

Class 7: Controlled Inputs + TailwindCSS

1. Update: Oral Exam I
2. Review: Last Class
3. Controlled Inputs
4. Utility First CSS Frameworks
5. Studio

Copyright 2026, Kyle J. Harms

Oral Exam I: A Mock Technical Interview

In a few weeks, you will be asked to code Vue.js component prototype in a **mock technical interview**.

No AI is permitted during the exam. You will need to demonstrate your ability to use Vue.js to implement a functional component prototype that meets the requirements of a design specification.

Review: Sprints

Scrum sprints are short, time-boxed periods (often a week or two, but less than a month) where teams focus on completing a set amount of work, enabling incremental delivery.

Source: [Atlassian](#)

Review: Coding the UI Design Too Early

Common mistake: Many teams code the user interface first (not the functionality), before evaluating whether it was possible to implement the design's functionality.

This often leads to **design lock-in**, because the team has invested a lot of time and effort into the design, and it can be **emotionally** difficult to change the design (sunk cost fallacy).

Review: Homework 2, Sprint 1

Implement an unstyled, but functional, drum simulator app prototype using the following methods:

- components
- props
- reactive variables
- prop binding
- event handling
- emit custom events
- conditional rendering using `v-if`
- conditional styling using `:class` bindings
- rendering lists of elements using the `v-for` directive
- watchers to react to changes in reactive variables

Controlled Inputs

Review: React Controlled Inputs

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

export default function App() {
  const [checked, setChecked] = useState(false);
  return (
    <div>
      <input
        type="checkbox"
        checked={checked}
        onChange={(e) => setChecked(e.target.checked)}
      />
      <p>Checked: {checked.toString()}</p>
    </div>
  );
}
```

Activity: Vue Controlled Inputs

Working with a peer, implement controlled inputs in Vue.js in **Part I** of your **handout**.

```
export default function App() {  
  const [checked, setChecked] = useState(false);  
  return (  
    <div>  
      <input  
        type="checkbox"  
        checked={checked}  
        onChange={(e) => setChecked(e.target.checked)}  
      />  
      <p>Checked: {checked.toString()}</p>  
    </div>  
  );  
}
```

Solution: Vue Controlled Inputs

```
<script setup>
import { ref } from "vue";
const checked = ref(false);
</script>

<template>
  <div>
    <input type="checkbox"
      :checked="checked"
      @change="(e) => checked = e.target.checked" />
    <p>Checked: {{ checked }}</p>
  </div>
</template>
```

v-model – A Better Controlled Input

Bind user input data from input elements (i.e. `<input>`, `<textarea>`, `<select>`) to a reactive variable using the `v-model` directive:

```
<script setup>
const checked = defineModel(false);
</script>

<template>
  <div>
    <input type="checkbox" v-model="checked" />
    <p>Checked: {{ checked }}</p>
  </div>
</template>
```

Note: `defineModel` returns a reactive variable.

Activity: Vue Controlled Inputs with `v-model`

Working with a **peer**, complete **part II** of your **handout** using the `v-model` directive to bind user input data from a `<textarea>` element to a reactive variable.

```
<script setup>
const checked = defineModel(false);
</script>

<template>
  <div>
    <input type="checkbox" v-model="checked" />
    <p>Checked: {{ checked }}</p>
  </div>
</template>
```

Utility First CSS Frameworks

TailwindCSS

TailwindCSS

TailwindCSS is a utility-first CSS framework for rapidly building custom user interfaces.

Instead of writing custom CSS styles, you apply pre-defined utility classes directly in your HTML or Vue.js templates.

Example:

```
<button class="bg-blue-500 text-white font-bold py-2 px-4 rounded">  
  Next  
</button>
```

Why Prototype with TailwindCSS?

TailwindCSS provides a **design system** that allows you to rapidly prototype user interfaces without writing custom CSS styles.

This helps provide a **consistent look and feel** across your app, while also allowing you to **quickly iterate** on your design.

Other utility-first CSS frameworks also work, like [daisyUI](#).

Prototyping with TailwindCSS

1. Identify the utility classes you need for your design.
2. Apply the utility classes directly in your HTML or Vue.js templates.
3. Refresh your browser to see the changes.

Gotcha: TailwindCSS does not automatically include all utility classes by default. It includes only the classes that are used in your project files to minimize file size. When adding a new class, you may need to **refresh** your browser to see the changes.

Demo: TailwindCSS

```
<button class="bg-blue-500 text-white font-bold py-2 px-4 rounded">  
  Next  
</button>
```

Activity: TailwindCSS Practice

1. **Discuss** with a peer how to design the drum simulator app to have a **fun, joyful, playful**, and **energetic** theme.
2. **Identify** TailwindCSS classes that would support your design.
3. **Working on one person's computer**, start implementing the design in your Homework 2 repository.

Note: If you have not submitted your Homework 2, Sprint 1, do not add any TailwindCSS classes to your app prototype until you have submitted Sprint 1.

Studio: Homework 2, Sprint 2

Goal: Style your functional drum simulator app prototype **mostly** using TailwindCSS and demonstrate your ability to use Vue.js's `v-model` directive.

App Theme

The drum simulator app should have a **fun, joyful, playful**, and **energetic** theme. It should not be professional or serious.

Tip: Change the entire color theme of the app.

`v-model`

Add a **speed** feature to your drum simulator app using an HTML input element.

Initialize the speed to around 350 ms between steps and allow the user to change the speed.

What's Next

Homework 2, Sprint 2 - Due Thursday, Sep 17, 2026, 11:59pm

Next Week: Sprint Planning + GitHub Projects, Homework 3

Following Week: Oral Exam I