React Lecture - 10

States

Introduction

Components often need to change what's on the screen as a result of an interaction.

Typing into the form should update the input field

Clicking "next" on an image carousel should change which image is displayed

Clicking "buy" should put a product in the shopping cart.

Why normal variables are not enough ?

1. **Local variables don't persist between renders**. When React renders this component a second time, it renders it from scratch—it doesn't consider any changes to the local variables.

2. **Changes to local variables won't trigger renders.** React doesn't realize it needs to render the component again with the new data.

Why normal variables are not enough ?

To update a component with new data, two things need to happen:

1. **Retain** the data between renders.
2. **Trigger** React to render the component with new data (re-rendering).

The useState Hook provides those two things:

1. A **state variable** to retain the data between renders.
2. A **state setter function** to update the variable and trigger React to render the component again.

Adding a state variable

To add a state variable, import useState from React at the top of the file:

```
import { useState } from 'react';
```

After importing, use the useState hook to create a state variable

```
const [index, setIndex] = useState(0);
```

The useState hook takes one argument which is the default value of the state variable.

useState hook **returns an array** with two elements. The first element is the state variable and the second element is the state setter function.

Adding a state variable

The [and] syntax here is called **array destructuring** and it lets you read values from an array. The array returned by useState always has exactly two items.

Anatomy of useState

When you call useState, you are telling React that you want this component to remember something:

```
const [index, setIndex] = useState(0);
```

In this case, you want React to remember index.

The convention is to name this pair like const **[something, setSomething]**.

You could name it anything you like, but conventions make things easier to understand across projects.

Anatomy of useState

Every time your component renders, useState gives you an array containing two values:

1. The **state variable** (index) with the value you stored.
2. The **state setter function** (setIndex) which can update the state variable and trigger React to render the component again.

Anatomy of useState

Here's how that happens in action:

1. **Your component renders the first time**. Because you passed 0 to useState as the initial value for index, it will return [0, setIndex]. React remembers 0 is the latest state value.
2. **You update the state.** When a user clicks the button, it calls setIndex(index + 1). index is 0, so it's setIndex(1). This tells React to remember index is 1 now and triggers another render.
3. **Your component's second render.** React still sees useState(0), but because React remembers that you set index to 1, it returns [1, setIndex] instead.
4. And so on!

State is isolated and private

State is local to a component instance on the screen. In other words, **if you render the same component twice, each copy will have completely isolated state!** Changing one of them will not affect the other.

This is what makes state different from regular variables that you might declare at the top of your module. State is not tied to a particular function call or a place in the code, but it's "local" to the specific place on the screen.

Trigger Render Commit

Trigger Render Commit

Before your components are displayed on screen, they must be rendered by React. Understanding the steps in this process will help you think about how your code executes and explain its behavior.

This process of requesting and serving UI has three steps:

1. **Triggering** a render
2. **Rendering** the component
3. **Committing** to the DOM

Step 1: Trigger a render

There are two reasons for a component to render:

1. It's the component's initial render.
2. The component's (or one of its ancestors') state has been updated.

Step 1: Trigger a render - Initial Render

When your app starts, you need to trigger the initial render. Frameworks and sandboxes sometimes hide this code, but it’s done by calling createRoot with the target DOM node, and then calling its render method with your component

Step 1: Trigger a render - Re renders when state updates

Once the component has been initially rendered, you can trigger further renders by **updating its state with the set function**. Updating your component's state automatically queues a render.

Step 2: React renders your components

After you trigger a render, React calls your components to figure out what to display on screen. **“Rendering” is React calling your components.**

1. **On initial render**, React will call the root component.
2. **For subsequent renders,** React will call the function component whose state update triggered the render.

Step 3: React commit Changes to DOM

After rendering (calling) your components, React will modify the DOM.

1. **For the initial render**, React will use the appendChild() DOM API to put all the DOM nodes it has created on screen.
2. **For re-renders,** React will apply the minimal necessary operations (calculated while rendering!) to make the DOM match the latest rendering output.

React only changes the DOM nodes if there's a difference between renders