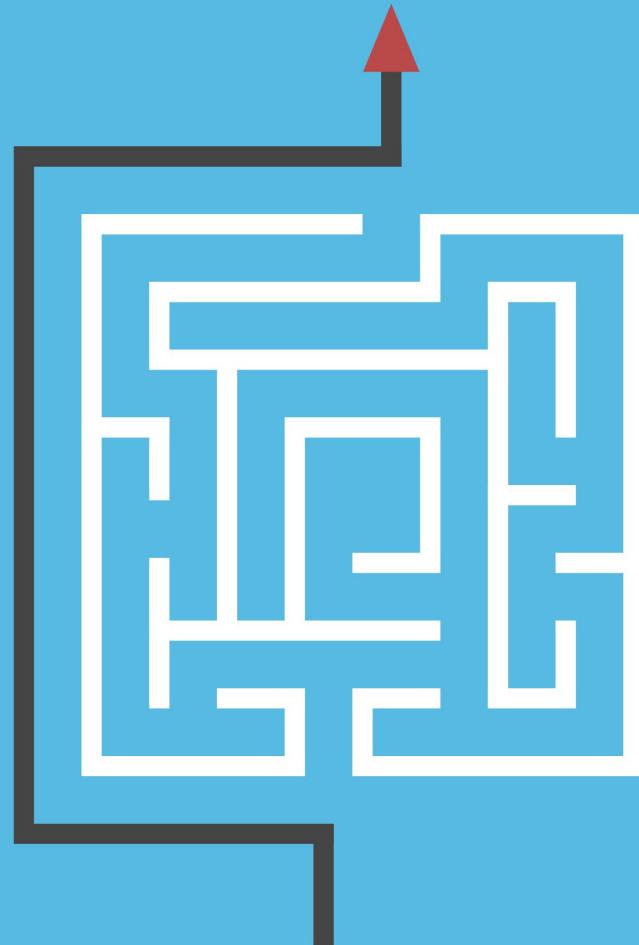
**react-hanger** is one of many  
custom Hooks packages on



**GitHub**

There are many ways to satisfy the requirements of this application.

It is recommended that you attempt the **most straightforward solution** first, then refactor your working app.





Time's Up! Let's Review.

# Next Class

Do your best to go through the following sections of the React documentation before the next class:

## React Hooks Docs

The screenshot shows the React Hooks documentation homepage. The main heading is "Introducing Hooks". Below it, a paragraph explains that Hooks are a new addition in React 16.8, allowing state and other features without writing a class. A code snippet demonstrates how to use useState and useEffect. On the right, there's a sidebar with navigation links for Installation, Main Concepts, Advanced Guides, API Reference, and Hooks (which is currently selected).

## A Complete Guide to useEffect()

The screenshot shows a blog post titled "A Complete Guide to useEffect()". It includes a "Overreacted" badge, a timestamp of March 9, 2019, and a reading time of 49 min. read. Below the title, it says "Translated by readers into: Español • Português do Brasil • 日本語 • 简体中文 • 繁體中文 • 한국어". The main content discusses common mistakes with Hooks and how to avoid them.

## List of React Hooks

The screenshot shows a GitHub repository named "Collection of React Hooks". It has a green header with the repository name. Below it, a message says "You can add your hooks by opening a pull-request at https://github.com/nikgraf/react-hooks.". There's a search bar labeled "filter by name" and a category navigation bar with "State Management (80)", "Sensor (29)", "UI (26)", "Web API (26)", and "Network (25)". A note says "Found 440 entries". Below, a specific commit is shown for "use-abortable-stream-fetch" by "marconi/use-abortable-stream-fetch".

*The  
End*