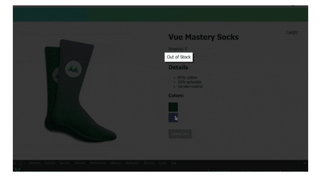
The Vue Instance

Throughout this course you will learn the fundamentals of Vue while we build this product page together.



Prerequisites:

This course assumes a foundational knowledge in HTML, CSS and JavaScript.

Our Goal

In this lesson, we'll show you how to use Vue to display data onto a webpage.

Our Starting Code

We're going to start with some very simple HTML and JS code, which looks like this:

index.html

```
<div id="app">
  <h1>Product Name</h1>
</div>

<script src="main.js"></script>
```

main.js

```
var product = "Socks";
```

Problem

We need a way to take the variable `product` from our JavaScript and have it show up in the `h1` of our HTML.

Our first step is to include Vue in our project, which we'll do by adding this line in our `index.html` file, just above the main.js script.

index.html

```
<script src="https://unpkg.com/vue"></script>
```

Next in our main.js we'll write the following:

main.js

```
var app = new Vue({
  el: '#app',
  data: {
    product: "Socks"
  }
})
```

And then in our index.html we'll use our first JavaScript expression:

index.html

```
<div id="app">
  <h2>{{ product }}</h2>
</div>
```

When we save this, we'll see "Socks" appear on our webpage.

Socks

It worked. Data from our JavaScript is showing up in our HTML. But what did we just do? Let's break it down:

The Vue Instance

main.js

```
var app = new Vue({options})
```

A Vue instance is the root of our application. It is created by passing an options object into it. Just like it sounds, this object has a variety of **optional** properties that give the instance the ability to store data and perform actions.

Plugging in to an Element

main.js

```
  el: '#app',
```

The Vue instance is then plugged into an element of your choosing, forming a relationship between the instance and that portion of the DOM. In other words, we're activating Vue on the div with the id of `app` by setting `'#app'` as the element (`el`) that our instance is plugged into.

Putting our data in its place

A Vue instance has a place for data, in its data property.

main.js

```
  data: {
    product: "Socks"
  }
```

The instance's data can be accessed from inside the element that the instance is plugged into.

Using an Expression

If we want our `product` to appear in our `h1` , we can put `product` inside these double curly braces.

index.html

```
<h1>{{ product }}</h1>
```

Socks

See? It works. Simple huh?

How does it work? Inside the double curly braces, we're using a JavaScript expression.

Important Term: Expression

Expressions allow us to utilize existing data values, together with logic, to produce new data values.

When Vue sees the expression `{{ product }}` , it recognizes that we are referencing the associated Vue instance's data, and it replaces that expression with the **value** of product instead, in this case: "Socks".

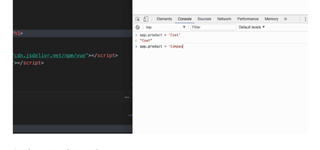
Some other ways expressions can be used:

```
{{ product + '?' }}
{{ firstName + ' ' + lastName }}
{{ message.split('').reverse().join('') }}
```

Introducing Reactivity

The reason Vue is able to display `product` 's value immediately is because Vue is **reactive**. In other words, the instance's data is linked to every place that data is being referenced. So not only can Vue make its data appear within the HTML that references it, but that HTML will be updated to display new values any time that referenced data changes.

To prove that, let's open the console and change the value of our product string. Look what happens.



See how easy that was?

What have we learned?

- How to begin writing a Vue application with a Vue instance, and how to load basic data onto the webpage.
- The Vue instance is the root of every Vue application
- The Vue instance plugs into an element in the DOM
- The Vue instance's data can be displayed using the mustache-like syntax `{{ }}` called an expression.
- Vue is reactive

Learn by doing

Challenge

Add a `description` key to our existing data object with the value "A pair of warm, fuzzy socks". Then display `description` using an expression in an `p` element, underneath the `h1` .

Forms & v-model

In this lesson we'll be learning how to work with forms in Vue in order to collect user input, and also learn how to do some custom form validation.

Goal

We'll be creating a form that allows users to submit a review of a product, but only if they have filled out the required fields.

Starting Code

Our app now looks like this:

index.html

```
<div id="app">
  <h1>{{ page }}</h1>
  <ul>{{ product.name.length }}</ul>
  <ul>
    <li v-for="product in products">{{ product.name }}</li></ul>
</div>
```

main.js

```
Vue.component('product', {
  data: function,
  { product: '',
    id: 0,
    type: 'Boots',
    reviews: [],
  },
  template: `
  <div>
    <h1>{{ product }}</h1>

    <div class="product-image">
      
    </div>

    <div class="product-info">
      <h1>{{ product }}</h1>
      <p v-if="inStock">In Stock</p>
      <p v-if="!inStock">Out of Stock</p>
      <p>Shipping: {{ shipping }}</p>

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

      <div class="color-box"
        v-for="(variant, index) in variants"
          :key="variant.variantId"
          :style="{ backgroundColor: variant.variantColor }"
          @mouseover="updateProduct(index)"
        {{ variant.variantColor }}
      </div>

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

    </div>

  </div>`,

  data() {
    return {
      product: 'Socks',
      brand: 'Vue Mastery',
      selectedVariant: 0,
      details: [ 80% cotton, 20% polyester, 'Gender-neutral' ],
      variants: [
        {
          variantId: 2234,
          variantColor: 'green',
          variantImage: './assets/vmSocks-green.jpg',
          variantQuantity: 10
        },
        {
          variantId: 2235,
          variantColor: 'blue',
          variantImage: './assets/vmSocks-blue.jpg',
          variantQuantity: 0
        }
      ],
    }
  },
  methods: {
    addToCart() {
      this.$emit('add-to-cart')
    },updateProduct(variantIndex) {
      this.selectedVariant = variantIndex
    },
      updateProduct(index) {
      this.selectedVariant = index
    },
  },
  computed: {
    title() {
      return this.brand + ' ' + this.product
    },
    image() {
      return this.variants[this.selectedVariant].variantImage
    },
    inStock(){
      return this.variants[this.selectedVariant].variantQuantity
    },
    shipping() {
      if (this.premium) {
        return "Free"
      } else {
        return 2.99
      }
    }
  }
})

var app = new Vue({
  el: '#app',
  data: {
    premium: true,
    cart: []
  },
  methods:
    addToCart: function
    this.cart.push(id)
  }
})
```

Problem

We want our users to be able to review our product, but we don't yet have a way to collect user input. We'll need to create a form for that.

Solution

We'll create a new component for our form, which will be called `product-review` since it is the component that collects product reviews. `product-review` will be nested within our `product` component.

Let's register our new component, start building out its template, and give it some data.

```
Vue.component('product-review', {
  template:
  `
  data() {
    return {
      name: null,
    }
  }
})
```

As you can see, our component has an input element, and `name` within its data.

How can we bind what the user types into the input field to our `name` data?

Earlier we learned about binding with `v-bind` but that was only for one-way binding, **from the data** to the template. Now, we want whatever the user inputs to be bound to `name` in our data. In other words, we want to add a dimension of data binding, from the template **to the data**.

The v-model directive

Vue's `v-model` directive gives us this two-way binding. That way, whenever something is entered into the input, the data changes. And whenever the data changes, anywhere using that data will update.

So let's add `v-model` to our input, and bind it to `name` in our component's data.

```
<input v-model="name">
```

So far so good. Let's add a complete form to our template.

```
<form class="review-form" @submit.prevent="onSubmit">

  <p>
    Name:
    <input id="name" v-model="name" placeholder="name">
  </p>

  <p>
    Review:
    <textarea id="review" v-model="review"></textarea>
  </p>

  <p>
    Would you recommend this product?
    <label id="recommend-yes">Yes<input type="radio"
    value="Yes" name="recommend" />
    <label id="recommend-no">No<input type="radio"
    value="No" name="recommend" />
    <br />
    Rating:
    <input id="rating" v-model="rating" />
  </p>

  <p>
    <input type="submit" value="Submit">
  </p>

</form>
```

As you can see, we've added `v-model` to the `input`, `textarea` and `select`. Note on the `select` we've added the `.number` modifier (more on this below). This ensures that the data will be converted into an integer versus a string.

These elements are now bound to this data:

```
data() {
  return {
    name: null,
    review: null,
    rating: null
  }
},
```

At the top of the form, you can see that our `onSubmit` method will be triggered when this form is submitted. We'll build out the `onSubmit` method in a moment. But first, what's that `.prevent` doing?

That is an **event modifier**, which is used to prevent the form from reloading our page. There are several other useful `v-on` event modifiers, which we won't cover in this lesson.

We're now ready to build out our `onSubmit` method. We'll start out with this:

```
onSubmit: {
  let productReview = {
    name: this.name,
    review: this.review,
    rating: this.rating
  }
  this.name = null
  this.review = null
  this.rating = null
},
```

As you can see, `onSubmit` is creating an object of our user inputted data, stored within a variable called `productReview`. We're also resetting the values of `name`, `review` and `rating` to be null. But we're not done yet. We need to send this `productReview` to a child of `product`. As we learned in the previous lesson, we can use `emit` to send up data from our child to our parent when an event occurs.

So let's add `emit` to our `onSubmit` method:

```
onSubmit: {
  let productReview = {
    name: this.name,
    review: this.review,
    rating: this.rating
  }
  this.$emit('review-submitted', productReview)
  this.name = null
  this.review = null
  this.rating = null
},
```

We're now emitting an event announcement by the name of `"review-submitted"`, and passing along with it the `productReview` object we just created.

Now we need to listen for that announcement on `product-review`.

```
<product-review @review-submitted="addReview">product-review
```

This translates to: when the `"review-submitted"` event happens, run `product`'s `addReview` method.

That method looks like this:

```
addReview(productReview) {
  this.reviews.push(productReview)
},
```

This function takes in the `productReview` object emitted from our `onSubmit` method, then pushes that object into the form `reviews` array on our `product` component's data. We don't yet have `reviews` on our product's data, so all that's left is to add that now:

```
reviews: [],
```

Awesome. Our form elements are bound to the `product-review` component's data, that data is created into a `productReview` object, and that `productReview` is being sent up to `product` when the form is submitted. Then that `productReview` is added to `product`'s `reviews` array.

Displaying the Reviews

Now all that's left to do is display our reviews. We'll do so in our `product` component, just above where the `product-review` component is nested.

```
<div>
  <h2>Reviews</h2>
  <p v-if="!reviews.length">There are no reviews yet.</p>
  <ul>
    <li v-for="review in reviews">
      <p>{{ review.name }}</p>
      <p>Rating: {{ review.rating }}</p>
      <p>{{ review.review }}</p>
    </li>
  </ul>
</div>
```

Here, we are creating a list of our reviews within `v-for` and printing them out using dot notation. Once each review is an object.

In the `p` tag, we're checking if the `review` array has a length. Has any `productReview` objects in it), and if it does not, we'll display: `"There are no reviews yet."`

Form Validation

Often with forms, we'll have required fields. For instance, we wouldn't want our user to be able to submit a review if the field they were supposed to write their review in is empty.

Fortunately, HTML5 provides you with the `required` attribute, like so:

```
<input required >
```

This will provide an automatic error message when the user tries to submit the form if that field is not filled in.

Name:

Please fill out this field.

While it is nice to have form validation handled natively in the browser, instead of in your code, sometimes the ways that the native form validation is happening may not be the best for your use-case. You may prefer writing your own custom form validation.

Custom Form Validation

Let's take a look at how you can build out your own custom form validation with Vue.

In our `product-review` component's data we'll add an array for errors:

```
data() {
  return {
    name: null,
    review: null,
    rating: null,
    errors: []
  }
},
```

We want to add an error into that array whenever one of our fields is empty. So we can say:

```
if(!this.name) this.errors.push("Name required.")
if(!this.review) this.errors.push("Review required.")
if(!this.rating) this.errors.push("Rating required.")
```

This translates to: if our `name` data is empty, push "Name required." into our `errors` array. The same goes for our `review` and `rating` data. If either are empty, an error string will be pushed into our `errors` array.

But where will we put these lines of code?

Since we only want errors to be pushed if we don't have our `name`, `review` or `rating` data filled in, we can place this code within some conditional logic in our `onSubmit` method:

```
onSubmit() {
  if(this.name && this.review && this.rating) {
    let productReview = {
      name: this.name,
      review: this.review,
      rating: this.rating
    }
    this.$emit('review-submitted', productReview)
    this.name = null
    this.review = null
    this.rating = null
  } else {
    if(!this.name) this.errors.push("Name required.")
    if(!this.review) this.errors.push("Review required.")
    if(!this.rating) this.errors.push("Rating required.")
  }
},
```

Now, we are checking to see if we have data filled in for our `name`, `review` and `rating`. If we do, we can emit our `productReview` object, and send it up to our parent, the `product` component. Then we reset our values to null.

If we don't have `name`, `review` and `rating`, we'll push errors into our `errors` array, depending on which data is missing.

All that remains is to display these errors, which we can do with this code:

```
<p v-if="errors.length">
  <b>Please correct the following error(s):</b>
  <ul>
    <li v-for="error in errors">{{ error }}</li>
  </ul>
</p>
```

This uses the `v-if` directive to check if there are any errors. In other words, if our errors array is not empty, then this p tag is displayed, which renders out a list with `v-for`, using the `errors` array for our data.

Great. Now we've implemented our own custom form validation.

Using `.number`

Using the `.number` modifier on `v-model`, it is a helpful feature, but please be aware there is a known bug with it. If the value is blank, it will turn back into a string. The bug is at Github.com offers the solution to wrap that data in the `Number` method, like so:

```
Number(this.age)
```

What's we learn

- We can use the `v-model` directive to create two-way binding on form elements
- We can use the `.number` modifier to tell Vue to cast that value as a number, but there is a bug with it
- We can use the `.prevent` event modifier to stop the page from reloading when the form is submitted
- We can use Vue to do fairly simple custom form validation

Learn by doing

Challenge

Add a question to the form: "Would you recommend this product". Then take that response from the user via radio buttons of "yes" or "no" and add it to the `productReview` object, with form validation.

Challenge

Add a question to the form: "Would you recommend this product". Then take that response from the user via radio buttons of "yes" or "no" and add it to the `productReview` object, with form validation.

Tabs

In this lesson, we'll learn how to add tabs to our application and implement a simple solution for global state management.

Goal

We'll learn how to create tabs to display our reviews and our review form separately.

Starting Code

index.html

```
<div id="app">
  <h1>Tabs</h1>
  <textarea ref="input" />Hi!</textarea>
  <div ref="output" />
</div>
```

main.js

```
fun component product(),
  ref input = document.input,
  ref output = document.output,
  ref reviews = [],
  ref error = false,
  ref loading = false,
  ref form = {
    ref name = "",
    ref email = "",
    ref rating = 5,
    ref review = "",
    ref errors = [],
    ref loading = false,
    ref success = false,
  },
  render {
    ref input.value = product.name
    ref output.innerHTML = "
      <div>
        <h2>Product Name</h2>
        <p>$product.name</p>
        <h2>Product Description</h2>
        <p>$product.description</p>
        <h2>Product Image</h2>
        
        <h2>Product Price</h2>
        <p>$product.price</p>
        <h2>Product Reviews</h2>
        <ul>
        $(reviews | map
          fun review(),
            "<li>
              <h3>$review.name</h3>
              <p>$review.email</p>
              <p>$(range(review.rating) | map
                fun _(),
                  "★
                | join(""))</p>
              <p>$review.review</p>
              
              <h3>Product Image</h3>
              
              <h3>Product Price</h3>
              <p>$product.price</p>
              <h3>Product Reviews</h3>
              <ul>
              $(reviews | map
                fun review(),
                  "<li>
                    <h4>$review.name</h4>
                    <p>$review.email</p>
                    <p>$(range(review.rating) | map
                      fun _(),
                        "★
                      | join(""))</p>
                    <p>$review.review</p>
                    
                    <h4>Product Image</h4>
                    
                    <h4>Product Price</h4>
                    <p>$product.price</p>
                    <h4>Product Reviews</h4>
                    <ul>
                    $(reviews | map
                      fun review(),
                        "<li>
                          <h5>$review.name</h5>
                          <p>$review.email</p>
                          <p>$(range(review.rating) | map
                            fun _(),
                              "★
                            | join(""))</p>
                          <p>$review.review</p>
                          
```

Attribute Binding

In this lesson, we'll explore ways you can connect data to the attributes of your HTML elements.

Our Goal

We'll use attribute-binding in order to display an image and attach alt text to it based on values from our instance's data.

Starting Code

Currently our HTML looks like this:

index.html

```
<div id="app">

  <div class="product">

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

    <div class="product-info">
      <h1>{{ product }}</h1>
    </div>

  </div>
</div>
```

We want image data to be dynamically bound here.

We've created a `div` for the product image and the product info, in order to style them with Flexbox.

And our JavaScript looks like this:

main.js

```
var app = new Vue({
  el: '#app',
  data: {
    product: "Socks",
    image: "./assets/vmSocks-green.jpg"
  }
})
```

New image data

Notice we've added an `image` source to our data.

Problem

We want an image to show up on our page, but we need it to be dynamic. We want to be able to update that image in our data and have the image automatically update on the page. Since our `src` attribute is what pulls the image into this element, we'll need data to be bound to `src` so that we can dynamically display an image based on the data at that time.

Important Term: Data Binding

When we talk about data binding in Vue, we mean that the place where it is used or displayed in the template is directly linked, or bound to the source of the data, which is the data object inside the Vue instance.

In other words, the host of the data is linked to the target of the data. In this case, our data is hosted by the data property of our Vue instance. And we want to target that data from our `src`.

Solution

To bind the value of `image` in our data object to the `src` in our `img` tag, we'll use Vue's `v-bind` directive.

index.html

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

This evaluates to:

index.html

```
<img src="./assets/vmSocks-green.jpg" />
```

Voila! Our image appears. If the value of `image` were to change, the `src` will update to match, and the new image will appear.

main.js

```
    image: "./assets/vmSocks-blue.jpg"
```

Again, this happens because the data that lives in `image` is **bound** to our `src` attribute.

Additional Usages

We can use `v-bind` again here if we want to bind alt text data to this same `img` element.

If we add this in our data:

main.js

```
    altText: "A pair of socks"
```

We can bind that to the `alt` attribute like so:

index.html

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

In both of these cases, we've used the syntax `v-bind` and after the colon `:`, we've stated which attribute we're binding the data to, `src` and `alt` in this case.

Now whenever the `image` and `altText` data changes, that updated data will remain linked to the `src` and `alt` attributes.

This is a very commonly used feature in Vue. Because it's so common, there's a shorthand for `v-bind`, and it's just a colon `:`

main.js

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

Simple, clean, and handy.

So what have we learned?

- Data can be bound to HTML attributes.
- Syntax is `v-bind:` or `:` for short.
- The attribute name that comes after the `:` specifies the attribute we're binding data to.
- Inside the attribute's quotes, we reference the data we're binding to.

Learn by doing

Challenge:

Add a link to your data object, and use `v-bind` to sync it up with an anchor tag in your HTML. Hint: you'll be binding to the `href` attribute.

Conditional Rendering

In this lesson we'll be uncovering how to conditionally display elements with Vue.

Our Goal

We want to display text that says if our product is in stock or not, based on our data.

In Stock

Starting Code

index.html

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

  <div class="product-info">
    <h1>{{ product }}</h1>
  </div>
 </div>
</div>
```

main.js

```
data: {
    product: "Socks",
    image: "./assets/vmSocks-green.jpg",
    inStock: true
  }
```

New inStock data

Notice we've added a new data property there at the bottom: `inStock`.

Problem

Often in a web application, we want elements to appear on the page depending on if a condition is met or not. For instance, if our product is not in stock, our page should display the fact that it's out of stock.

So how could we conditionally render these elements, depending on whether our product is in stock or not?

index.html

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

Solution

Vue's solution is simple and straightforward.

Given that our data contains this new property:

main.js

```
inStock: true
```

We can use the `v-if` and `v-else` directives to determine which element to render.

index.html

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

If `inStock` is truthy, the first paragraph will render. Otherwise, the second paragraph will. In this case, since the value of `inStock` is true, the first paragraph will render.

In Stock

Great. We've used conditional rendering to display whether our product is in stock or not. Our feature is done. But let's explore conditional rendering some more before we move onto our next topic.

Additional Conditional Syntax: v-else-if

We can add a third degree of logic with `v-else-if`. To demonstrate, let's use an example that is a bit more complex.

If our data looked something like this:

main.js

```
inventory: 100
```

We could use expressions, inside the quotes, to make our conditions more specific.

```
<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 element that will render is the first element whose expression evaluates to true.

Additional Conditional Syntax: v-show

If your app needs an element to frequently toggle on and off the page, you'll want to use the `v-show` directive. An element with this directive on it will always be present in our DOM, but it will only be visible on the page if its condition is met. It will conditionally add or remove the CSS property `display: none` to the element.

This method is more performant than inserting and removing an element over and over with `v-if` / `v-else`.

index.html

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

However, in the product app we're building, using a `v-if` and `v-else` works just fine, so we'll keep that as our solution.

What'd we learn

- There are Vue directives to conditionally render elements:
  - v-if
  - v-else-if
  - v-else
  - v-show
- If whatever is inside the directive's quotes is truthy, the element will display.
- You can use expressions inside the directive's quotes.
- V-show only toggles visibility, it does not insert or remove the element from the DOM.

Learn by doing

Challenge:

Add an `onSale` property to the product's data that is used to conditionally render a `span` that says "On Sale!"

List Rendering

In this lesson, we'll learn how to display lists onto our webpages with Vue.

Our Goal

We want to be able to display a list of our product's details.

- 80% cotton
- 20% polyester
- Gender-neutral

Starting Code

Our project's code currently looks like this:

index.html

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


  <div class="product-info">
    <h1>{{ product }}</h1>
    <p v-if="inStock">In Stock</p>
    <p v-else>Out of Stock</p>
  </div>
 </div>
</div>
```

main.js

```
data: {
    product: "Socks",
    image: "./assets/vmSocks-green.jpg",
    inStock: true,
    details: ["80% cotton", "20% polyester", "Gender-neutral"]
}
```

New details

What's new here is our array of `details` at the bottom.

Problem

We want our page to display our product's `details`. How can we iterate through this array to display its data?

main.js

```
details: ["80% cotton", "20% polyester", "Gender-neutral"]
```

Solution

Another Vue directive to the rescue. The `v-for` directive allows us to loop over an array and render data from within it.

index.html

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

- 80% cotton
- 20% polyester
- Gender-neutral

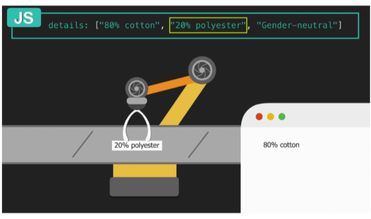
Now we have our details showing up in a list. But how is this working?

The syntax within the quotes of the `v-for` directive may look familiar if you have used JavaScript's `for of` or `for in` before. The `v-for` works like this:

We use a singular noun (`detail`) as an **alias** for the string in the array we are iterating over. We then say `in` and name the **collection** (`details`) that we are looping through. Inside the double curly braces, we specify what data to be displayed there(`{{ detail }}`).

Since our v-for is inside an `<li>`, Vue will print out a new `<li>` for each detail in our `details` array. If our `v-for` was inside a `<div>`, then a `<div>` would have been printed out for each array item along with its content.

You can envision `v-for` like an assembly line, where a mechanical arm that takes an element from the collection one at a time in order to construct your list.



Let's take a look at a more complex example, displaying an object's data in a div.

Iterating Over Objects

Problem

The product page we're building needs to be able to show different versions of the same product, based on an array in our data of `variants`. How would we iterate through this array of objects to display its data?

main.js

```
  variants: [
    {
      variantId: 2234,
      variantColor: "green"
    },
    {
      variantId: 2235,
      variantColor: "blue"
    }
  ]
}
```

Let's display the color of each variant. We'll write:

index.html

```
<div v-for="variant in variants">
  <p>{{ variant.variantColor }}</p>
</div>
```

In this case, we just want to display the color from the variant object, so we're using dot notation to do so. If we wrote `{{ variant }}` we'd display the entire object.

Note that it is recommended to use a special key attribute when rendering elements like this so that Vue can keep track of each node's identity. We'll add that in now, using our variant's unique `variantId` property.

index.html

```
<div v-for="variant in variants" :key="variant.variantId">
  <p>{{ variant.variantColor }}</p>
</div>
```

What'd we learn

- The `v-for` directive allows us to iterate over an array to display data.
- We use an alias for the element in the array being iterated on, and specify the name of the array we are looping through. Ex: `v-for="item in items"`
- We can loop over an array of objects and use dot notation to display values from the objects.
- When using `v-for` it is recommended to give each rendered element its own unique key.

Learn by doing

Challenge:

Add an array of `sizes` to the data and use `v-for` to display them in a list.

Event Handling

In this lesson we'll be learning how to listen for DOM events that we can use to trigger methods.

Goal

We want to have a button that increments the number of items in our cart.

Starting Code

index.html

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

    <div class="product-info">
      <h1>{{ product }}</h1>
      <p v-if="inStock">In Stock</p>
      <p v-else>Out of Stock</p>

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

      <div v-for="variant in variants" :key="variant.variantId">
        <p>{{ variant.variantColor }}</p>
      </div>

    </div>
  </div>
</div>
```

main.js

```
var app = new Vue({
  el: '#app',
  data: {
    product: "Socks",
    image: "./assets/vmSocks-green.jpg",
    inStock: true,
    details: ["80% cotton", "20% polyester", "Gender-neutral"],
    variants: [
      {
        variantID: 2234,
        variantColor: "green"
      },
      {
        variantId: 2235,
        variantColor: "blue"
      }
    ]
  }
})
```

Problem

We need a button to listen for click events on it, then trigger a method when that click happens, in order to increment our cart total.

First, we'll add a new data property for our `cart`.

main.js

```
    cart: 0
```

In our HTML, we'll create a `div` for our cart. We'll add a `p` inside it to display our `cart` data's value.

index.html

```
    <div class="cart">
      <p>Cart({{ cart }})</p>
    </div>
```

We'll also make a `button` to add items to our `cart`.

index.html

```
    <button v-on:click="cart += 1">Add to cart</button>
```

As you can see, we're using Vue's `v-on` directive to increment the value of `cart`

This works. But how is it working?

Let's dissect this syntax. We say `v-on`, which let's Vue know we're listening for events on this button, and after the `:` we specify the kind of event we are listening for, in this case: a click. Inside the quotes, we're using an expression that adds 1 to the value of `cart` every time the button is clicked.

This is simple, but not entirely realistic. Rather than using the expression `cart += 1`, let's make the click trigger a method that increments the value of `cart` instead, like this:

index.html

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

As you can see, `addToCart` is the name of a method that will fire when that click event happens. We haven't yet defined that method, so let's do that now, right on our instance.

Just like it does for its data, the Vue instance has an optional property for methods. So we'll write out our `addToCart` method within that option.

main.js

```
  methods: {
    addToCart() {
      this.cart += 1
    }
  }
```

Now, when we click our `button`, our `addToCart` method is triggered, which increments the value of `cart`, which is being displayed in our `p` tag.

Let's break this down further.

Our `button` is listening for click events with the `v-on` directive, which triggers the `addToCart` method. That method lives within the methods property of the Vue instance as an anonymous function. The body of that function adds 1 to the value of `this.cart`. Because `this` refers to the data of the instance we're currently in, our function is adding 1 to the value of `cart`, because `this.cart` **is** the `cart` inside our data property.

If we just said `cart += 1` here, we'd get an error letting us know that "cart is not defined", so we use `this.cart` to refer to the `cart` from `this` instance's data.

You might be thinking, "But wait. We're only incrementing the number of items in the cart, we're not actually adding a product to the cart." And you're right. We'll build that out in a future lesson.

Now that we've learned the basics of event handling in Vue, let's look at a more complex example.

First, let's add a `variantImage` to each of our variants.

main.js

```
    variants: [
      {
        variantId: 2234,
        variantColor: "green",
        variantImage: "./assets/vmSocks-green.jpg"

      },
      {
        variantId: 2235,
        variantColor: "blue",
        variantImage: "./assets/vmSocks-blue.jpg"
      }
    ],
```

Now each variant has an image with green and blue socks, respectively.

Problem

We want to be able to hover our mouse over a variant's color and have its `variantImage` show up where our product image currently is.

Solution

We'll use the `v-on` directive again, but this time we'll use its shorthand `@` and listen for a `mouseover` event.

index.html

```
      <div v-for="variant in variants" :key="variant.variantId">
        <p @mouseover="updateProduct(variant.variantImage)">
          {{ variant.variantColor }}
        </p>
      </div>
```

Notice that we're passing `variant.variantImage` in as an argument to our `updateProduct` method.

Let's build out that method.

main.js

```
    updateProduct(variantImage) {
      this.image = variantImage
    }
```

This is very similar to what we did to increment the value of `cart` earlier.

But here, we are updating the value of `image`, and its updated value is now the `variantImage` from the variant that was just hovered on. We passed that variant's image into the `updateProduct` function from the event handler itself:

index.html

```
        <p @mouseover="updateProduct(variant.variantImage)">
```

In other words, the `updateProduct` method is ready to fire, with a parameter of `variantImage`.

When it's called, `variant.variantImage` is passed in as `variantImage` and is used to update the value of `this.image`. As we just learned, `this.image` **is** `image`. So the value of `image` is now dynamically updating based on the variant that was hovered on.

ES6 Syntax

Instead of writing out an anonymous function like `updateProduct: function(variantImage)`, we can use the ES6 shorthand and just say `updateProduct(variantImage)`. These are equivalent ways of saying the same thing.

What'd we Learn

- The `v-on` directive is used to allow elements to listen for events
- The shorthand for `v-on` is `@`
- You can specify the type of event to listen for:
  - click
  - mouseover
  - any other DOM event

- The `v-on` directive can trigger a method
- Triggered methods can take in arguments
- `this` refers to the current Vue instance's data as well as other methods declared inside the instance

Learn by doing

Challenge:

Create a new button and method to decrement the value of `cart`.

Computed Properties

In this lesson, we'll be covering Computed Properties. These are properties on the Vue instance that calculate a value rather than store a value.

Goal

Our first goal in this lesson is to display our `brand` and our `product` as one string.

Starting Code

index.html

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

    <div class="product-info">
        <h1>{{ product }}</h1>
        <p v-if="inStock">In Stock</p>
        <p v-else-if="inStock">Out of Stock</p>

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

        <div class="size-box">
            v-for="variant in variants"
            :key="variant.variantId"
            :style="{ backgroundColor: variant.variantColor }"
            @mouseover="updateProduct(variant.variantImage)"
        >
        </div>

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

        <div class="cart">
            <p>Cart({{ cart }})</p>
        </div>
    </div>
</div>
```

main.js

```
var app = new Vue({
    el: '#app',
    data: {
        product: 'Socks',
        brand: 'Vue Mastery',
        image: './assets/vmSocks-green.jpg',
        inStock: true,
        details: [ "80% cotton", "20% polyester", "Gender-neutral"],
        variants: [
            {
                variantId: 2234,
                variantColor: 'green',
                variantImage: './assets/vmSocks-green.jpg'

            },
            {
                variantId: 2235,
                variantColor: 'blue',
                variantImage: './assets/vmSocks-blue.jpg'
            }
        ],
        cart: 0
    },
    methods: {
        addCart() {
            this.cart += 1
        }
        updateProduct(variantImage) {
            this.image = variantImage
        }
    }
})
```

New brand data

Notice we've added a `brand`.

Problem

We want `brand` and `product` to be combined into one string. In other words, we want to display "Vue Mastery Socks" in our `h1` instead of just "Socks. How can we concatenate two values from our data?

Solution

Since **computed properties** *calculate* a value rather than store a value, let's add the `computed` option to our instance and create a computed property called `title`.

main.js

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

This is pretty straightforward. When `title` is called, it will concatenate `brand` and `product` into a new string and return that string.

Now all we need to do is put `title` within the `h1` of our page.

So instead of:

index.html

```
<h1>{{ product }}</h1>
```

We now have:

index.html

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

It works! "Vue Mastery Socks" is appearing in our `h1`.

Vue Mastery Socks

We've taken two values from our data and computed them in such a way that we've created a new value. If `brand` were to update in the future, let's say to "Vue Crafters", our computed property would not need to be refactored. It would still return the correct string: "Vue Crafters Socks". Our computed property `title` would still be using `brand`, just like before, but now `brand` would have a new value.

That was a pretty simple but not entirely practical example, so let's work through a more complex usage of a computed property.

A More Complex Example

Currently, the way we are updating our image is with the `updateProduct` method. We are passing our `variantImage` into it, then setting the image to be whichever variant is currently hovered on.

main.js

```
updateProduct(variantImage) {
    this.image = variantImage
}
```

This works fine for now, but if we want to change more than just the image based on which variant is hovered on, then we'll need to refactor this code. So let's do that now.

Instead of having `image` in our data, let's replace it with `selectedVariant`, which we'll initialize as 0.

main.js

```
selectedVariant: 0,
```

Why 0? Because we'll be setting this based on the index that we hover on. We can add index to our v-for here, like so.

index.html

```
<div class="size-box"
    v-for="(variant, index) in variants"
    :key="variant.variantId"
    :style="{ backgroundColor: variant.variantColor }"
    @mouseover="updateProduct(variant.variantImage)"
>
</div>
```

Now instead of passing in the variantImage, we'll pass in the index.

index.html

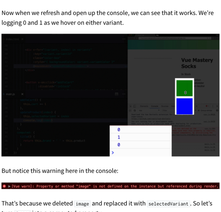
```
@mouseover="updateProduct(index)"
```

In our `updateProduct` method, we'll pass in the index, and instead of updating this.image, we'll update this.selectedVariant with the index of whichever variant is currently hovered on. Let's put a console.log in here too, to make sure it's working.

main.js

```
updateProduct(variantIndex) {
    this.selectedVariant = index
    console.log(index)
}
```

Now when we refresh and open up the console, we can see that it works. We're logging 0 and 1 as we hover on either variant.



But notice this warning here in the console:

```
▶ [Vue warn]: Property or method "image" is not defined on the instance but referenced during render.
```

That's because we deleted `image` and replaced it with `selectedVariant`. So let's turn `image` into a computed property.

main.js

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

Inside, we are returning `this.variants`, which is our array of variants, and we are using our `selectedVariant`, which is either 0 or 1, to target the first or second element in that array, then we're using dot notation to target its image.

When we refresh, our image is toggling correctly like it was before, but now we're using a computed property to handle this instead.

Now that we have refactored the `updateProduct` method to update the `selectedVariant`, we can access other data from the variant, such as the `variantQuantity` they both now have.

main.js

```
variants: [
    {
        variantId: 2234,
        variantColor: 'green',
        variantImage: './assets/vmSocks-green.jpg',
        variantQuantity: 10
    },
    {
        variantId: 2235,
        variantColor: 'blue',
        variantImage: './assets/vmSocks-blue.jpg',
        variantQuantity: 0
    },
],
```

Just like we did with `image`, let's remove `inStock` from our data and turn it into a computed property that uses our variant's quantities.

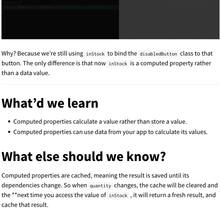
main.js

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

This is very similar to our `image` computed property, we're just targeting the `variantQuantity` now rather than the `variantImage`.

Now when we hover on the blue variant, which has a quantity of zero, inStock will evaluate to false since 0 is "falsey", so we'll now see Out of Stock appear.

Notice how our button is still conditionally turning gray whenever `inStock` is false, just like before.



Why? Because we're still using `inStock` to bind the `disabledButton` class to that button. The only difference is that now `inStock` is a computed property rather than a data value.

What'd we learn

- Computed properties calculate a value rather than store a value.
- Computed properties can use data from your app to calculate its values.

What else should we know?

Computed properties are cached, meaning the result is saved until its dependencies change. So when `quantity` changes, the cache will be cleared and the `*` method you access the value of `inStock`, it will return a fresh result, and cache that result.

With that in mind, it's more efficient to use a computed property rather than a method for an expensive operation that you don't want to re-run every time you access it.

It is also important to remember that you should not be mutating your data model from within a computed property. You are merely computing values based on other values. Keep these functions pure.

Learn by doing

Challenge:

Add a boolean data property `onSale` and create a computed property that takes `brand`, `product` and `onSale` and prints out a string whenever `onSale` is true.

Components

Welcome

In this lesson we'll be learning about the wonderful world of components. Components are reusable blocks of code that can have both structure and functionality. They help create a more modular and maintainable codebase.

Goal

Throughout the course of this lesson we'll create our first component and then learn how to share data with it.

Starting Code

```
app.component('app',
  template: `<div class="product">
    <img src="image" />
    <div>
      <h1>Name</h1>
      <ul>
        <li>Size: product.size</li>
        <li>Color: product.color</li>
        <li>Shipping: product.shipping</li>
        <li>Sale: product.onSale</li>
        <li>Inventory: {{ product.inventory }}</li>
      </ul>
      <div>
        <button v-on:click="updateCart"
          disabled="disabled"
          v-bind:disabled="inventory <= 0">Add to Cart</button>
        <button v-on:click="updateCart">Remove from Cart</button>
        <div>Cart: {{ cart.length }}</div>
      </div>
    </div>
  </div>`,
  data() {
    return {
      product: 'Socks',
      image: 'vmsocks.jpg',
      size: 'In Stock',
      color: 'Green',
      shipping: '2.99',
      onSale: true,
      inventory: 10,
      cart: 0
    }
  },
  methods: {
    updateCart() {
      this.cart += 1
    },
    removeFromCart() {
      if (this.cart > 0) {
        this.cart -= 1
      }
    }
  }
)
```

```
app.mount('#app')
```

Problem

In a Vue application, we don't want all of our data, methods, computed properties, etc. living on the root instance. Over time, that would become unmanageable. Instead, we'll want to break up our code into modular pieces so that it is easier and more flexible to work with.

Solution

We'll start out by taking the bulk of our current code and moving it over into a new component for our product.

We register the component like this:

```
app.component('product', {
  template: `<div class="product">
    
```

Communicating Events

In our previous lesson, we learned how to create components and pass data down into them via props. But what about when we need to pass information back up? In this lesson we'll learn how to communicate from a child component up to its parent.

Goal

By the end of this lesson, our `product` component will be able to tell its parent, the root instance, that an event has occurred, and send data along with that event notification.

Starting Code

Currently, our app looks like this:

```
    <div id="app">
      <product :premium="premium"></product>
    </div>
```

```
Vue.component('product', {
  props: {
    premium: {
      type: Boolean,
      required: true
    },
  },
  template:
  <div id="product">

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

    <div class="product-info">
      <div class="cart">
        <p>Cart({{ cart }})</p>
      </div>

      <h1>{{ title }}</h1>
      <p>Shipping: {{ shipping }}</p>

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

      <h2>Details</h2>
      <ul>
        <li v-for="detail in details">{{ detail }}</li>
      </ul>
      <h2>Colors:</h2>
      <div v-for="variant in variants" :key="variant.variantId">
        <div class="color-box" style="{ backgroundColor:
variant.variantColor }" @mouseover="updateProduct(index)"></div>
      </div>
      <button :class="{ disabledButton: !inStock }" v-on:click="addToCart"
:disabled="!inStock">Add to Cart</button>
      </div>
    </div>

  data() {
    return {
      product: "Socks",
      brand: "Vue Mastery",
      selectedVariant: 0,
      details: ["80% cotton", "20% polyester", "Gender-neutral"],
      variants: [
        {
          variantId: 2234,
          variantQuantity: 15,
          variantColor: "green",
          variantImage: "./assets/vmSocks-green.jpg"
        },
        {
          variantId: 2235,
          variantQuantity: 0,
          variantColor: "blue",
          variantImage: "./assets/vmSocks-blue.jpg"
        }
      ],
      cart: 0
    }
  },
  methods: {
    addToCart() {
      this.cart += 1
    },
    updateProduct(index) {
      this.selectedVariant = index
    },
  },
  computed: {
    title() {
      return this.brand + " " + this.product
    },
    image() {
      return this.variants[this.selectedVariant].variantImage
    },
    inStock() {
      if (this.quantity > 0) {
        return true
      } else {
        return false
      }
    },
    shipping() {
      if (this.premium) {
        return "Free"
      } else {
        return 2.99
      }
    }
  }
})

var app = new Vue({
  el: '#app',
  data: {
    premium: true
  }
})
```

Problem

Now that `product` is its own component, it doesn't make sense for our cart to live within `product`. It would get very messy if every single product had its own cart that we need to keep track of. Instead, we'll want the cart to live on the root instance, and have `product` communicate up to that cart when its "Add to Cart" button is pressed.

Solution

Let's move the cart back to our root instance.

```
var app = new Vue({
  el: '#app',
  data: {
    premium: true,
    cart: 0
  }
})
```

And we'll move our cart's template into our index.html:

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

      <product :premium="premium"></product>
    </div>
```

As expected, now if we click on the "Add to Cart" button, nothing happens.

What do we want to happen? When the "Add to Cart" button is pressed in `product`, the root instance should be notified, which then triggers a method it has to update the `cart`.

First, let's change out the code we have in our component's `addToCart` method.

It was this:

```
      addToCart() {
        this.cart += 1
      },
```

Now it's this:

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

What does this mean?

It means: when `addToCart` is run, `emit` an event by the name of "add-to-cart". In other words, when the "Add to Cart" button is clicked, this method fires, announcing that the click event just happened.

But right now, we have nowhere *listening* for that announcement that was just emitted. So let's add that listener here:

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

Here we are using `@add-to-cart` in a similar way as we are using `:premium`. Whereas `:premium` is a funnel on `product` that data can be passed down into, `@add-to-cart` is essentially a radio that can receive the event emission from when the "Add to Cart" button was clicked. Since this radio is on `product`, which is nested within our root instance, the radio can blast the announcement that a click happened, which will trigger the `updateCart` method, which lives on the root instance.

```
    @add-to-cart="updateCart"
```

This code essentially translates to: "When you hear that the "Add to Cart" event happened, run the `updateCart` method.

That method should look familiar:

```
      methods: {
        updateCart() {
          this.cart += 1
        }
      }
```

It's the method that used to be on `product`. Now it lives on our root instance, and is called whenever the "Add to Cart" button is pressed.

Now when our button is pressed, it triggers `addToCart`, which emits an announcement. Our root instance hears the announcement through the radio on its `product` component, and the `updateCart` method runs, which increments the value of `cart`.

So far so good.

But in a real application, it's not helpful to only know that a product was added to the cart, we'd need to know *which* product was just added to the cart. So we'll need to pass data up along with the event announcement.

We can add that in as a second argument when we emit an event:

```
      addToCart() {
        this.$emit('add-to-cart',
this.variantId[this.selectedVariant].variantId)
      },
```

Now, along with the announcement that the click event occurred, the id of the product that was just added to the cart is sent as well. Instead of just incrementing the number of cart, we can now make cart an array:

```
    cart: []
```

And push the product's id into our cart array:

```
      methods: {
        updateCart(id) {
          this.cart.push(id)
        }
      }
```

Now our array has one product within it, whose `id` is being displayed on the page.

Cart([2234])

But we don't need to display the contents of our array here. Instead, we just want to display the amount of products in our array, so we can say this in our template instead:

```
    <p>Cart({{ cart.length }})</p>
```

Now we're just displaying the length of the array, or in other words: the number of products in the cart. It looks just like it did before, but instead of only incrementing the value of `cart` by 1, now we're actually sending data about which product was just added to the cart.

Great work!

What'd we learn

- A component can let its parent know that an event has happened with `$emit`
- A component can use an event handler with the `v-on` directive (`@` for short) to listen for an event emission, which can trigger a method on the parent
- A component can `$emit` data along with the announcement that an event has occurred
- A parent can use data emitted from its child

Learn by doing

Challenge:

Add a button that removes the product from the cart array by emitting an event with the id of the product to be removed.