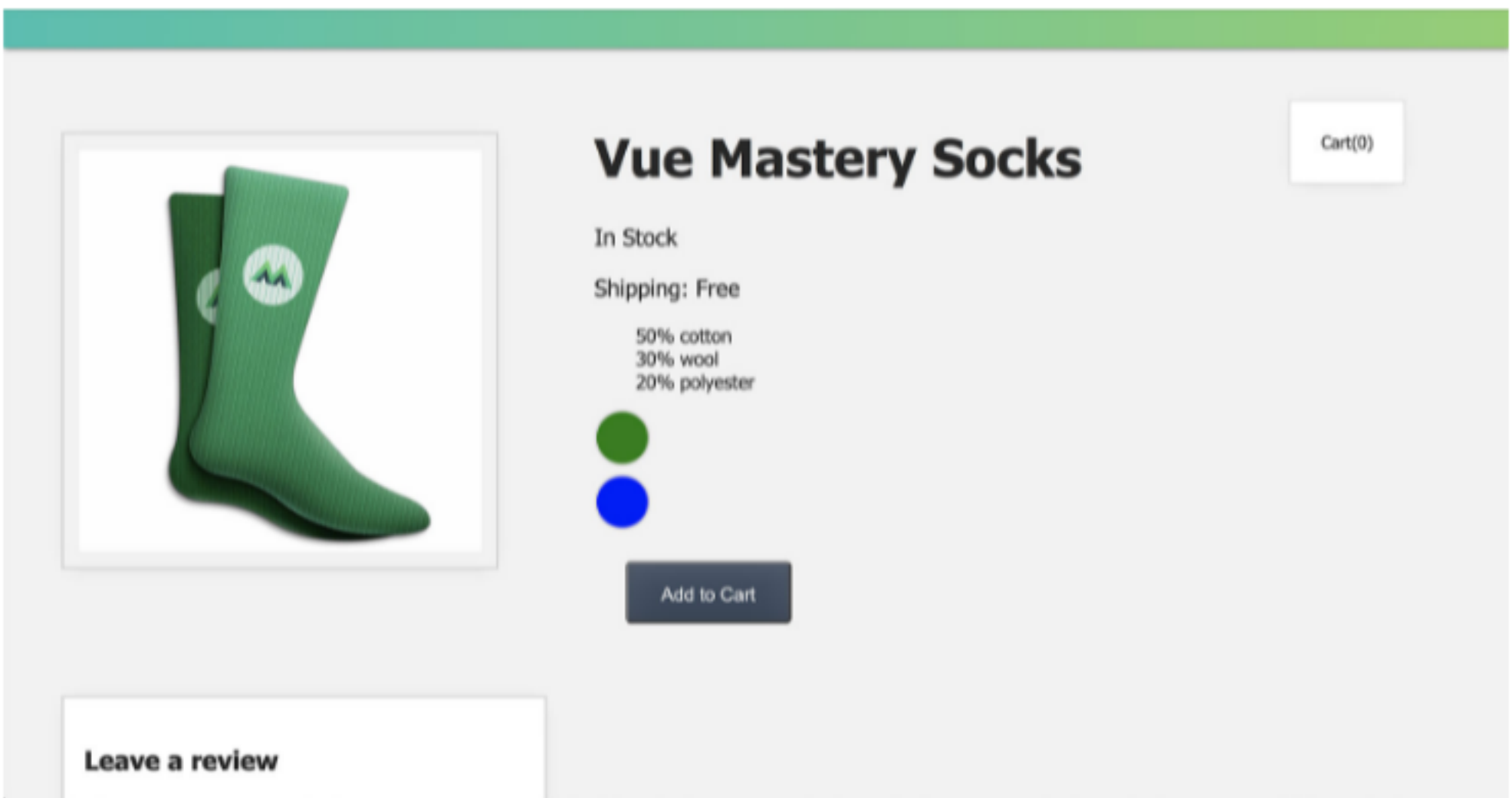
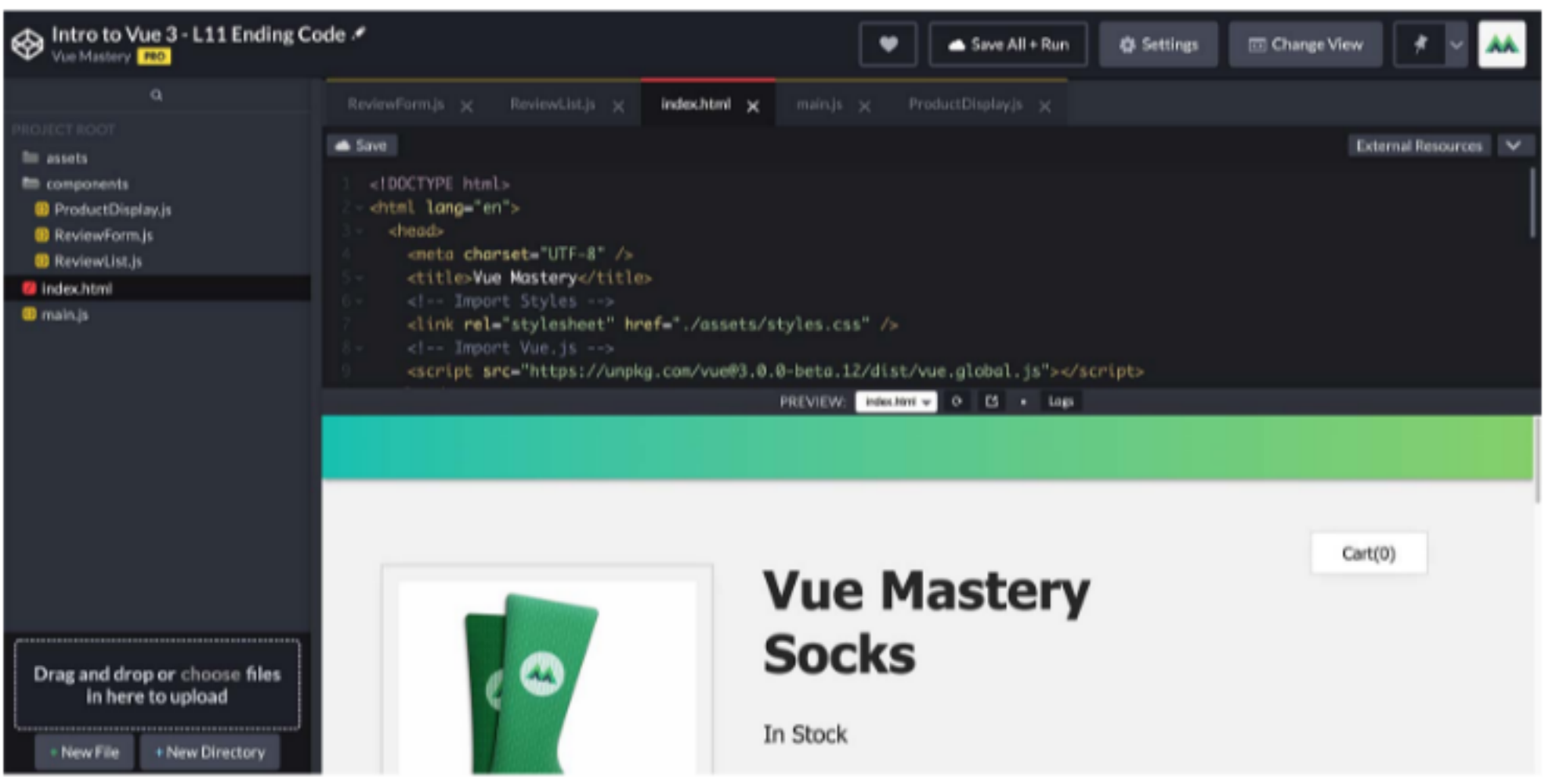
Intro to Vue 3

Welcome to Vue Mastery's Intro to Vue 3 course. We're going to take a journey through the Vue universe to explore the technology and build a solid foundation of new skills, so we can build upon that to head towards Vue mastery. This course is your first step along the way, and on our platform you'll find a whole library of courses you can take to level-up your skills.

Throughout this course, we'll learn the fundamentals of Vue.js and build an app to put these concepts in practice.



To get the most out of this course, you're going to want to be coding along. And there's two ways to do that, depending on your experience level. If you're newer to programming, you're going to want to click on the link below this video that takes you over to our Codepen account.



There, you'll find our starting code, which you can fork to your personal account. However, with the free CodePen account, you can only have one project at a time. So if you already have a project in your account, you're going to have to delete it out to make a new one for this course, which you build upon each lesson.

If you have more experience with programming and using git, and Github, you're going to want to head over to the course's repo and clone the project. From there, you can switch between different branches for the starting and ending code for each lesson. For example, in the next lesson (lesson two) you can check out `L2-start` or `L2-end` for the starting and ending code, respectively.

Throughout this course, you're going to see me using VS code. I do recommend this code editor for Vue development, but if you already have a code editor of your choice, you can stick with that one. If you want to use VS code and you don't already have it, you're going to want to download it now and also install an extension es6-string-html. You'll see why this extension will be useful in later lessons.

Finally, at the end of each lesson, you'll see a coding challenge so you can put the concepts into practice. Now, if you're ready, let's create our first Vue app in the next lesson.

Communicating Events

In this lesson, we're going to look at the concept of communicating events that happen within our components. If you're coding along with the repo, you can checkout the `L10-start` branch or the starting code on Codepen.

Our Goal

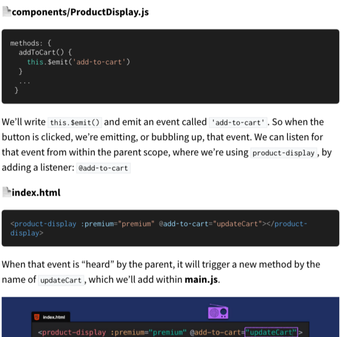
Give our component the ability to let its parent know an event happened within it.

Emitting the event

When we refactored things in our last lesson, moving product-related code into the new `product-display` component, we broke the ability to click the Add to Cart button and increment the value of `cart`. This is because the `cart` lives outside of the `product-display` component, inside the root Vue app in **main.js**.

We need to give the `product-display` component a way to *announce* that its button is clicked. How do we get this to happen?

We already know that props are a way to pass data down into a component, but what about when something happens within that component, like a button click? How do we let other parts of our app know that that event happened?



The answer is to emit that event, telling the parent that it happened. Let's add this ability within our `product-display` component, by amending the `addToCart()` method.

components/ProductDisplay.js

```
methods: {
  addToCart() {
    this.$emit('add-to-cart')
  }
  ...
}
```

We'll write `this.$emit()` and emit an event called `'add-to-cart'`. So when the button is clicked, we're emitting, or bubbling up, that event. We can listen for that event from within the parent scope, where we're using `product-display`, by adding a listener: `@add-to-cart`

index.html

```
<product-display :premium="premium" @add-to-cart="updateCart"></product-display>
```

When that event is "heard" by the parent, it will trigger a new method by the name of `updateCart`, which we'll add within **main.js**.

main.js

```
const app = Vue.createApp({
  data() {
    return {
      cart: [],
      ...
    }
  },
  methods: {
    updateCart() {
      this.cart += 1
    }
  }
})
```

If we check this out in the browser, we should now be able to click the Add To Cart button, which lets the parent know the `add-to-cart` event happened, triggering the `updateCart()` method.

Adding product id to the cart

To make our app more realistic, our `cart` shouldn't just be a number. It should be an array that contains the ids of the products that are added into it. So let's do a bit of refactoring.

main.js

```
const app = Vue.createApp({
  data() {
    return {
      cart: [],
      ...
    }
  },
  methods: {
    updateCart(id) {
      this.cart.push(id)
    }
  }
})
```

Now, `cart` is an array and `updateCart(id)` pushes the product `id` into it. We just need to add a payload to our `add-to-cart` event emission, so `updateCart` has access to that `id`.

components/ProductDisplay.js

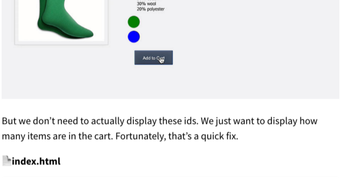
```
methods: {
  addToCart() {
    this.$emit('add-to-cart', this.variants[this.selectedVariant].id)
  }
  ...
}
```

Here, we've added a second parameter and grabbed the product's `id` much like we grabbed the `image` and `quantity` before:

components/ProductDisplay.js

```
computed: {
  image() {
    return this.variants[this.selectedVariant].image
  },
  inStock() {
  return this.variants[this.selectedVariant].quantity
  }
}
```

Now in the browser, you can see that we're adding the product ids into the cart, which is now an array.



But we don't need to actually display these ids. We just want to display how many items are in the cart. Fortunately, that's a quick fix.

index.html

```
<div id="app">
  ...
  <div class="cart">Cart({{ cart.length }})</div>
  ...
</div>
```

By adding `cart.length` we'll now only display the *amount* of items in the `cart`.

Coding Challenge

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

Add a new button to `product-display`, which removes the product from the `cart`.

As a reminder, if you're coding along with our repo, you can check out `L10-end` branch, and you can view the solution code on Codepen.

Forms & v-model

In this lesson, we're going to look at the concept of attribute binding. If you're coding along with the repo, you can checkout the `01-start` branch or the `starting-code` on Codepen.

Our Goal

Create a form for users to add product reviews.

Introducing v-model

At the beginning of this course, we learned about `v-bind`, which creates a one-way binding. Now it's time to do the template. When working with forms, however, this one way binding isn't enough. We also need to be binding from the template to the data.

For example, when a user inputs their name into an input field, we want to record and store that value in our data. The `v-model` directive helps us achieve this, creating two way data binding.

```
<div id="app">
  <input v-model="name" />
  <p>{{ name }}</p>
</div>

const app = new Vue({
  el: '#app',
  data: {
    name: 'Joe Smith'
  }
})
```

To see this all in action, we're going to create a new `review-form` component.

The review-form component

We'll add a new **ReviewForm.js** file into our components folder, and scaffold out the component.

components/ReviewForm.js

```
const component = Vue.extend({
  template:
  `<div>
    <h3>Leave</h3><br>review-form>
    <label for="name">Name:</label>
    <input id="name">
    <br>
    <br>
    <label for="review">Review:</label>
    <textarea id="review"></textarea>
    <br>
    <br>
    <label for="rating">Rating:</label>
    <br>
    <select id="rating">
      <option>1</option>
      <option>2</option>
      <option>3</option>
      <option>4</option>
      <option>5</option>
    <br>
    <br>
    <button type="submit" value="Submit">
  data() {
    return {
      name: '',
      review: '',
      rating: null
    }
  }
});
```

We'll achieve this by adding the `v-model` directive to each of those input elements.

components/ReviewForm.js

```
const component = Vue.extend({
  template:
  `<div>
    <h3>Leave</h3><br>review-form>
    <label for="name">Name:</label>
    <input id="name">
    <br>
    <br>
    <label for="review">Review:</label>
    <textarea id="review"></textarea>
    <br>
    <br>
    <label for="rating">Rating:</label>
    <br>
    <select id="rating">
      <option>1</option>
      <option>2</option>
      <option>3</option>
      <option>4</option>
      <option>5</option>
    <br>
    <br>
    <button type="submit" value="Submit">
  data() {
    return {
      name: '',
      review: '',
      rating: null
    }
  }
});
```

Notice how on the `textarea` element, we used `v-model.number`, this is a modifier that typecasts the value as a number.

Submitting the Review Form

In order to submit this form, we'll add a listener at the top:

components/ReviewForm.js

```
const component = Vue.extend({
  template:
  `<div>
    <h3>Leave</h3><br>review-form @submit.prevent="onSubmit">
    <label for="name">Name:</label>
    <input id="name">
    <br>
    <br>
    <label for="review">Review:</label>
    <textarea id="review"></textarea>
    <br>
    <br>
    <label for="rating">Rating:</label>
    <br>
    <select id="rating">
      <option>1</option>
      <option>2</option>
      <option>3</option>
      <option>4</option>
      <option>5</option>
    <br>
    <br>
    <button type="submit" value="Submit">
  data() {
    return {
      name: '',
      review: '',
      rating: null
    }
  }
});
```

We're using another modifier, `@submit.prevent="onSubmit"` to prevent the default behavior (a browser refresh). When this form is submitted, it will trigger the `onSubmit()` method, which we'll write now:

components/ReviewForm.js

```
const component = Vue.extend({
  template:
  `<div>
    <h3>Leave</h3><br>review-form @submit.prevent="onSubmit">
    <label for="name">Name:</label>
    <input id="name">
    <br>
    <br>
    <label for="review">Review:</label>
    <textarea id="review"></textarea>
    <br>
    <br>
    <label for="rating">Rating:</label>
    <br>
    <select id="rating">
      <option>1</option>
      <option>2</option>
      <option>3</option>
      <option>4</option>
      <option>5</option>
    <br>
    <br>
    <button type="submit" value="Submit">
  data() {
    return {
      name: '',
      review: '',
      rating: null
    }
  },
  methods: {
    onSubmit(review) {
      productReviews = {
        name: this.name,
        review: this.review,
        rating: this.rating,
      }
      this.$emit('review-submitted', productReviews)
      this.name = '',
      this.review = '',
      this.rating = null
    }
  }
});
```

That method will create a `productReview` object, containing the `name`, `review` and `rating` from our data. It will then fire a `review-submitted` event, sending that `productReview` along as the payload.

Finally, we're cleaning out the data fields.

Using the Review Form

Now that our review form is complete, we can import it within **index.html**.

index.html

```
<!-- scripts -->
<script src="./components/ReviewForm.js"></script>
```

Then we'll head into `product-display`, and actually use the component within its template, below the "product-container".

components/ProductDisplay.js

```
const component = Vue.extend({
  template:
  `<div>
    <div class="product-display">
      <div class="product-container">
      ...
    </div>
    <review-form></review-form>
  </div>`
});
```

Now in the Browser, we can see the review form.

It looks like it's working... except when we click the submit button, we're sending the event, but we haven't listened for it anywhere. Like we learned in the previous lesson, we need to listen for the `review-submitted` event in the parent scope (in `product-display`).

When the event is "heard", we'll add the `productReview` payload to the `product-display` component's data.

Adding product reviews

We'll add the event listener here on the `review-form`, where it's being used:

components/ProductDisplay.js

```
const component = Vue.extend({
  template:
  `<div>
    <div class="product-display">
      <div class="product-container">
      ...
    </div>
    <review-form @review-submitted="addReview"></review-form>
  </div>`
});
```

When the event happens, we'll trigger a new `addReview()` method. This will add product reviews to our `product-display` component, which means that component needs a new `reviews` array in its data.

components/ProductDisplay.js

```
const component = Vue.extend({
  template:
  `<div>
    <div class="product-display">
      <div class="product-container">
      ...
    </div>
    <review-form @review-submitted="addReview">
  </div>`
  data() {
    return {
      reviews: []
    }
  },
  methods: {
    addReview(review) {
      this.reviews.push(review)
    }
  }
});
```

Now let's flesh out the `addReview()` method:

components/ProductDisplay.js

```
const component = Vue.extend({
  template:
  `<div>
    <div class="product-display">
      <div class="product-container">
      ...
    </div>
    <review-form @review-submitted="addReview">
  </div>`
  data() {
    return {
      reviews: []
    }
  },
  methods: {
    addReview(review) {
      this.reviews.push(review)
    }
  }
});
```

As you can see, it adds taken in the `review` that we get from the `review-submitted` event payload, and pushes it into the `reviews` array.

Displaying the reviews

Now that we've implemented the ability to add reviews, we need to be displaying those reviews. Let's create a new component to do that. That component will be called `review-list`, which we'll scaffold out like this:

components/ReviewList.js

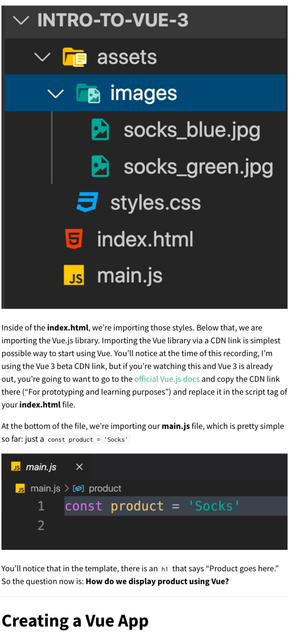
```
const component = Vue.extend({
  template:
  `<div>
    <h3>Reviews:</h3>
    <br>
    <br>
    
```

Creating the Vue App

Are you ready to create a Vue app? To get started with this lesson, you can either check out the starting code on the `L2-start` branch of the repo, or head over to CodePen to get started there.

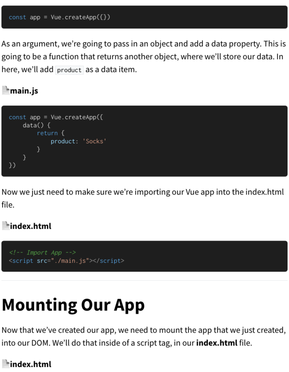
Touring the Starting Code

To take a tour of the starting code, you'll see we have an assets directory. Inside of there, there's a directory for images. We've got one for blue socks, and green socks. We also have a CSS file for all of our styles.



Inside of the **index.html**, we're importing those styles. Below that, we are importing the Vue.js library. Importing the Vue library via a CDN link is simplest possible way to start using Vue. You'll notice at the time of this recording, I'm using the Vue 3 beta CDN link, but if you're watching this and Vue 3 is already out, you're going to want to go to the official Vue.js docs and copy the CDN link there ("For prototyping and learning purposes") and replace it in the script tag of your **index.html** file.

At the bottom of the file, we're importing our **main.js** file, which is pretty simple so far: just a `const product = 'Socks'`



You'll notice that in the template, there is an `h1` that says "Product goes here." So the question now is: **How do we display product using Vue?**

Creating a Vue App

To display our data within our HTML, we'll first have to create a Vue app. In our **main.js** file, we'll create our app with:

main.js

```
const app = Vue.createApp({})
```

As an argument, we're going to pass in an object and add a data property. This is going to be a function that returns another object, where we'll store our data. In here, we'll add `product` as a data item.

main.js

```
const app = Vue.createApp({
    data() {
        return {
            product: 'Socks'
        }
    }
})
```

Now we just need to make sure we're importing our Vue app into the index.html file.

index.html

```
<!-- import App -->
<script src="./main.js"></script>
```

Mounting Our App

Now that we've created our app, we need to mount the app that we just created, into our DOM. We'll do that inside of a script tag, in our **index.html** file.

index.html

```
<!-- Mount App -->
<script>
  const mountedApp = app.mount('#app')
</script>
```

We'll say `app`, which refers to the app that we just created, and then `.mount()`, which is a method that requires a DOM selector as an argument. This lets us plug the Vue app into that piece of our DOM.

Displaying the Data

Now that we've created, imported and mounted out Vue app, we can now start displaying the data that lives within it.

To render the `product` data within the `h1`, we'll write:

index.html

```
<div id="app">
  <h1>{{ product }}</h1>
</div>
```

Now if we check the browser, we'll see "Product" is being displayed. Great! But how exactly is this working?

Understanding the Vue Instance

When we created our Vue app, we passed in the options object, which allowed us to add some optional properties to configure the application. Doing this creates our Vue instance, the heart of our Vue application, which powers everything.

main.js

```
const app = Vue.createApp({Options Object})
```

By importing this app, and mounting it to the DOM, we've essentially plugged the app into our DOM, giving our HTML a direct line into the app. This way, our template code can access options from the app, such as its data.

If you're wondering what's happening with this double curly brace syntax, you can imagine it like a phone, which has access to a phone within our Vue app. From our template, we're able to ask the app, "Hey, what's the value of product?" And the app responds, "Socks." When the page renders, we see "Socks" display on the page.

If this double curly brace syntax, or mustache syntax, is new to you, it allows us to write JavaScript expressions. In other words, it allows us to run valid JavaScript within our HTML.

Vue's Reactivity

What would happen if we changed the value of `product` from "Socks" to "Boots"?

main.js

```
const app = Vue.createApp({
    data() {
        return {
            product: 'Boots' // updated data value //
        }
    }
})
```

Because of how Vue works, the `h1`'s expression that is relying upon `product` would automatically receive that new value, and our DOM would update to display "Boots".

index.html

```
<div id="app">
  <h1>{{ product }}</h1> <! -- will reactively receive any updates to product
  -->
</div>
```

This is because Vue is *reactive*. Under the hood, Vue has an entire reactivity system that handles updates. When a data value changes, anywhere relying on that data will automatically update for us. We don't have to do anything to make that happen.

(As a side note, here on Vue Mastery, we have an entire Vue 3 reactivity course. So if you're interested in exploring the reactivity system under the hood, you can take that advanced course later.)

Coding Challenge

We've reached the end of this lesson, which brings us to the first coding challenge. You can find the solution code by checking out the `L2-end` branch of the repo, or viewing the solution on Codepen.

Add a `description` to the data object.

Display the `description` using an expression within a `p` tag.

Attribute Binding

In this lesson, we're going to look at the concept of attribute binding. If you're coding along with the repo, you can checkout the `L3-start` branch or the starting code on Codepen.

Our Goal

In the starting code, we have a new `div` with the class of `product-image`.

index.html

```
<div class="product-image">
  <!-- image goes here -->
</div>
```

By the end of the lesson, we're going to have an `img` element here that is reactively bound to a new `image` property on our data. Whenever the value of that image changes, our image will update in the DOM.

Adding an image to our data

Remember in our **assets** directory we have an **images** folder, with images for green and blue socks? Let's target one of those images from a new data property on our Vue app. We'll do so by setting `image` equal to a path so it can grab that image.

main.js

```
const app = Vue.createApp({
    data() {
        return {
            product: 'Socks',
            image: './assets/images/socks_green.jpg'
        }
    }
})
```

Now we're ready to add an `img` element in the template.

index.html

```
<div class="product-image">
  <img src="image">
</div>
```

In the source attribute, we'll say `image`. Right now, this won't do anything. We want `src` to pull in the `image` path from our data, similar to how we pulled in the `product` data value in the `h1` expression in the previous lesson.

So the question here is: **How do we bind the `src` attribute to the `image` data?**

Introducing Attribute Binding

To create a bond between an HTML element's attribute and a value from your Vue app's data, we'll use a Vue directive called `v-bind`.

index.html

```
<img v-bind:src="image">
```

Now, we've created a reactive bond between what lives in this attribute (`"image"`) and the `image` data itself.

Looking in the browser, we'll now see our green socks image displaying.

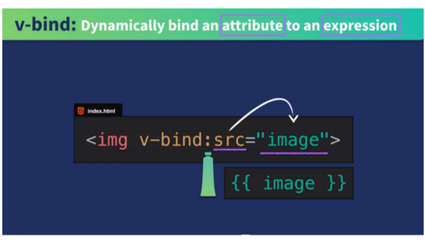
Understanding v-bind

How exactly is the `v-bind` directive working? We use this directive to dynamically bind an attribute to an expression. In this case, the attribute is `src` and the expression is whatever is in that attribute's quotes: `"image"`

index.html

```
<img v-bind:src="image"> <! -- src attribute bound to the image data -->
```

If you're thinking this doesn't look like a typical JavaScript expression, you can imagine it looking like this: `v-bind:src="{{ image }}"`. Under the hood, Vue is going to evaluate it just the same.



A Reactive Bond

Because of Vue's reactivity system, if we updated our `image` data to a path that points to the image of our blue socks (`image: './assets/images/socks_blue.jpg'`), the expression that our `src` attribute is bound to would update and our browser would display the blue socks image.

A shorthand for v-bind

Using `v-bind` is super common—so common that there is a shorthand for it, and it's just the colon, like so:

```
<img :src="image">
```

As you can imagine, since there's so many different HTML attributes, there are many use cases for `v-bind`. For example, you might be binding a description to an `alt` attribute, binding a URL to an `href`, binding some dynamic styles to a `class` or `style` attribute, disabling and enabling a button, and so on.

Coding Challenge

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

Add a `url` to the data object

Bind the `url` to an anchor tag's `href` attribute

As a reminder, if you're coding along with our repo, you can check out `L3-end` branch, and you can view the solution code on Codepen.

Conditional Rendering

In this lesson, we're going to look at the concept of conditional rendering. If you're coding along with the repo, you can checkout the `L4-start` branch or use the starting code on Codepen.

Our Goal

We want to display different HTML elements based upon a condition. We'll display a `p` tag that says "in stock" when our product is in stock, or one that says "out of stock" when it's not.

To render or not to render

In our index.html file, we'll add two new `p` tags.

index.html

```
<p>In Stock</p>
<p>Out of Stock</p>
```

We only want one of these to show up depending on if our product is in stock or not, so we'll head into our Vue app's data object and add a new `inStock` boolean.

main.js

```
const app = Vue.createApp({
    data() {
        return {
            product: 'Socks',
            image: './assets/images/socks_blue.jpg',
            inStock: true // new data property //
        }
    }
})
```

Now that we've added the elements we want to conditionally render, and the condition (`inStock`) that we'll use to decide which one to render, we're ready to learn about another Vue directive.

The v-if directive

We can add the `v-if` directive onto an element to render it based upon a condition, like so:

```
<p v-if="inStock">In Stock</p>
```

Now, this element will render only *if* `inStock` is truthy.

We can combine the `v-if` directive with its sister directive `v-else` to display another element as the fallback if the first condition turns out to be falsey.

index.html

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

Now, if `inStock` is `false`, we'll see "Out of Stock" gets rendered to the page.

Show and Hide

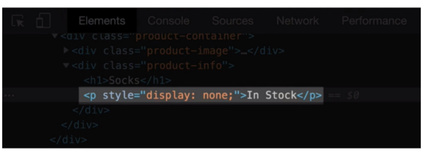
It's worth noting that you don't always need to pair `v-if` with `v-else`. There are plenty of use cases where you don't need a fallback element to render. However, in these cases, it is sometimes a better option to use the `v-show` directive.

index.html

```
<p v-show="inStock">In Stock</p>
```

The `v-show` directive is used for toggling an element's **visibility** instead of adding and removing the element from the DOM entirely, like `v-if` does.

As you might imagine, this is a more performant option if you have something that's toggling off and on the screen often. We can verify this by setting `inStock` to `false` and viewing the element in the browser's Developer Tools. When `v-show` is used, we can see that the element is still present in the DOM, but it's now hidden with an inline style of `display: none;` added to it.



Chained Conditional Logic

Earlier, we looked at `v-if` with `v-else`, now let's take a look at how we can add additional layers of conditional logic.

To do this, we'll replace `inStock` with `inventory`:

main.js

```
const app = Vue.createApp({
    data() {
        return {
            ...
            inventory: 100
    }
```

Since our condition (`inventory`) is now an integer, we can use a bit more complex logic within our expression. For example:

index.html

```
<p v-if="inventory > 10">In Stock</p>
<p v-else>Out of Stock</p>
```

Now, we will only render the first `p` tag if `inventory` is greater than `10`.

Let's say we want display a new message when the product is almost sold out. In this situation, we could add another conditional level, where we're watching out for `inventory` to get below `10` but above `0`.

index.html

```
<p v-if="inventory > 10">In Stock</p>
<p v-else-if="inventory <= 10 && inventory > 0">Almost sold out!</p>
<p v-else>Out of Stock</p>
```

The `v-else-if` directive gives us a middle layer of logic. So in this example, if `inventory` were `8`, this `p` tag would get rendered.

Of course, if `inventory` is zero, we'll default to the final level of `v-else` and display "Out of stock".

Coding Challenge

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

Add an `onSale` Boolean to the data object.

Use `onSale` to conditionally render a `p` tag that says "On Sale," whenever `onSale` is `true`.

As a reminder, if you're coding along with our repo, you can check out `L4-end` branch, and you can view the solution code on Codepen.

List Rendering

In this lesson, we're going to look at the concept of list binding. If you're coding along with the repo, you can checkout the `L5-start` branch or view the starting code on Codepen.

Our Goal

Render HTML lists from an array in our data.

Looping through data arrays

In the starting code, we now have an array of `details`.

main.js

```
const app = Vue.createApp({
    data() {
        return {
            ...
            details: ['50% cotton', '30% wool', '20% polyester']
        }
    }
})
```

The question now is: how do we display this data as a list?

We'll start by creating an unordered list in our **index.html**. On the `li` inside of it, we'll add another Vue directive: `v-for`

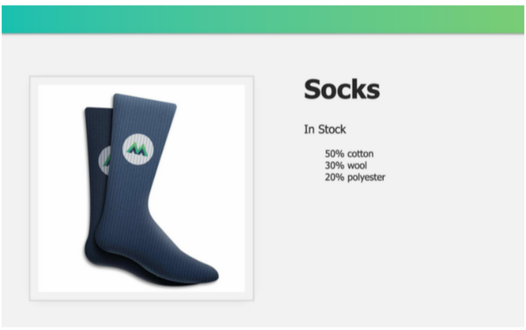
index.html

```
<ul>
  <li v-for="detail in details">{{ detail }}</li>
</ul>
```

Inside the `v-for` expression, we wrote: `detail in details`. Here, `details` refers to the `details` array in our data, and `detail` is the alias for the current element from that array, as we're looping through it to print out a new `li`.

Each `li` will display that array element because in the inner HTML we've written an expression: `{{ detail }}` to print out each detail.

If we check the browser, we'll see a list of the `details` is displayed.



So far so good, but how is `v-for` actually working?

Product Variant Colors

To get more familiar with list rendering with `v-for`, we'll work on another example within our app. Let's add a new `variants` array to our data:

main.js

```
data() {
  return {
    ...
    variants: [
      { id: 2234, color: 'green' },
      { id: 2235, color: 'blue' }
    ]
  }
}
```

We now have an array that contains an object for each variant of our product. Each product variant has an `id`, and a `color`. So for our next task, we'll print out each variant color, and use the `id` to help Vue keep track of our list items.

index.html

```
<div v-for="variant in variants" :key="variant.id">{{ variant.color }}</div>
```

Notice how we're using dot notation to print out each `variant` as we loop through the `variants` array. But what is that `:key` attribute doing there?

Key Attribute: An essential for list items

By saying `:key="variant.id"`, we're using the shorthand for `v-bind` to bind the variant's `id` to the `key` attribute. This gives each DOM element a unique key so that Vue can grasp onto the element and not lose track of it as things update within the app.

This provides some performance improvements, and later down the line, if you're doing something like animating your elements, you'll find that the `key` attribute really helps Vue effectively manage your elements as they move around the DOM.

Coding Challenge

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

Add an array of `sizes` to the data object.

Use `v-for` to display the `sizes` in a list.

As a reminder, if you're coding along with our repo, you can check out `L5-end` branch, and you can view the solution code on Codepen.

Event Handling

In this lesson, we're going to look at the concept of Event Handling. If you're coding along with the repo, you can checkout the `L6-start` branch or the starting code on Codepen.

In the starting code, you'll see that we now have an Add to Cart `button`, along with a cart `div`, which includes an expression to print out the value of our new `cart` data.

index.html

```
<div class="cart">Cart({{ cart }})</div>
...
<button class="button">Add to Cart</button>
```

main.js

```
data() {
  return {
    cart: 0,
    ...
  }
}
```

Our Goal

We want to be able to click the `button` and increment the value of `cart`.

Listening for Events

In order to know when the button is clicked, we need to be listening for events on that element, specifically *click* events. We can achieve this by using another Vue directive: `v-on`.

index.html

```
<button class="button" v-on:click="logic to run">Add to Cart</button>
```

Here, we are telling `v-on` what type of event to listen for: a `click`. Inside the quotes, we place the logic (or method name) we want to run when that event happens.

If we write `v-on:click="cart += 1"`, we'll increment the value of cart by `1`, when a click event happens.

Triggering a method

Because the logic `cart += 1` is very simple, we could keep it in-line on the `button` element, like we have it. But often, we need to trigger more complex logic. In those situations, we can add a method name to fire when the event happens. So let's do that now.

index.html

```
<button class="button" v-on:click="addToCart">Add to Cart</button>
```

Now, when the button is clicked, the `addToCart` method will be run. Let's add that method to our Vue app's options object, like so:

```
const app = Vue.createApp({
  data() {
    return {
      cart: 0,
      ...
    }
  },
  methods: {
    addToCart() {
      this.cart += 1
    }
  }
})
```

Notice how we added the `methods` option, and inside of that we added the new `addToCart` method, which contains the same logic we just had in-line. The difference here is now we're saying `this.cart` to refer to *this* `cart` in *this* Vue instance's data.

In the browser, we should now be able to click the **Add To Cart** `button` and see the value of cart climb up by 1.

Understanding v-on

Let's take a deeper look into how this event handling is working.

By adding `v-on` to an element, we're essentially giving it an ear that can listen for events. In this case, we've specified that we're listening for click events. When a click happens, the `addToCart` method runs, which as we just saw, takes the value of cart and increments it by one.

A shorthand for v-on

As you can imagine, listening for events on your elements is super common. Just like how `v-bind` had a shorthand (`:`), `v-on` has a shorthand: `@`

So our code could be simplified to:

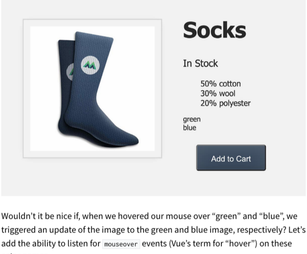
index.html

```
<button class="button" @click="addToCart">Add to Cart</button>
```

Another Example: Mouseover Events

Now that we understand the basics of event handling, let's listen for another kind of event within our Vue app.

Currently we're displaying the variant colors, "green" and "blue", just below the product details:



Wouldn't it be nice if, when we hovered our mouse over "green" and "blue", we triggered an update of the image to the green and blue image, respectively? Let's add the ability to listen for `mouseover` events (Vue's term for "hover") on these color names.

Because we want to update the image that we're displaying when we mouse over the variant colors, I've added a new property to each variant object.

main.js

```
data() {
  return {
    ...
    variants: [
      { id: 2234, color: 'green', image: './assets/images/socks_green.jpg' },
      { id: 2235, color: 'blue', image: './assets/images/socks_blue.jpg' },
    ]
  }
}
```

Now each variant has an image path for the green and blue socks, respectively. We're ready to add a listener for `mouseover` events on the variant color `div`.

main.js

```
<div v-for="variant in variants" :key="variant.id"
@mouseover="updateImage(variant.image)">{{ variant.color }}</div>
```

When a `mouseover` event happens, we're triggering the `updateImage` method, passing in the image path of each variant. That method looks like this:

```
methods: {
  ...
  updateImage(variantImage) {
    this.image = variantImage
  }
}
```

It expects the `variantImage` as the parameter, and when it's run, it sets `this.image` (in *this* Vue instance's data) equal to the variant image that was passed in.

Now in the browser, when we hover our mouse over "green", we should see the green image. When we hover over "blue", we should see the blue image.

Coding Challenge

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

Create a new button that decrements the value of cart.

As a reminder, if you're coding along with our repo, you can check out `L6-end` branch, and you can view the solution code on Codepen.

Class & Style Binding

In this lesson, we're going to look at the concept of class and style binding. If you're coding along with the repo, you can checkout the `13-start` branch or the starting code on Codepen.

Our Goal

Bind classes and styles to our elements based on our app's data.

Style Binding

In the last lesson, we added the feature where if you hover over "green" or "blue", you update the image that is being displayed; the green or blue socks, respectively. But wouldn't the user experience be nicer if instead of hovering over the "word" green or "blue", we hovered over the actual colors green and blue?

Let's create green and blue circles that we can hover on. We can achieve this by using style binding.

First, to style our div-like circles, we'll we need to add a new class `.color-circle` to the variant div.

index.html

```
<div
  v-for="variant in variants"
  :key="variant.id"
  @mouseover="updateImage(variant.image)"
  class="color-circle"
>
</div>
```

This class already lives in our css file. As you can see, it simply transform our divs into a circle with a 30px diameter.

styles.css

```
.color-circle {
  width: 30px;
  height: 30px;
  margin-top: 8px;
  background: red;
  border: 1px solid #d8d8d8;
  border-radius: 50;
}
```

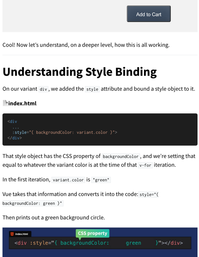
Now that we've got that out of the way, we can move on to the actual style binding. Just like it sounds, we want to bind styles to the variant divs. We do so by using `v-bind` (or its shorthand `:`) on the `style` attribute, and binding a style object to it.

index.html

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

Here, we're setting the divs' `backgroundColor` equal to the `variant.color`. So instead of printing out those strings, "green" and "blue", we're using them to set the background-color of our circles.

Checking this out in the browser, we should now see two color circles filled in with a green and blue background.



Cool! Now let's understand, on a deeper level, how this is all working.

Understanding Style Binding

On our variant `div`, we added the `style` attribute and bound a style object to it.

index.html

```
<div
  :style="{ backgroundColor: variant.color }"
>
</div>
```

That style object has the CSS property of `backgroundColor`, and we're setting that equal to whatever the variant color is at the time of that `v-for` iteration.

In the first iteration, `variant.color` is "green".

Vue takes that information and converts it into the code: `style="{ backgroundColor: green }"`

Then prints out a green background circle.

It repeats this process for the second variant color to create the blue circle.

Camel vs Kebab

There are some important things to consider when using style binding like this.

```
:style="{ backgroundColor: variant.color }"></div>
```

Inside of this expression, remember that this style object is all JavaScript. That's why I used camelCase in the property name. If I had said `background-color`, that `-` would've been interpreted as a minus sign. But we're not doing any math here. We're setting a CSS property name.

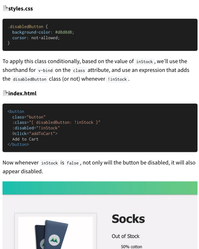
So since we're in this JavaScript object, we have to use camelCased unless we want to use 'kebab-cased' in quotes to avoid the mathematical misinterpretation, like so:

```
:style="{ 'background-color': variant.color }"></div>
```

Both options will work, as long as you remember your quotation marks.

Style Binding: Objects

Sometimes you might want to add a bunch of styles to an element, but adding them all in-line could get messy. In these situations, we can bind to an entire style object that lives within our data.



Now that we've taken a look into the topic of style binding, let's look at a similar topic: class binding.

Class Binding

Back in our app, you'll notice that when our `inStock` data value is false, we can still click the Add to Cart button and increment the value of the cart. But if the product is out of stock, maybe we don't want the user to be able to add the product to the cart. So let's change up this behavior, disabling the button whenever `inStock` is `false` AND making the button appear disabled, using class binding.

To get this started, we'll use the shorthand for `v-bind` on the `disabled` attribute to add that attribute whenever our product is not in stock.

index.html

```
<button
  class="button"
  :disabled="!inStock"
  @click="addToCart">
  Add to Cart
</button>
```

Now, whenever `inStock` is `false` and we click the Add to Cart button, nothing will happen since it's disabled. But the button still appears active, which is misleading to our users. So let's use class binding to add a `disabledButton` class as well, whenever `inStock` is `false`.

You'll see in our CSS file that we already have this `disabledButton` class, which sets the `background-color` to gray and makes the `cursor` not-allowed.

styles.css

```
.disabledButton {
  background-color: #d8d8d8;
  cursor: not-allowed;
}
```

To apply this class conditionally, based on the value of `inStock`, we'll use the shorthand for `v-bind` on the `class` attribute, and use an expression that adds the `disabledButton` class (or not) whenever `!inStock`.

index.html

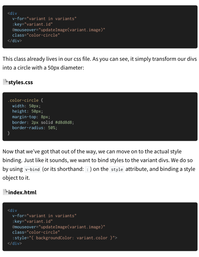
```
<button
  class="button"
  :class="{ disabledButton: !inStock }"
  :disabled="!inStock"
  @click="addToCart">
  Add to Cart
</button>
```

Now whenever `inStock` is `false`, not only will the button be disabled, it will also appear disabled.

Multiple Class Names

When getting started with class binding, there are some things to note. For example, what happens when we already have an existing class and we want to conditionally add another class based on a data value?

For example, if we already have the `color-circle` class on this `div`, and we conditionally add the `active` class, how will this look?



Those classes are going to be combined like so:

```
<div class="color-circle active">
</div>
```

Ternary Operators

A helpful tool that class binding gives us is the ability to use in-line ternary operators to add different classes based upon a condition.

In this case, because `isActive` is `true`, we are indeed adding the `activeClass`. If it were `false`, we'd add no class (`''`); alternatively, we could have added an entirely different class.

The variations in syntax and use cases that I just showed you with class and style binding is only the start. So I recommend checking out the Vue docs for more use cases and examples.

Coding Challenge

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

Bind the `out-of-stock-img` class to the image whenever `inStock` is `false`.

As a reminder, if you're coding along with our repo, you can check out `13-end` branch, and you can view the solution code on Codepen.

Computed Properties

In this lesson, we're going to look at the concept of computed properties. If you're coding along with the repo, you can checkout the `L8-start` branch or the starting code on Codepen.

Our Goal

Update both the variant image AND whether it's in stock or not, using computed properties.

A Simple Computed Property

In the starting code, you'll notice we have a new data property:

main.js

```
data() {
  return {
    product: 'Socks',
    brand: 'Vue Mastery'
  }
```

What if we wanted to combine the `brand` and the `product`, in our template? We could do that within an expression like so:

index.html

```
<h1>{{ brand + ' ' + product }}</h1>
```

If we checked this out in the browser, we'd see "Vue Mastery Socks" displayed. But wouldn't it be neat if, instead of handling this logic in the inner HTML, our app had the ability to compute that value for us? For example, taking the `brand` and the `product`, adding them together, and returning that new value.

Computed properties are exactly like they sound: properties we can add to a Vue app that compute values for us. They help us keep computational logic out of the template and give us performance improvements that we'll cover soon. For now, let's turn this simple example into a computed property. We'll alter the `h1`'s expression like so:

index.html

```
<h1>{{ title }}</h1>
```

Now, `title` is the name of a computed property that we'll create now. First, we'll add the `computed` option to the app, just below our `methods`, then create the `title` property.

main.js

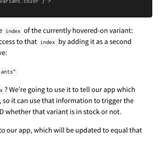
```
...
computed: {
  title() {
    return this.brand + ' ' + this.product
  }
}
```

If we check out the browser, we'll still see "Vue Mastery Socks" displayed, except now we've abstracted that computational logic out of the template and contained it neatly on the options object.

But how exactly are computed properties working? Let's take a deeper look.

Think of them like a Calculator

I like to think of computed properties kind of like a calculator, because they calculate or *compute* values for us. This computational calculator takes our values, `brand` and `product`, adds them together, and gives us the result.



Like I mentioned earlier, computed properties provide us a performance improvement. This is because they cache the calculated value. The value (`'Vue Mastery Socks'`) gets stored away and only updates when it needs to, when one of its dependencies change. For example, if the `brand` were to change from `'Vue Mastery'` to `'Node Mastery'`, our computed property would receive that new `brand` dependency, then recalculate and return the new value: `'Node Mastery Socks'`

Now that we're starting to comprehend computed properties, let's implement a more practical example in our Vue app.

Computing Image & Quantity

Heading back into our code, let's add a new `quantity` property to our variant objects.

main.js

```
data() {
  return {
    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 },
    ]
  }
}
```

Notice how the green socks have a `quantity` of `50` while the blue socks have `0`. In other words, the green socks are in stock and the blue socks are out of stock. However, we're currently displaying "In stock" or "Out of stock" based on the `inStock` data value, which no longer reflects the truth about our product and its variant quantities. So we'll want to create a computed property we can use to display "In stock" or "Out of stock" based on these new quantities.

To get started, remember how we updated the variant image, based on which variant color is moused over? Instead of that mouseover event triggering the `updateImage()` method, we're going to have it trigger a new method called `updateVariant()`.

index.html

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

Notice how we're passing in the `index` of the currently hovered-on variant: `updateVariant(index)`. We got access to that `index` by adding it as a second parameter in our `v-for` directive:

`v-for="(variant, index) in variants"`

Why are we passing in the `index`? We're going to use it to tell our app which variant is currently hovered on, so it can use that information to trigger the updating of both the image AND whether that variant is in stock or not.

We'll add a new data property to our app, which will be updated to equal that `index`

main.js

```
data() {
  return {
    ...
    selectedVariant: 0,
    ...
  }
}
```

Our `updateVariant()` method will set the `selectedVariant`'s value equal to the `index` of the currently hovered-on variant.

main.js

```
updateVariant(index) {
  this.selectedVariant = index
}
```

Now, we've implemented a way for our app to know which product variant is being engaged with, and we're able to use that information to trigger the computing of which image to show and whether to show "In stock" or "Out of stock", based on which variant the user is moused over.

We're now ready to delete `image` and `inStock` from our data, and replace those with computed properties of the same names.

main.js

```
computed: {
  image() {
    return ??
  },
  inStock() {
    return ??
  }
}
```

So how do we grab the variant image and quantity? That will look like this:

main.js

```
image() {
  return this.variants[this.selectedVariant].image
}
```

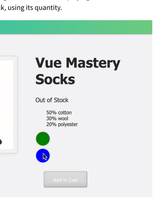
We're targeting the first or second element of our `variants` array based off the `selectedVariant`, which is either `0` or `1`, depending on which variant color circle is hovered on. Then we just use dot notation to grab the `image` off that variant.

The logic for computing `inStock` is nearly identical:

main.js

```
inStock() {
  return this.variants[this.selectedVariant].quantity
}
```

Checking this out in the browser, when we hover over the color circles, not only are we updating the variant image, but we're also displaying whether that variant is in stock or out of stock, using its quantity.



Notice how the button is still automatically updating for us, enabling and disabling. That's because, in our template, we're still using `inStock`.

index.html

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

Now `inStock` is no longer a data property; it's the new computed property.

Coding Challenge

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

Add an `onSale` boolean to the data.

Use a computed property to display the string: `'brand + ' ' product + ' ' is on sale'`, whenever `onSale` is `true`.

As a reminder, if you're coding along with our repo, you can check out `L8-end` branch, and you can view the solution code on Codepen.

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 lesson, you can checkout the `L9-start` branch or the `starting-code` on Codepen.

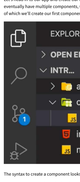
Our Goal

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:
    `<div>
      <div class="product-display">
        <div class="product-container">

          <img v-bind:src="image">
        </div>
        <div class="product-info">
          <h1>{{ title }}</h1>

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

          <p v-if="inventory > 10">In Stock</p>

          <p v-if="inventory <= 10 && inventory > 0">Almost gone!</p>

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

```