Intro to Vue 3

Lessons

1. Intro to Vue 3

2:22

2. Creating the Vue App

6:56

3. Attribute Binding

3:53

4. Conditional Rendering

5:11

5. List Rendering

3:31

6. Event Handling

4:31

7. Class & Style Binding

6:37

8. Computed Properties

6:23

9. Components & Props

6:38

10. Communicating Events

3:57

11. Forms & v-model

8:27

Components & Props

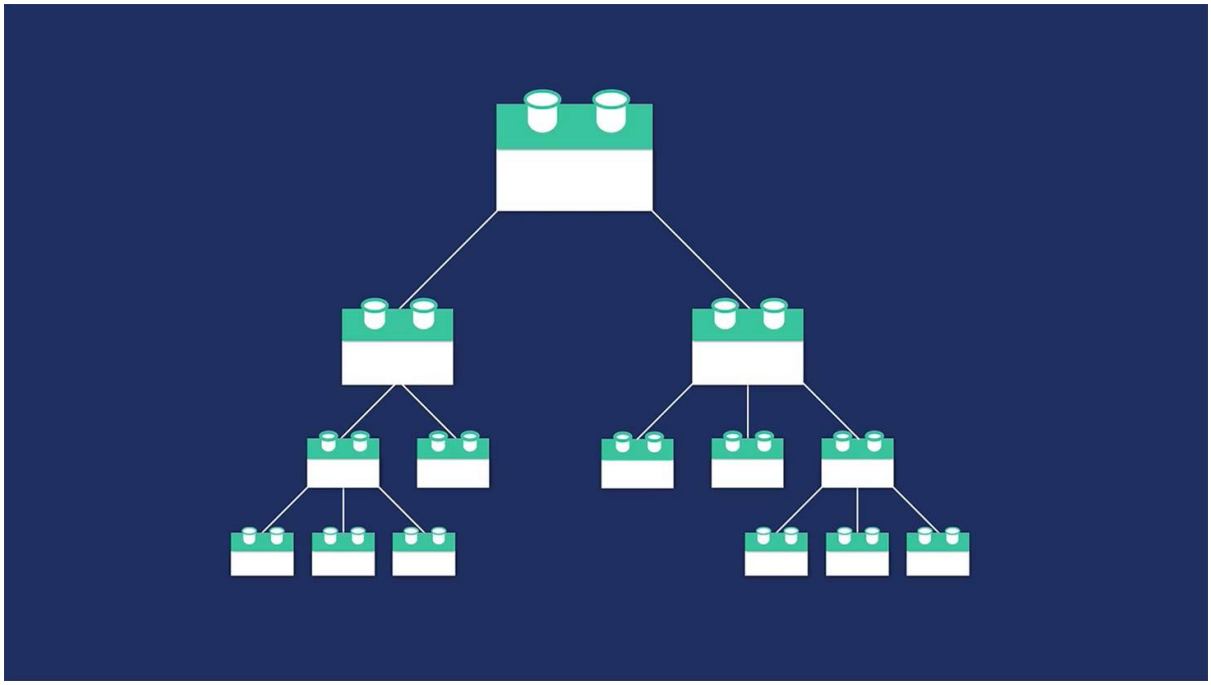
In this lesson, we're going to look at the concept of Vue.js components and props. If you're coding along with the repo, you can checkout the `L9-start` branch or the starting code on Codepen.

Our Goal

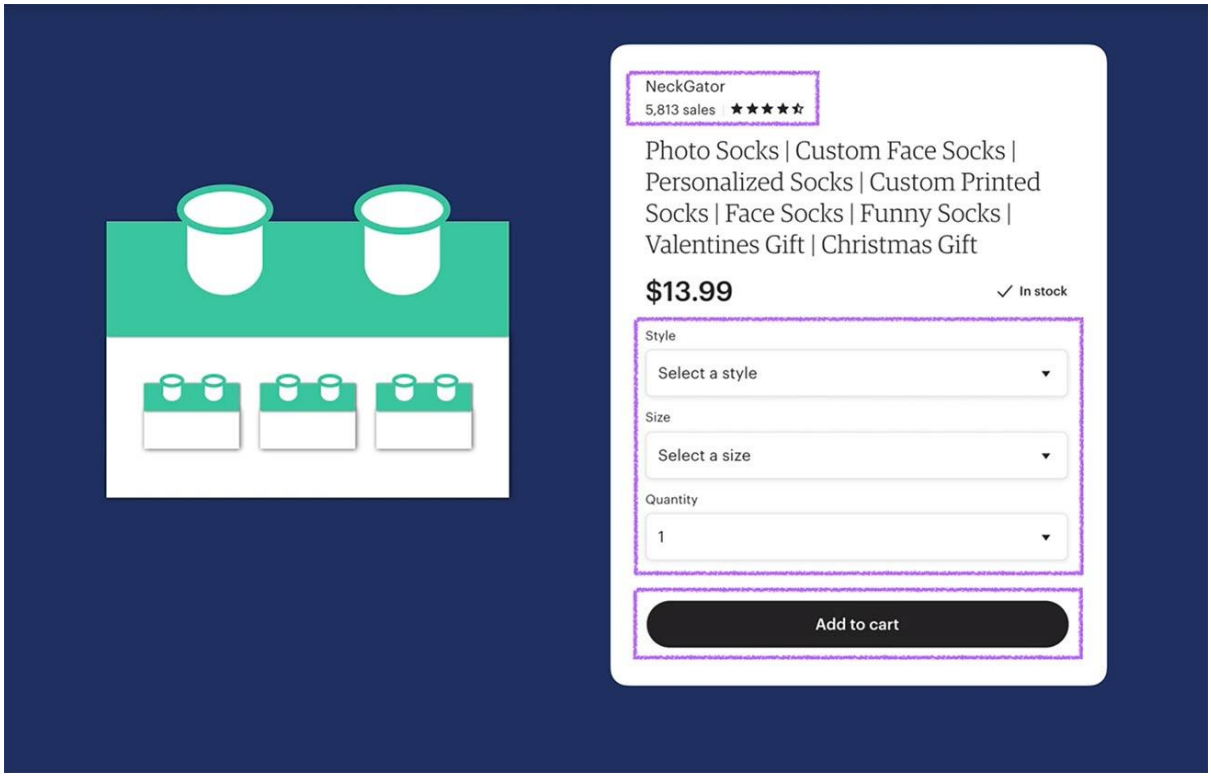
Refactor the example app to use a product component, which uses a prop.

Building Blocks of a Vue App

In modern front-end JavaScript frameworks, components are the building blocks of an app, and that's certainly the case with Vue. You can imagine components a bit like Legos that you can plug into each other in a component family tree hierarchy.

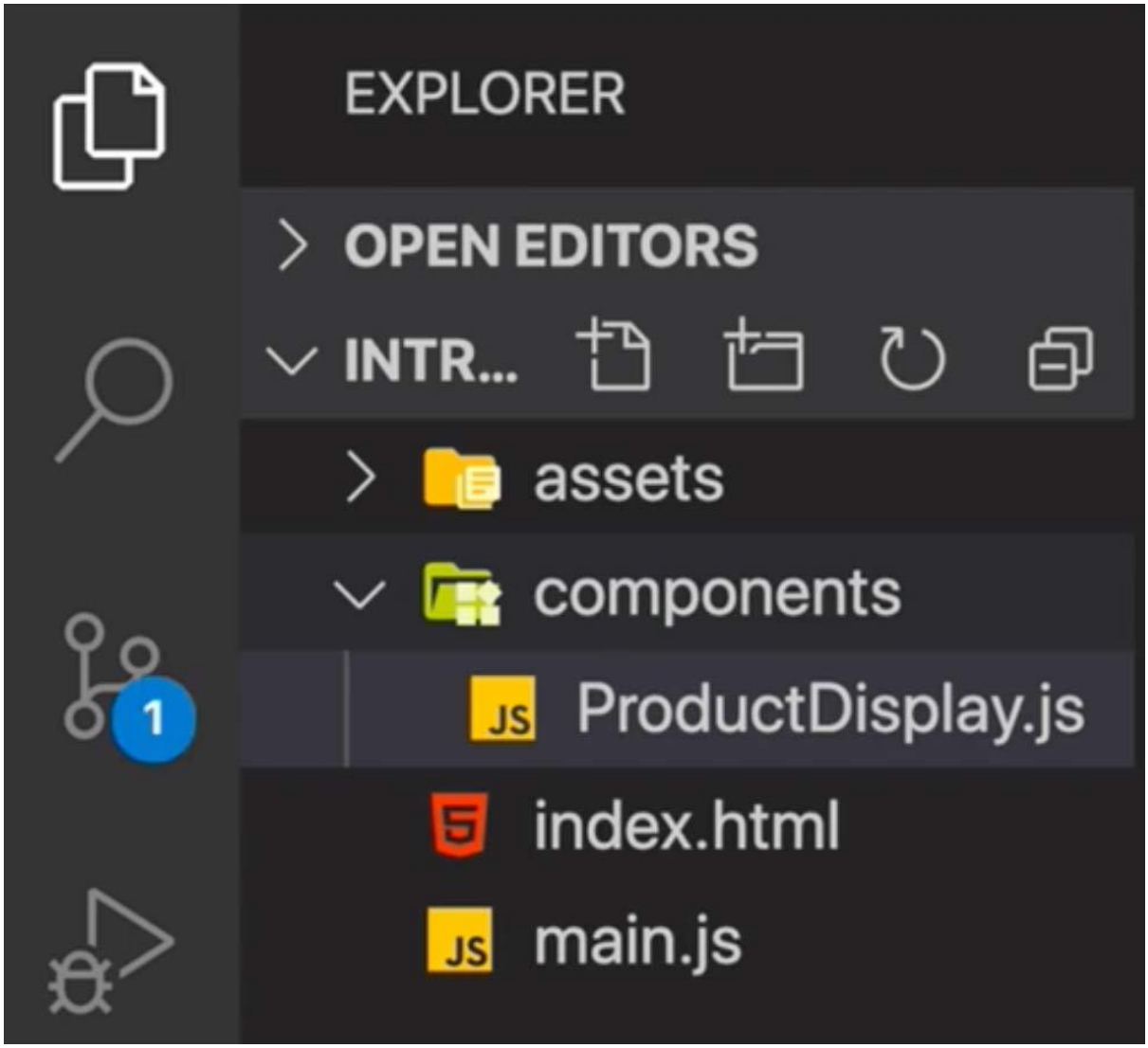


Any given web page may be composed of multiple components, and it's common for components to be "parents" that have child components nested within them.



Creating our First Component

Let’s head in to our app and create our first component. Because our app will eventually have multiple components, we’ll create a **components** folder, inside of which we’ll create our first component, called **ProductDisplay.js**.



The syntax to create a component looks like this:

components/ProductDisplay.js

```
app.component('product-display', {})
```

The first argument is the component name, `'product-display'` in this case, and the second argument is an object to configure our component (similar to the options object used to configure our root Vue app).

Template

Because we need our component to have structure, we'll add the `template` property and paste all of the product-based HTML code that currently resides in **index.html into here**, within a template literal.

📄components/ProductDisplay.js

```
app.component('product-display', {
  template:
    /*html*/
    `<div class="product-display">
      <div class="product-container">
        <div class="product-image">
          <img v-bind:src="image">
        </div>
        <div class="product-info">
          <h1>{{ title }}</h1>

          <p v-if="inStock">In Stock</p>
          <p v-else>Out of Stock</p>

          <div
            v-for="(variant, index) in variants"
            :key="variant.id"
            @mouseover="updateVariant(index)"
            class="color-circle"
            :style="{ backgroundColor: variant.color
}">
          </div>

          <button
            class="button"
            :class="{ disabledButton: !inStock }"
            :disabled="!inStock"
            v-on:click="addToCart">
            Add to Cart
          </button>
        </div>
      </div>
    </div>`
})
```

Note that we haven't changed any of this code, we're simply moving it into the `product-display` component so that it's encapsulated there. If you're wondering what the `/*html*/` is doing there, that activates the VS Code extension es6-string-html, which gives us syntax highlighting even though we're in this template literal.

Data & Methods

Now that we've given this component its template, or its structure, we need to give it its data and methods, which still live in **main.js**. So we'll paste them in now:

📄components/ProductDisplay.js

```
app.component('product-display', {
  template:
    /*html*/
    `<div class="product-display">
      ...
    </div>`,
  data() {
    return {
        product: 'Socks',
        brand: 'Vue Mastery',
        selectedVariant: 0,
        details: ['50% cotton', '30% wool', '20%
polyester'],
        variants: [
          { id: 2234, color: 'green', image:
'./assets/images/socks_green.jpg', quantity: 50 },
          { id: 2235, color: 'blue', image:
'./assets/images/socks_blue.jpg', quantity: 0 },
        ]
    }
  },
  methods: {
      addToCart() {
          this.cart += 1
```

```
        },
        updateVariant(index) {
            this.selectedVariant = index
        }
    },
    computed: {
        title() {
            return this.brand + ' ' + this.product
        },
        image() {
            return
this.variants[this.selectedVariant].image
        },
        inStock() {
            return
this.variants[this.selectedVariant].quantity
        }
    }
})
```

We'll make sure to delete `cart` from the `data` here because we don't need each product to have its own cart.

Cleaning up main.js

Now that we've encapsulated all of this product-specific code within our `product-display` component, we can clean up our **main.js** file.

📄main.js

```
const app = Vue.createApp({
    data() {
      return {
        cart: 0,
      }
    },
    methods: {}
})
```

We've left the `cart` and the `methods` option because it'll have a new method later on.

Importing the Component

In order to make use of `product-display`, we need to import it into we'll **index.html**.

📄index.html

```
<!-- Import Components -->
<script src="./components/ProductDisplay.js"></script>
```

Now that it's imported, we can use it within our template.

📄index.html

```
<div id="app">
  <div class="nav-bar"></div>

  <div class="cart">Cart({{ cart }})</div>
  <product-display></product-display>
</div>
```

If we check this out in a browser, we'll see everything is still showing up just like it was before, but now since we've rearranged things, the Add to Cart button is no longer incrementing the cart. We'll fix that in the next lesson.

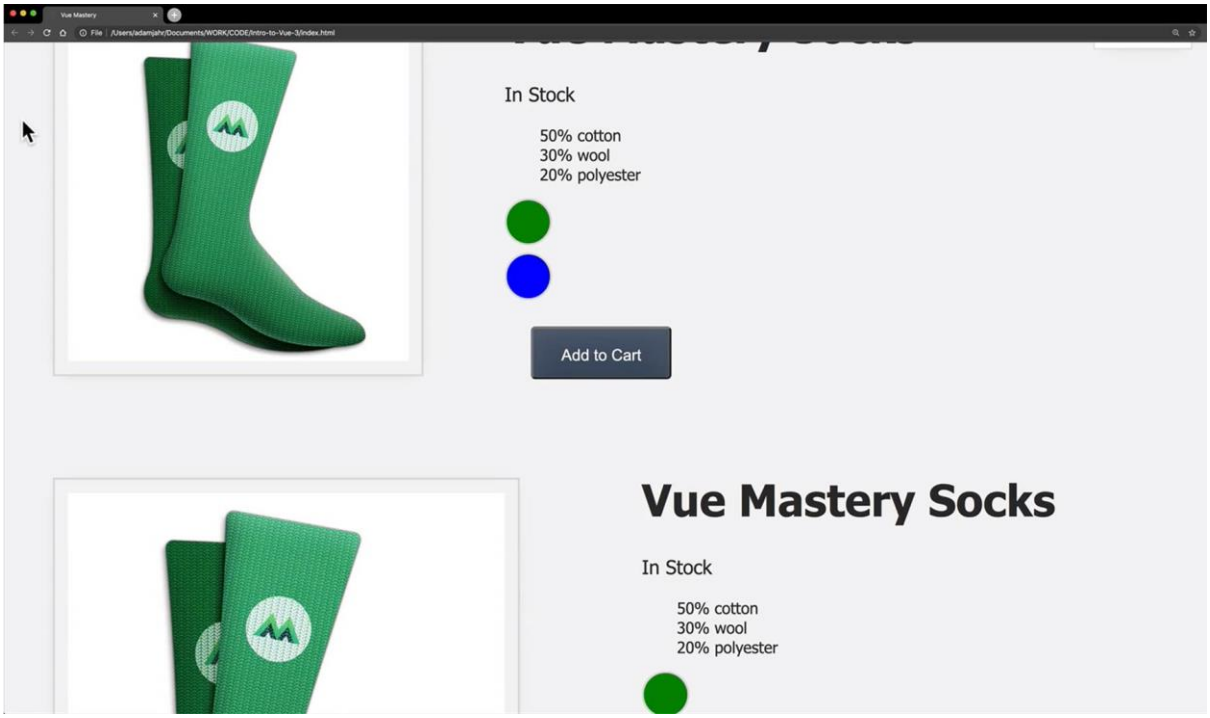
For now, to show you how helpful these reusable blocks of code can be, I'm going to add two more `product-display` components.

📄index.html

```
<div id="app">
  <div class="nav-bar"></div>
```

```
  <div class="cart">Cart({{ cart }})</div>
  <product-display></product-display>
  <product-display></product-display>
  <product-display></product-display>
</div>
```

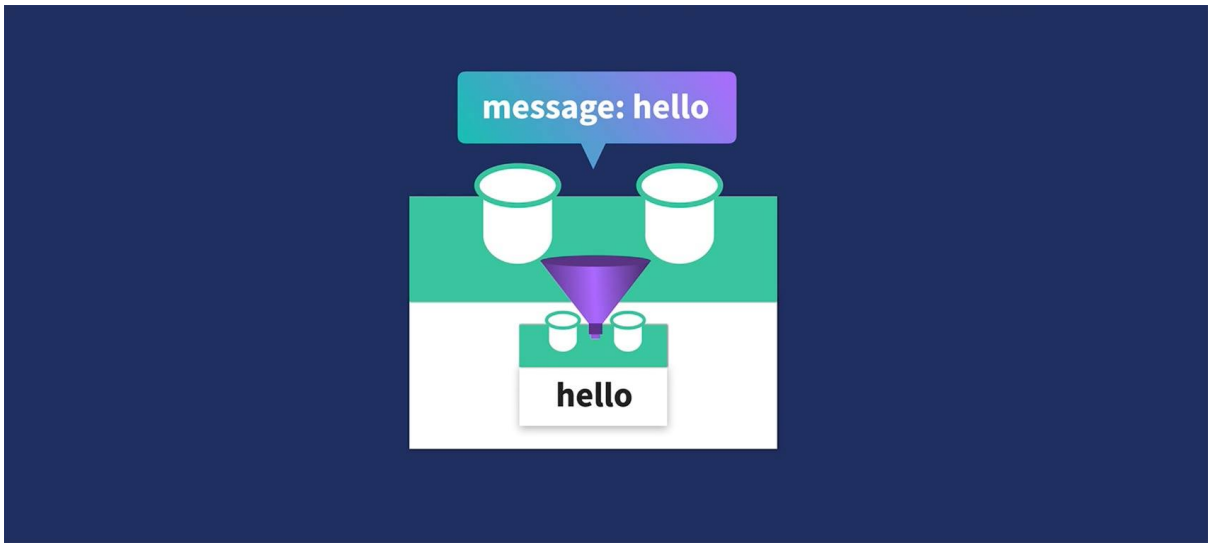
When we refresh the browser, we'll see them all showing up. Each of them are independently functional.



Props

Now that we're starting to learn how to encapsulate reusable code into these components, what happens when our component needs something that is outside of itself? For example, what if the parent, so-to-speak, had some `message` data, and the child needed it? Because a component has its own isolated scope, it can't reach up outside of itself to grab something that's outside of its scope.

The answer here is **props**. These are custom attributes for passing data into a component. They function kind of like a funnel, into which you can pass the data the component needs.



Let's add the ability for our `product-display` component to take in a prop.

Giving our component a prop

Let's give our root Vue app, located in **main.js**, a new data property, which indicates whether the user was a `premium` user or not.

📄main.js

```
const app = Vue.createApp({
  data() {
    return {
      cart: 0,
      premium: true
    }
  }
})
```

If a user is `premium`, their shipping will be free. So our `product-display` component needs access to this data. In other words, it needs a custom attribute (a funnel) that we can feed this data into. Let's add that now, which we'll do by giving the component a `props` option, and adding a `premium` prop to it.

📄components/ProductDisplay.js

```
app.component('product-display', {
  props: {
    premium: {
      type: Boolean,
      required: true
    }
  },
  ...
}
```

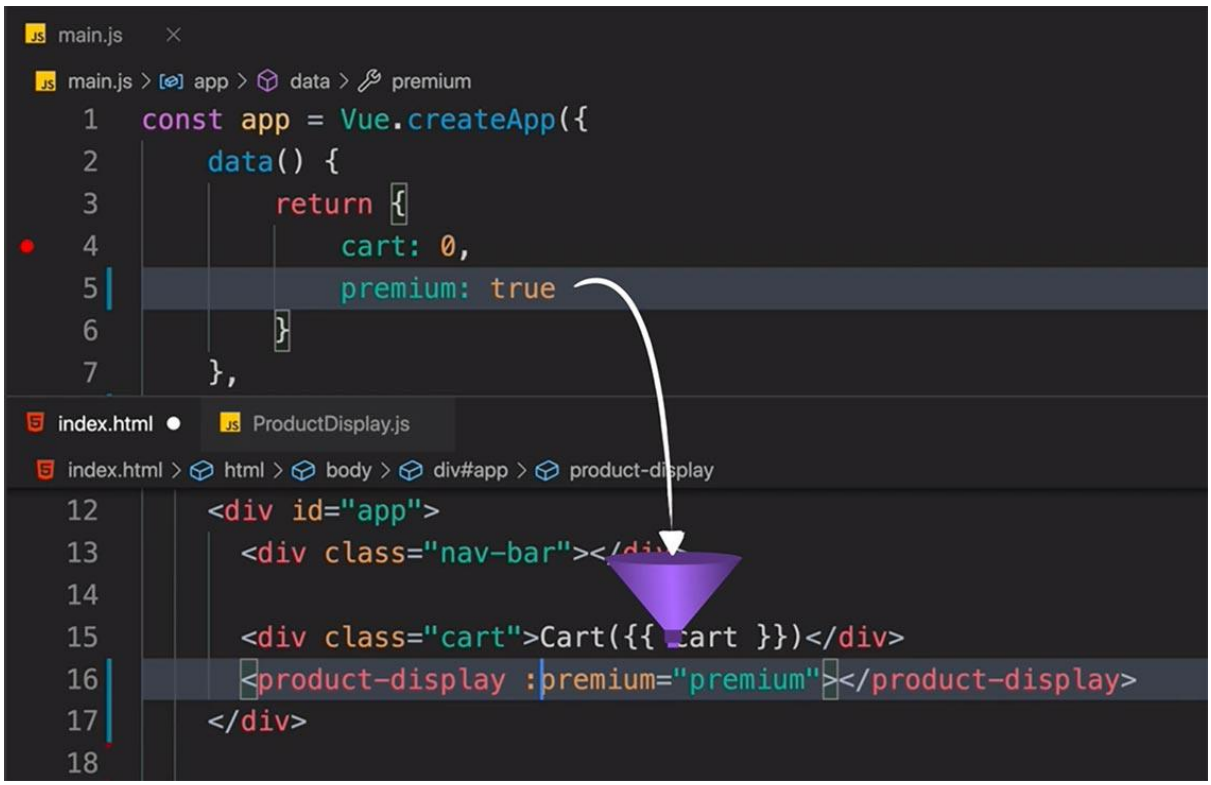
Notice how Vue's props feature has built-in validation, so we can specify things like the prop's `type` and whether it's `required`, etc.

Now that we've configured this, we can add that custom attribute onto the `product-display` component where we're using it.

📄index.html

```
<div id="app">
  <div class="nav-bar"></div>

  <div class="cart">Cart({{ cart }})</div>
  <product-display :premium="premium"></product-
display>
</div>
```



Notice how we're using the shorthand for `v-bind` so we can reactively receive the new value of `premium` if it updates (from `true` to `false`).

Using the Prop

Now that our `product-display` component has the `premium` prop, we can make use of it within the component. Remember, we want to use the fact a user is `premium` or not to determine what they need to pay for shipping.

In the component's template, we'll add:

components/ProductDisplay.js

```
template:
  /*html*/
  `<div class="product-display">
    ...
      <p>Shipping: {{ shipping }}</p>
    ...
  </div>`,
```

Here, `shipping` is the name of a new computed property on the `product-display` component, which looks like this:

components/ProductDisplay.js

```
computed: {
  ...
  shipping() {
    if (this.premium) {
      return 'Free'
    }
      return 2.99
    }
}
```

The computed property checks whether the `premium` prop is `true`, and if so, returns `'Free'`. Otherwise, it returns `2.99`

Coding Challenge

We've reached the end of the lesson and we're onto our challenge:

Create a new component called `'product-details'`, which receives the `details` through a prop called `details`.

As a reminder, if you're coding along with our repo, you can check out `L9-end` branch, and you can view the solution code on Codepen.

Download Video
Share Lesson

Lesson Resources

- Starting Code
- Ending Code

Discuss in our Facebook Group Send us Feedback

Previous Lesson Next Lesson

-
-
-
-
-
-
-

ABOUT US

-
-
-
-