CSE 331
Software Design & Implementation

Spring 2020
“Section 8” – HW8: React Canvas + Common Bugs
Agenda

• Overview of HW8 – “Connect the Dots”

• Canvas HTML object
  – how to use from React

• Common React bugs & how to fix them
HW8 Overview

• Starter code has (most of) the pieces, but not much functionality.
  – Lots of hard-coded values, placeholders (console.log instead of actually doing stuff), etc..
• Your job: "wire all the pieces together"
  – Accept user input
  – Process/parse the data
  – Error check – users do weird stuff, make sure you can't crash
  – Move data between components as necessary
  – Add the actual functionality in response to user input.
• Structure:
  – Top-level <App> component, with three child components.
Running a React App

\texttt{npm}: Similar to gradle, but we need to install manually the first time.

In the terminal, change directory until you're in the same place as the "\texttt{package.json}" file for the project you want to run.

To Install (first time): \texttt{npm install}
To Run (every time): \texttt{npm start}

Once started, you can edit and save files and the page will automatically reload – no need to restart. Use Control-C to shut down when you're done developing.
CANVAS
Example 2: Drawing on a Canvas

- `<canvas>` tag: creates a blank drawing surface that you can “draw” on with JS
  - Create lines, shapes, draw images.
  - Has `width` and `height` attributes to determine the size of the drawing surface.
- We’re using `<canvas>` in HW8 and HW9 to draw lines/paths on top of images (like a map of campus!)
- Javascript is going to need some kind of Canvas object in order to call functions and draw pictures.
  - How do we get this object?
Modifying HTML with JS

- JS exists to allow webpages (meaning the HTML inside them) to change dynamically. So JS has to have a way to access/change the HTML tags.
- Implementation: Every HTML element has an associated JS object that the browser maintains.
  - Can get a reference in JS by using the “id” attribute.
  - Every tag can have an ID - value is a string that uniquely identifies an element.

HTML:

```html
<p id="thePar">Hello, World!</p>
```

JS:

```javascript
let parObj = document.getElementById("thePar");
parObj.innerHTML = "Hello, 331!";
```
Example 2 Code

```html
<html>
<head>
  <title>2. HTML5 Canvas</title>
</head>
<body>
  <script type="text/javascript">
    function drawSomething() {
      let canvas = document.getElementById("theCanvas");
      let context = canvas.getContext("2d");
      context.fillStyle = "teal";
      context.fillRect(50, 50, 150, 100);
    }
  </script>
  <button onclick="drawSomething()">Draw Something Cool</button>
  <br />
  <canvas id="theCanvas" width="500" height="500"></canvas>
</body>
</html>
```
Example 4:
React Canvas

Our previous canvas code from Example 2 converted to use React!

1. Need to get a canvas object to draw like last time. This is different in React.
   a. It’s React’s job to manage the HTML for us, grabbing something with an ID defeats that purpose and can cause bugs.
   b. In React, we use “Ref” objects instead of ID strings, but they work similarly.
2. Write an `updateCanvasImage()` method to draw a rectangle on the canvas like before.
3. Use `componentDidMount()` to find out when React is ready for us to start drawing things, then call `updateCanvasImage()`
Component Life-Cycle

Creation of a component:
1. constructor (passed props) — create initial state
2. render — produce the HTML
3. componentDidMount — anything you need to do after HTML exists in the DOM

Update of component (after setState):
1. render
2. componentDidUpdate

Removal of component (after parent’s re-render removes child):
1. componentWillUnmount
Add some more components:

- New component to choose a color (red, green, blue)
- New component that puts a title above the canvas

Add state:

- **App** stores a current color
  - passed to child components in **props**
  - **ButtonGroup uses a callback to notify App** that a new color has been chosen
    - callback is passed in via **props** also
  - **App calls setState, which causes a re-render**
    - React re-creates or updates the children as necessary
The *Flow*
Summary

- Components are reusable blocks of code that allow modular design and proper cohesion.
- Components contain other components and HTML tags to determine how they appear on a webpage.
  - React is responsible for managing the underlying webpage.
- Data owned/controlled by a component is stored in that component’s state.
- Data flows $down$ from parent to child through props.
- Data flows $up$ from child to parent through callbacks from the child into the parent’s code.
- React notifies components of changes to their data through lifecycle methods, like `componentDidUpdate`
REACT BUGS
Common React Bugs

• Most common bugs in React are:
  – Reading from React state before the data has been populated.
  – Not properly understanding the React life cycle (the order that things happen within your app).

• This is because of specific asynchronous updates to React’s internal representation of the webpage.
  – Note: There may be a very slight delay to updating your React components.

• IMPORTANT: You need to be careful when updating your React component’s state and trying to access data!
Debugging React Strategies

When you hit a bug…

1. Walk step by step through the order that your code runs, checking how the state should be populated.
   – Use documentation about the React lifecycle to help you figure out which things happen in which order.

2. Put a `console.log()` in your methods if needed, and in `componentDidUpdate()` to check when your state was updated.

3. **Last resort:** Googling may be useful! Be **very** careful about this.
Bug 1 – “Read before Write”

Expected Functionality:
• Adds a canvas to the page and displays a blue rectangle immediately.

Current Functionality:
• TypeError when the page is loaded.
It seems like `this.canvasRef.current` is null, when it's supposed to be our Canvas object.

- Why doesn't the Canvas object exist yet? Let's think about how the `<canvas>` is eventually inserted into the page...

1. Our component is created and inserted into the page (in this case by `ReactDOM.render()`)
2. React constructs the component and then calls the component’s `render()` to get the HTML tags we want.
3. React inserts those tags into the webpage **and then** sets up all the reference objects.
Bug 1 – “Read before Write”

(Part of) The React Lifecycle

- Order that React calls methods.
- We're accessing the reference during the constructor.
- React doesn't update the refs (yellow box) until after `render()` — so they don’t exist when the constructor is running!

Bug 1 – The Fix

(Part of) The React Lifecycle

Mounting
- constructor

Updating
- New props
- setState()
- forceUpdate()
- render
- React updates DOM and refs
- componentDidMount
- componentDidUpdate

Solution
Override `componentDidMount` called when React is done inserting all the DOM nodes and updating refs.

In `componentDidMount`, we know it's safe to use the ref ("read"), since it’s guaranteed to happen after the updating the refs (the "write") has finished.

Bug 1 – The Fix

• Move the `updateCanvasImage` call into `componentDidMount`
  – Still called during the component "mounting" phase – so we're able to set up the "first look" of the canvas like we wanted.
  – Happens after React sets up our refs, so we know we'll have a valid Canvas object to work with.

• Common idea in React:
  – Set something up (like the `<canvas>` tag) and give it to React (by returning from `render`)
  – Some time later, React will do its job.
  – React makes a `callback` (like `componentDidMount`) to let us know that it's done and we can use whatever we set up (like accessing the Canvas through its ref).
Bug 2 – “React Doesn’t Know”

Expected Functionality:
• When the button is clicked, the message on the page changes to "I've been clicked!"

Current Functionality:
• The message on the page never changes.
• We know that the button event is working because the `console.log()` inside the listener is being run, so the bug must be somewhere else.
Bug 2 – The Problem

• The `this.clicked` variable is being updated correctly
  – You can print it out to double check, if you'd like.

• The only place we can modify what text is being put in the `<p>` is during the `render()` method – we need to return a different `<p>` element to change what's on the page.
  – But React doesn't know it's supposed to call `render` again!
  – More accurately: React doesn't know that the contents of the `this.clicked` variable matters for `render`. 
Bug 2 – The Solution?

- React has a special place for variables that affect how a component renders: `this.state`.
  - Store an object ( `{...}` ) inside `this.state`, put whatever properties we want in that object to track the data we need.
  - Instead of `this.clicked`, we write `this.state.clicked`.
- "clicked" isn't a special name here – just a variable name. Could easily call it "this.state.pizza"
- "state" is a special name: part of the React convention, React code expects that you store your state variables in `this.state`.

- Not quite a fix yet, but a step in the right direction.
Bug 2 – The New Problem

• We're now storing our data in the right place, but we still aren't telling React when we change it.
  – React requires that you notify it when you want to change the data, instead of changing it yourself.

• To request a state change, call `this.setState` and pass it an object representing the changes you want to make.
  – You should never directly modify the contents of `this.state` (except for constructor initialization)

• Since you use `this.setState` (which is React code) to update the state, React knows that you’ll need things to be updated based on what changed. (So, React will re-do the render).
Bug 2 – The Fix

By calling `setState` to update our state, we trigger a "component update cycle". During an update, React will change the state and then re-call `render()`. In `render()`, we can return the new text to be displayed on the page.

Bug 3 – “Read before Write (is done)”

Expected Functionality:
• When a button is clicked, a square of that color appears in the canvas.
• The current color is displayed in the text above the buttons.

Current Functionality:
• The text above the buttons seems to be working correctly.
• The canvas is lagging behind one click – displays the color from two clicks ago instead of the most recent click.
Bug 3 – The Problem

• Remember that, in React, `setState` is a *request* for a *future change* to state. *When `setState` returns, the state has not yet been updated.*
  – React delays state changes for performance reasons.
  – Means we need to be careful about reading state: when do we know that it's guaranteed to be up-to-date?

• The problem is that we're trying to access the state immediately after calling `setState` – React hasn't gotten around to updating the state yet, so we're seeing the old value.
  – This is why the canvas is "lagging behind" by one: when we draw the canvas, we're seeing the value of state from the previous button press.
Bug 3 – What’s Actually Happening

(Part of) The React Lifecycle

Mounting

- constructor

- componentDidMount

Updating

- New props
- setState()
- forceUpdate()

- render

- React updates DOM and refs

- componentDidUpdate

setState returns, our code keeps going

Our code calls this.drawSquare, which sees the old value still in this.state

this.state gets changed later (by React)

The <p> gets updated here, which is why it sees the correct state (state is guaranteed to be updated before render is called)

Bug 3 – The Fix

(Part of) The React Lifecycle

- Mounting
  - constructor
  - componentDidMount

- Updating
  - New props
  - setState()
  - forceUpdate()
  - render
  - componentDidUpdate
  - React updates DOM and refs

setState returns, our code keeps going

this.state gets changed later (by React)

<p>gets updated here, so it sees the correct state

Solution
Should call drawSquare, here, since it is guaranteed to happen after the this.state value has been updated.