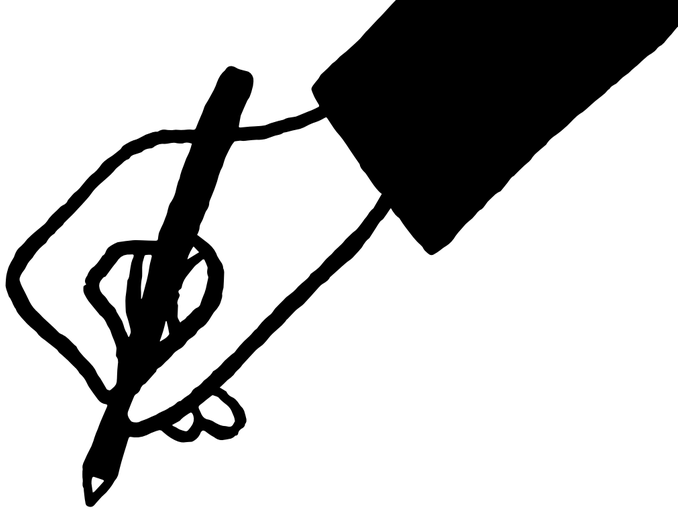
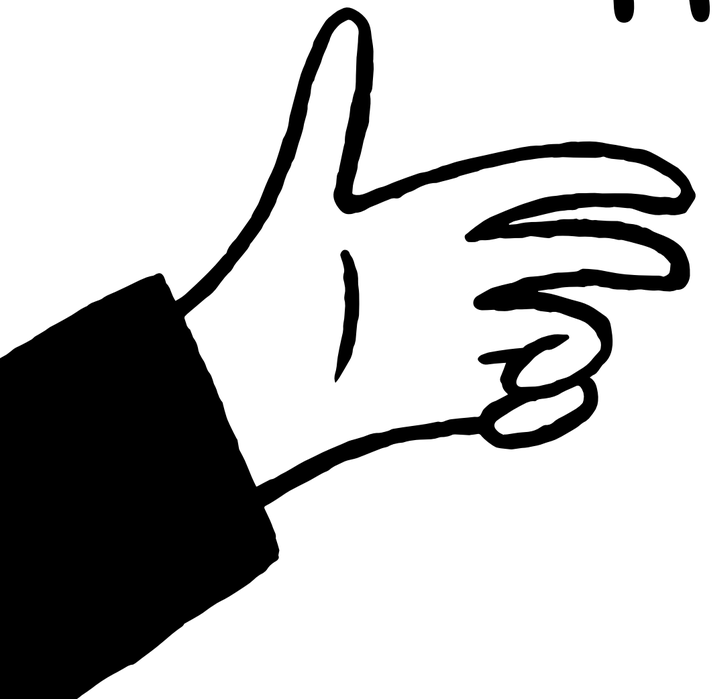
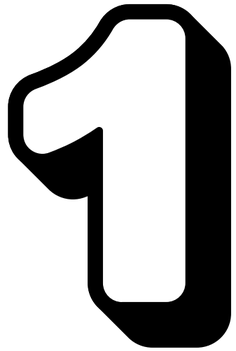


ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ
И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Frontend Development Projects with Vue.js 3

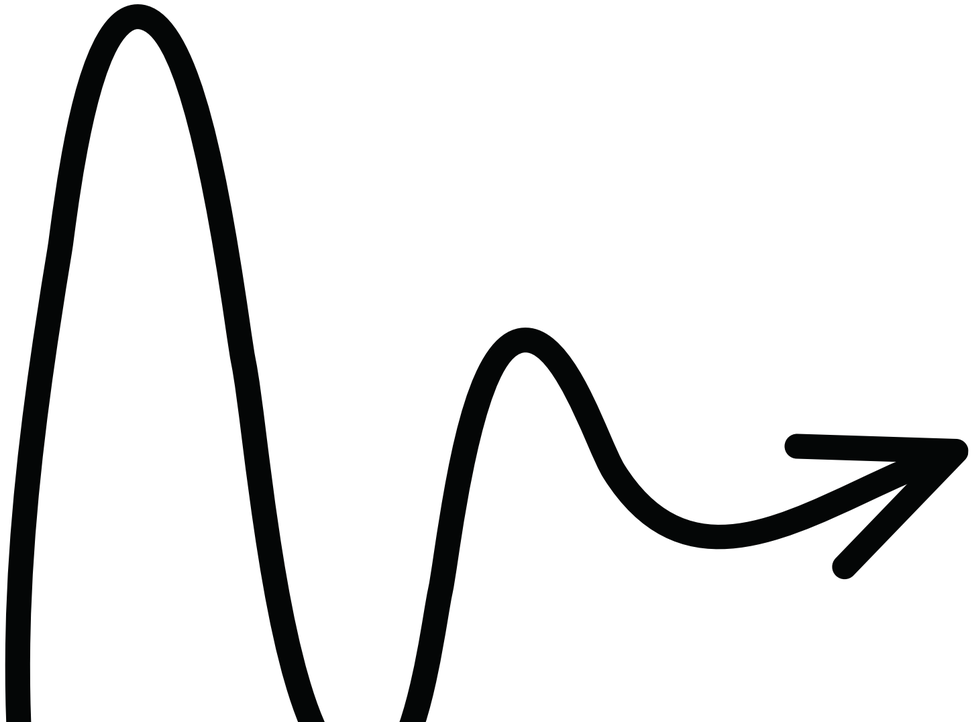


Advanced Web Design

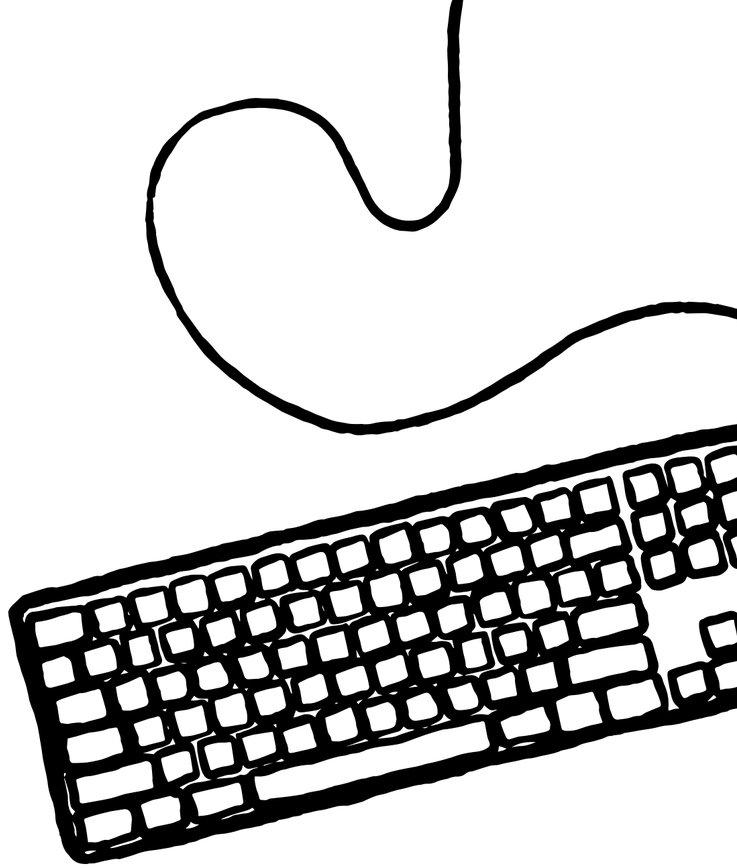


Starting Your First Vue Project

- Understanding Vue as a framework
- Setting up a Vite-powered Vue application
- Exploring data properties as a local state
- Writing components with <script setup>
- Understanding Vue directives
- Enabling two-way binding using v-model
- Understanding data iteration with v-for
- Exploring methods
- Understanding component lifecycle hooks
- Styling components
- Understanding CSS modules

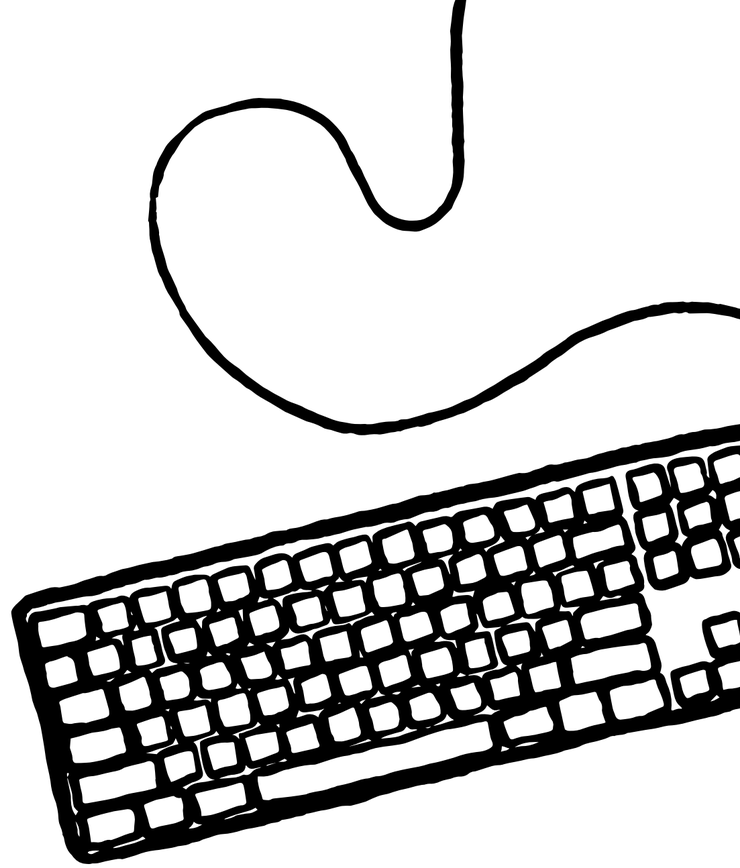


- Angular is MVVM-based, TypeScript-friendly, and includes AoT rendering, routing, and CLI tools.
- Angular lacks built-in global state management; developers must use Flux or NgRx.
- Vue simplifies Angular’s rigid structure and patterns, supports TypeScript from version 3.0, and avoids enforced coding styles.
- Vue is easier to set up, more efficient, and more intuitive for developers



Angular vs. Vue

- React uses JSX pattern, which blends HTML with JavaScript, increasing the learning curve.
- Both frameworks support modular, component-based architecture.
- Vue offers flexibility: developers can choose JSX or traditional separation of HTML, CSS, and JS.
- Vue's Single-File Components (SFCs) enhance modularity while maintaining readability.



React vs. Vue

Why Vue?

Advantages of Using Vue

- Gentle learning curve makes onboarding easier and reduces team overhead.
- SFCs allow gradual integration into existing sites without full rewrites
- Vuex and Pinia offer robust global state management
- Vue's ecosystem (Vue Router, Vuex, Pinia) and community support make it a cost-effective and scalable choice

1

```
<!DOCTYPE html>
<html>
<head>
<title>Vue.js project with CDN</title>
<script src="https://unpkg.com/vue@3"></script>
</head>
</html>
```

Getting Started with Vue Architecture

- Begin by importing the Vue package into your coding environment.
- The simplest method is using the official Vue CDN.

Getting Started with Vue Architecture

2

ViewModel - abstraction of the view that describes the state of the data in the model.
Binding a Vue instance to vm helps you to keep track of your Vue instance in a block of code.

```
const vm = Vue.createApp({
// options
})
```

- Each Vue application consists of only one root Vue instance
- The Vue class constructor accepts an options object for the configurations and behavior of components.
- We call this approach Options API and we can use it for all corresponding Vue components.

Understanding the Vue instance

3

```
<body>
<div id="vue-app"></div>
<script>
const vm = Vue.createApp({
//Options
})
</script>
</body>
```

- Vue engine renders the application instance, in index.html file, using element as the main entry point for the application

4

```
<body>
 <div id="vue-app"></div>
  <script>
   const vm = Vue.createApp({
    //Options
    })
   vm.mount('#vue-app')
  </script>
</body>
```

- Render the Vue application in the browser → trigger vm.mount() to mount the root component to the targeted HTML element using a unique selector

5

data() is a function that returns an Object instance containing the local state (or local variables) of a component.

```
const vm = Vue.createApp({
  data() {
    return {
      text: 'Start using Vue.js today!'
    }
  }
})
```

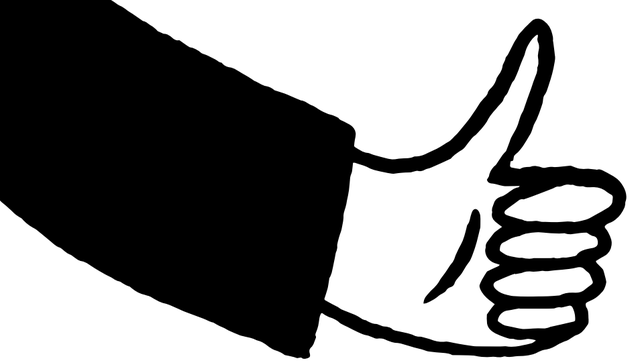
- But how to render the “text” variable that contains the reactive content?

Vue template syntax, represented by double curly braces

```
<div id="vue-app">{{ text }}</div>
```

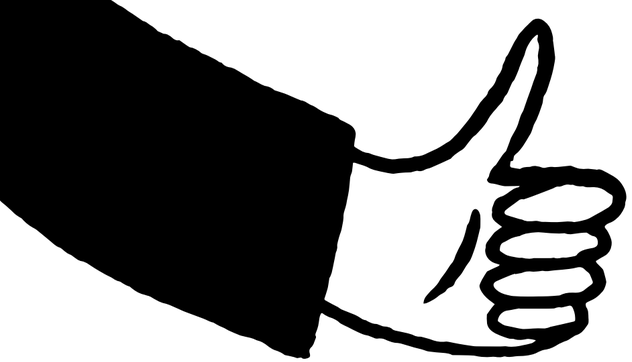
Displaying “Start using Vue.js today!” using a local data property

Start using Vue.js today!



Setting up a Vite-powered Vue application

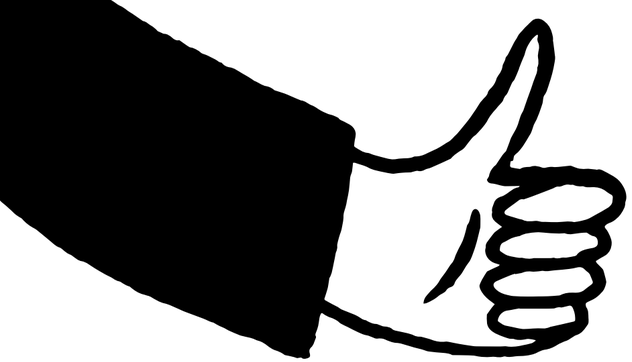
https://vuejs.org/guide/quick-start



Vite is a build tool that aims to provide a faster and leaner development experience for modern web projects.

It consists of two major parts:

- A dev server that provides rich feature enhancements over native ES modules, for example extremely fast Hot Module Replacement (HMR).
- A build command that bundles your code with Rollup, pre-configured to output highly optimized static assets for production.



Vue's SFC architecture

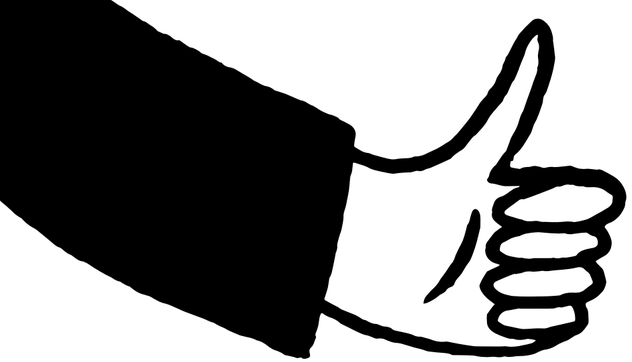
Vue **Single-File Components** (a.k.a. ***.vue** files, abbreviated as SFC) is a special file format that allows us to encapsulate the template, logic, and styling of a Vue component in a single file.

JavaScript

HTML

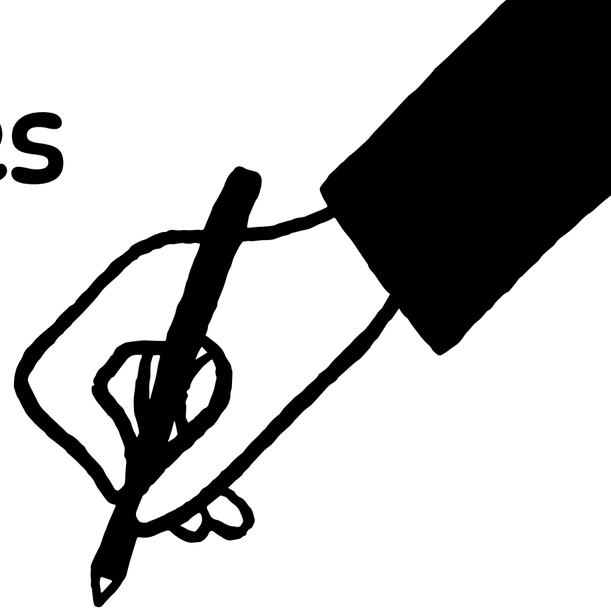
CSS

```vue
<script setup>
import { ref } from 'vue'
const greeting = ref('Hello World!')
</script>

<template>
  <p class="greeting">{{ greeting }}</p>
</template>

<style>
.greeting {
  color: red;
  font-weight: bold;
}
</style>
```



Exercise 1.01 – building your first component

Exploring data properties as a local state

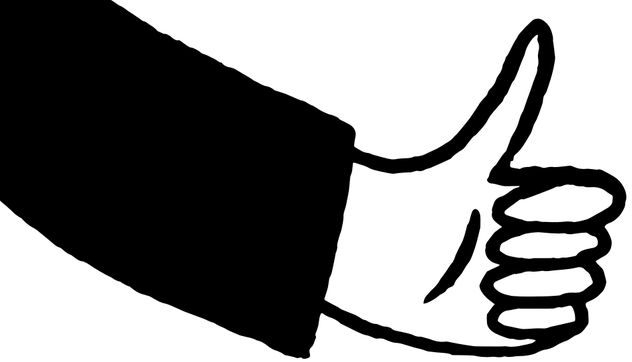


```
<script>
 export default {
  data() {
   return {
    color: 'red'
   }
  }
 }
</script>
```

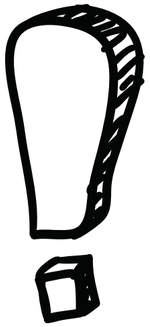
One of the most used terms and reactive elements used when constructing Vue components is data properties. These manifest themselves within the data() function of a Vue instance.

```
<template>
 <span> {{ color }}</span>
</template>

<script>
 export default {
  data() {
   return {
    color: 'red'
   }
  }
 }
</script>
```

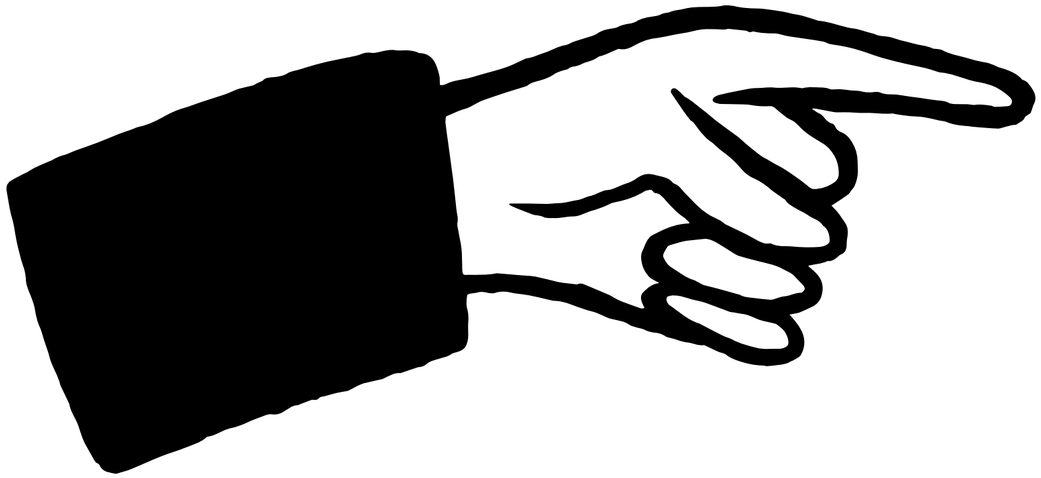
Once we have defined our local data, we need to bind it to the template section to display its values in the UI, which is called data interpolation.
Interpolation is the insertion of something of a different nature into something else. In the Vue context, this is where you would use mustache syntax (double curly braces) to define an area in which you can inject data into a component's HTML template.



Exercise 1.02 – interpolation with conditionals



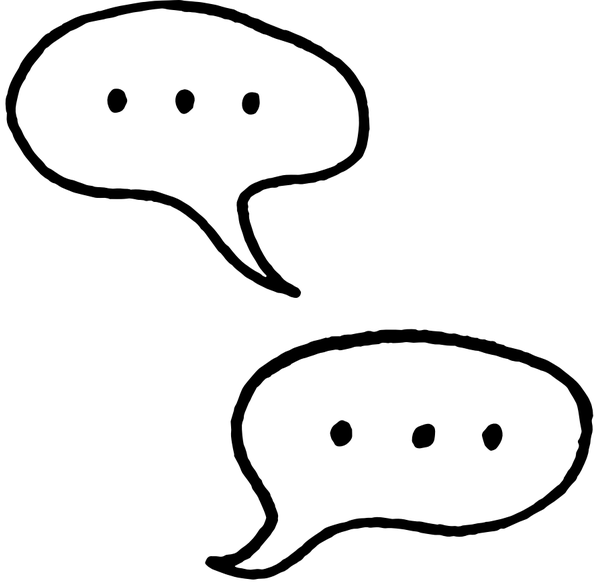
Understanding Vue directives



All Vue-based directives start with a v-* prefix as a Vue-specific attribute

```
<template>
  <div v-text="msg">My placeholder</div>
</template>

<script setup>
  const msg = "My message"
</script>
```

v-text



- The **v-text** directive has the same reactivity as with **interpolation**. Interpolation with **{{ }}** is more performant than the v-text directive.
- However, you may find yourself in situations where you have **pre-rendered** text from a server and want to **override** it once your Vue application has finished loading.
- For example, you can pre-define a **static placeholder text** while waiting for the Vue engine to eventually replace it with the **dynamic value received from v-text,** as shown in the following code block:

```
<!-- single element -->
<span v-once>This will never change: {{msg}}</span>

<!-- the element has children -->
<div v-once>
  <h1>Comment</h1>
  <p>{{msg}}</p>
</div>

<!-- component -->
<MyComponent v-once :comment="msg"></MyComponent>

<!-- `v-for` directive -->
<ul>
  <li v-for="i in list" v-once>{{i}}</li>
</ul>
```

v-once

- When used, it indicates the **starting point of static content**.
- The Vue engine will render the component with this attribute and its children **exactly once**.
- It also **ignores** all **data updates** for this component or element **after the initial render**. This attribute is handy for scenarios with **no reactivity needed** for certain parts.
- You can combine **v-once with v-text, interpolation**, and any Vue directive.

V-html

```
<div v-html="html"></div>
```

- Vue will parse the value passed to this directive and render your text data as a valid HTML code into the target element.
- We don’t recommend using this directive, especially on the client side, due to its **performance impact** and the **potential security leak**.
- The script tag can be embedded and triggered using this directive.

v-bind

```vue
<template>
  <img v-bind:src="logo" />
</template>

<script setup>
 const logo = '../assets/logo.png';
</script>
```

```vue
<template>
  <img :src="logo" />
</template>
```

- Most popular
- You can use this directive to enable **one-way binding** for a data variable or an expression to an HTML attribute
- You can also use it to pass a local data variable as **props** to another component. A shorter way is using the **:attr** syntax instead of v-bind:attr.

v-if

```
<div v-if="type === 'A'">
  A
</div>
<div v-else-if="type === 'B'">
  B
</div>
<div v-else-if="type === 'C'">
  C
</div>
<div v-else>
  Not A/B/C
</div>
```

- This directive operates like the **if...else and if...else if...** conditions.
- It comes with supporting directives, such as v-else, standing for the else case, and v-else-if, standing for the else if case.

v-show

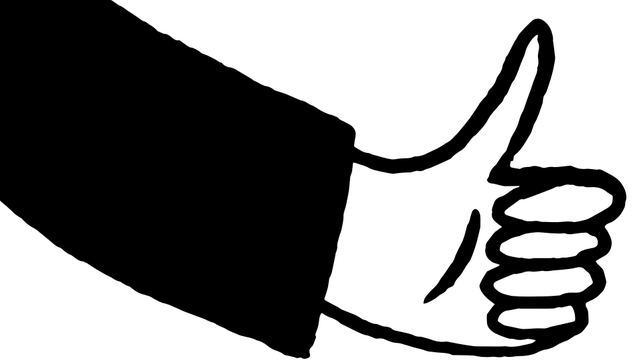
```
<h1 v-show="ok">Hello!</h1>
```

- You can also control the visible state of HTML elements by using v-show.
- Unlike v-if, with v-show, the Vue engine still mounts the element to the DOM tree but hides it using the display: none CSS style.
- You can still see the content of the hidden element visible in the DOM tree upon inspecting it, but it is not visible on the UI to end users.
- This directive does not work with v-else or v-else-if. If v-show results in a true Boolean, it will leave the DOM element as is.
- If it resolves as false, it will apply the display: none style to the element.

v-for

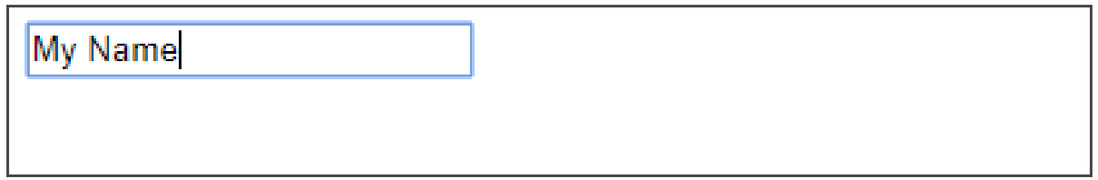
```
<div v-for="item in items">
  {{ item.text }}
</div>
```

```
<div v-for="item in items" :key="item.id">
  {{ item.text }}
</div>
```



Exercise 1.03 – exploring basic directives (v-text, v-once, v-html, v-bind, v-if, v-show)

Enabling two-way binding using v-model

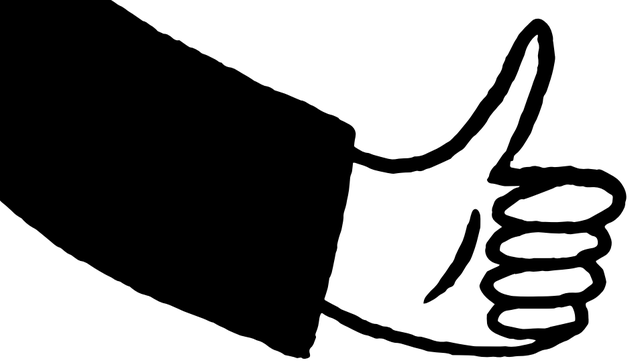



```vue
<template>
  <input v-model="name" />
</template>

<script>
  export default {
   data() {
    return {
     name: ''
    }
   }
  }
</script>
```

Vue achieves **two-way data binding** by creating a dedicated directive that **watches** a data property within your Vue component.

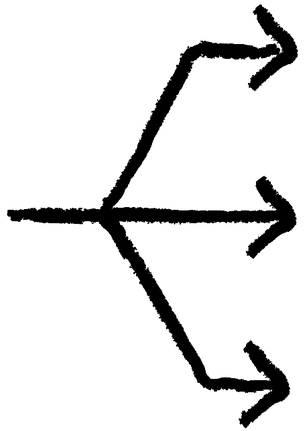
The **v-model** directive triggers data updates when the target data property is modified on the UI. This directive is usually useful for HTML form elements that need to **both display the data and modify it reactively**—for example, *input*, *textarea*, *radio buttons*, and so on.

We can enable two-way binding by adding the v-model directive to the target element and **binding** it to our desired data **props**



Exercise 1.04 – experimenting with two-way binding using v-model

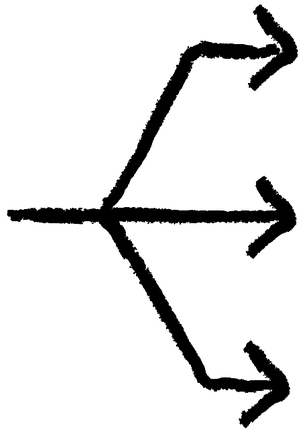
Understanding data iteration with v-for



```
v-for="(item, index) in items" :key="index"
```

We are iterating through a list of items.
We have access to a single item and its index in the list in each iteration.
:key is a required attribute, acting as the unique identifier of each iterating element rendered for the Vue engine to keep track.

Understanding data iteration with v-for



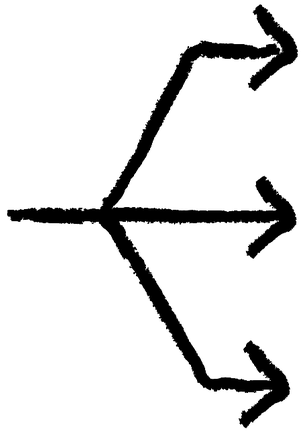
```
<template>
<div v-for="n in 2" :key="'loop-1-' + n">
{{ n }}
</div>
<template>
```

If you have multiple loops in one component, you should **randomize** the key attribute with extra characters or context-related strings to avoid **key duplication conflicts.**

There are various use cases for this direction. One straightforward use case is to perform **anonymous loops**, in which you can define a number, X, as a symbolic list, and the loop will iterate that X times.

This can be handy in situations in which you strictly control the number of iterations you want or render some placeholder content.

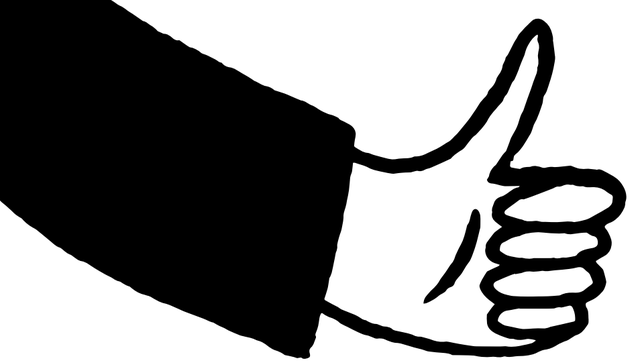
Understanding data iteration with v-for



```
<template>
  <div v-for="n in 5" :key="`loop-2-${n}`">
   {{ n }}
  </div>
<template>
```

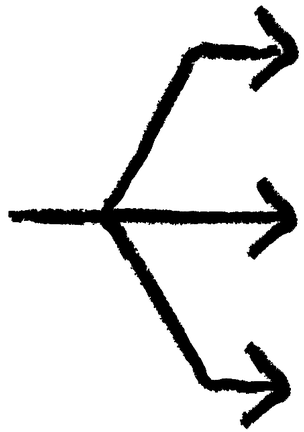
In the following example, we see an anonymous loop in which the total iterations are 2 and we define key with a loop-1 prefix

1
2
3
4
5



Exercise 1.05 – using v-for to iterate through an array of strings

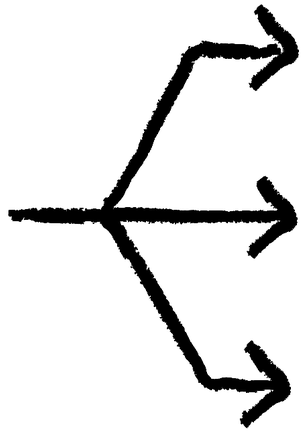
Iterating through an array of objects



```
v-for="(item, index) in items" :key="index"
```

Vue makes it easy to control various data states through its directive syntax. Like iterating through an array of strings, the directive syntax remains the same

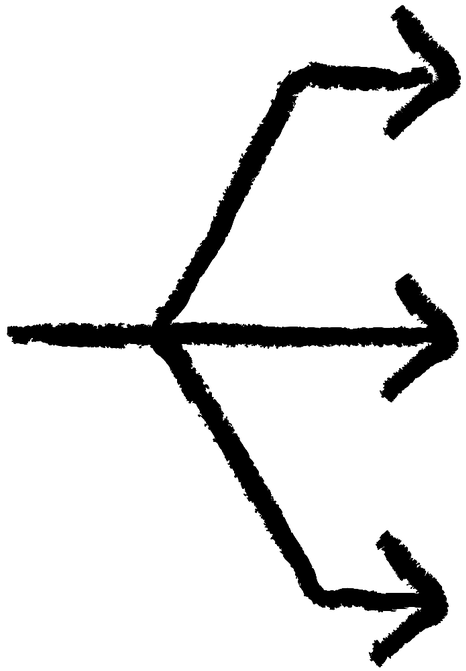
Iterating through an array of objects



```
<template>
  <ul>
   <li v-for="(item, index) in items" :key="item.id">
    <h2>{{ item.title }}</h2>
    <span>{{ item.description }}</span>
   </li>
  </ul>
</template>
```

→ Since characteristics is an array, we display its values by using a v-for directive again for characteristics.
→ You don't have to use the same name, item, that the syntax example shows.
Instead, you can give it a different name depending on how you want your variable to be.

Iterating through an array of objects



①

```
<template>
 <ul>
  <li v-for="item in items" :key="item.id">
   <h2>{{ item.title }}</h2>
   <span>{{ item.description }}</span>
   <ul>
    <li v-for="(str, index) in item.characteristics"
:key="index">
     <span>{{ str }}</span>
    </li>
   </ul>
  </li>
 </ul>
</template>
```

②

```
<script setup>
const items = [{
id: 1,
title: "Item 1",
description: "About item 1",
characteristics: ["Summer", "Winter", "Spring", "Autumn"]
}, {
id: 2,
title: 'Item 2",
description: 'About item 2",
characteristics: ["North", "West", "East", "South"]
}]
</script>
```

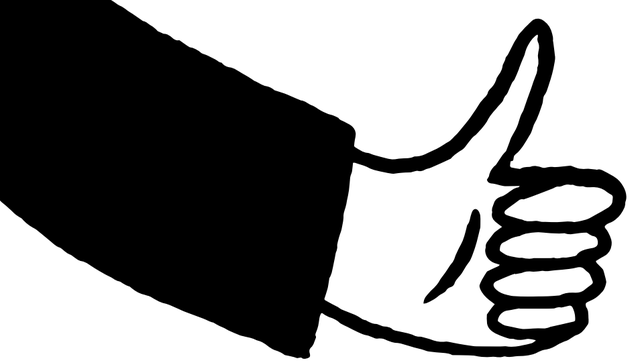
③

- **items 1**

  About item 1
  - Summer
  - Winter
  - Spring
  - Autumn

- **Item 2**

  About item 2
  - North
  - West
  - East
  - South



Exercise 1.06 – using v-for to iterate through an array of objects and using their properties in v-if conditions

Iterating through a keyed collection (Object)

```
<script setup>
const course = {
 title: 'Frontend development with Vue',
 description: 'Learn the awesome of Vue',
 lecturer: 'Maya and Raymond'
}
</script>
```

```
<template>
 <ul>
  <li v-for="(value, key, index) in course" :key="key">
{{index}}. {{key}}: {{value}}
  </li>
 </ul>
</template>
```

- 0. title: Frontend development with Vue
- 1. description: Learn the awesome of Vue
- 2. lecturer: Maya and Raymond

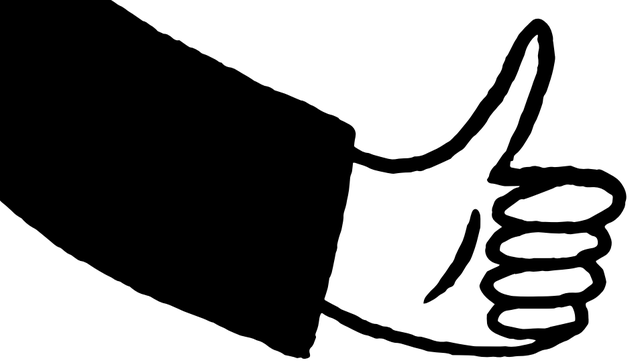
```
v-for="(value, key, index) in obj"
```

We can generally use v-for for looping through any **iterative data** collection type.

Object in JavaScript is a **key-value data collection**, and we can iterate through its properties using v-for.

Here, we change the naming convention from (item, index) to (**value, key**), in which key is the object's property, and value is that key property's value.

Vue also exposes one more parameter—**index**—to indicate that property's appearance index in the target object



Exercise 1.07 – using v-for to loop through the properties of Object

Exploring methods

Vue 2.0

```
<script>
  export default {
   methods: {
    myMethod() { console.log('my first method'); }
   }
  }
</script>
```

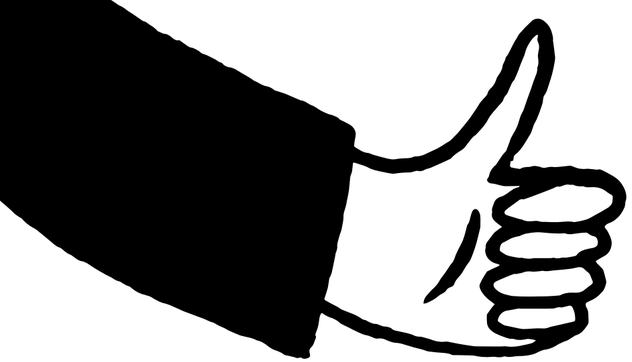
Vue 3.0

```
<script setup>
  const myMethod = () => { console.log('my first method'); }
</script>
```

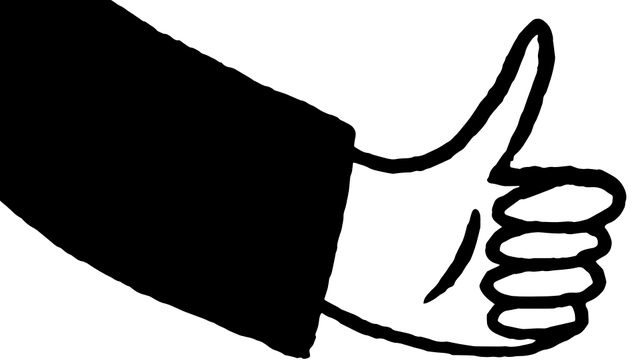
```
<template>
  <button id="click-me" v-on:click="myMethod">Click me </button>
  <button id="click-me" @click="myMethod">Click me shorter</button>
</template>
```

- Vue defines component ***methods*** inside the methods object as part of a Vue instance.
- You compose each component method as a **normal JavaScript function**.
- The Vue method is **scoped to your Vue componen**t and can be run from anywhere inside the component it belongs to.
- It also has access to the ***this*** instance, which indicates the instance of the component.

- You then can bind the methods to HTML events of an element as its **event listeners** in the template section.
- When binding events to HTML elements in Vue, you would use the **@ symbol**. For example,
- **v-on:click** is equivalent to **@click**, as shown in the following code block



Exercise 1.08 – triggering methods



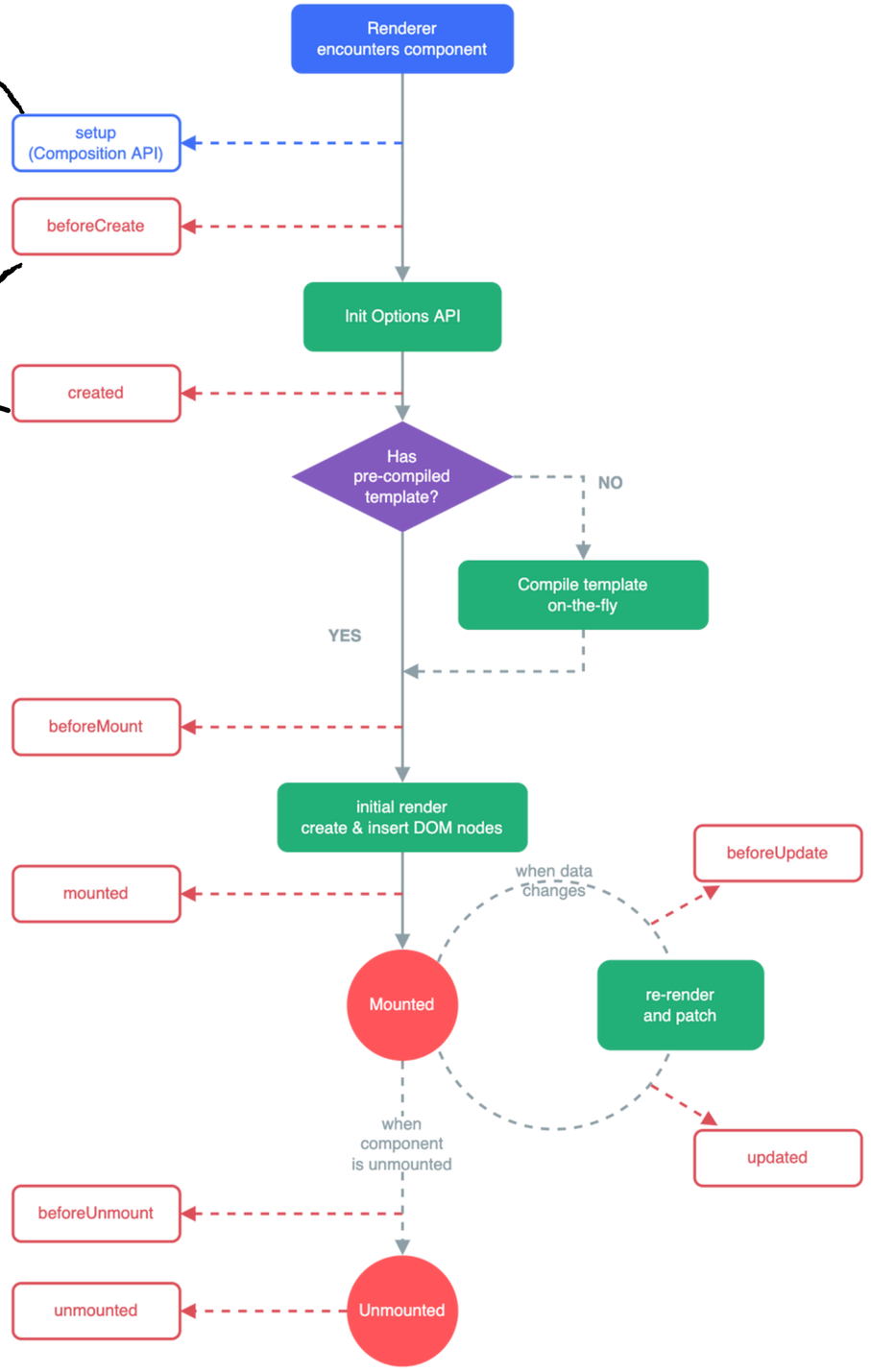
Exercise 1.09 – returning data using Vue methods

setup: This event runs before all other hooks. It doesn't have access to this instance since the instance has not yet been created at this point.

beforeCreate: This runs when your component has been initialized. *data* has not been made reactive and events are not set up in your DOM.

created: to access reactive *data* and *events*, but the templates and DOM are not mounted or rendered. This hook is generally good to use when requesting asynchronous data from a server since you will more than likely want this information as early as possible before the virtual DOM is mounted.

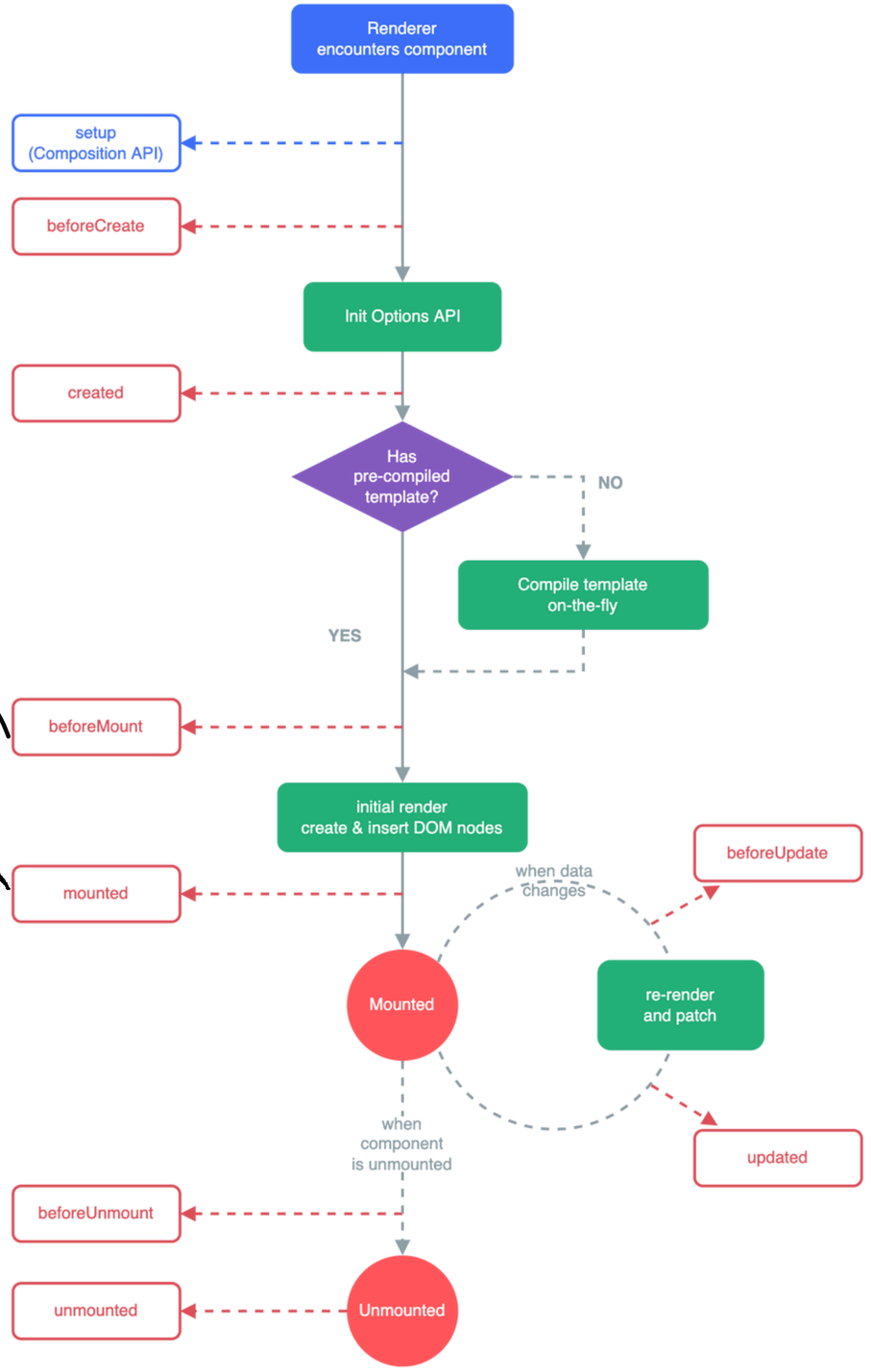
Understanding component lifecycle hooks



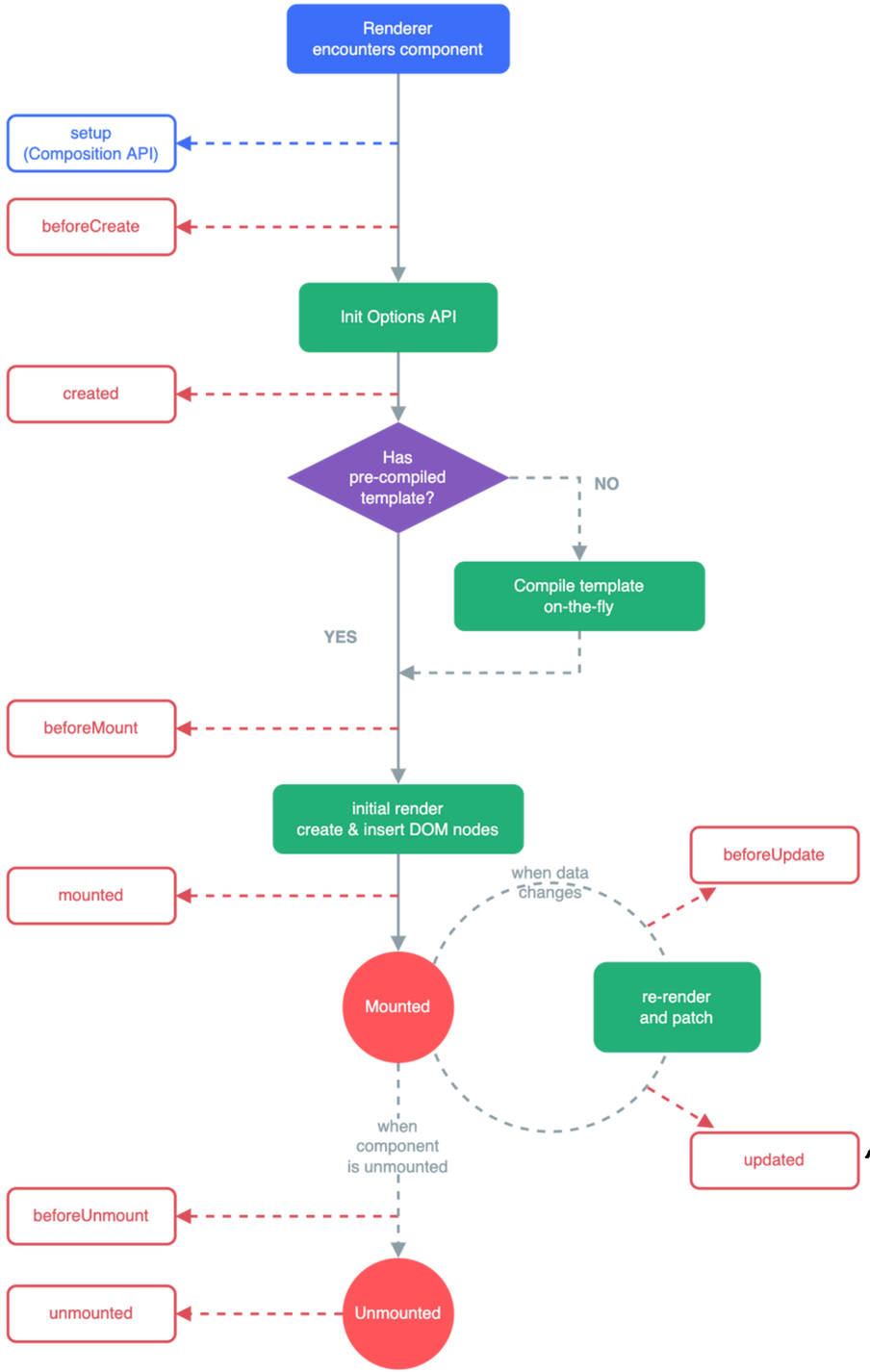
Understanding component lifecycle hooks

beforeMount: A very uncommon hook, as it runs directly before the first render of your component

mounted: among the most common hooks you will use - they allow you to access your DOM elements so that non-Vue libraries can be integrated.



Understanding component lifecycle hooks



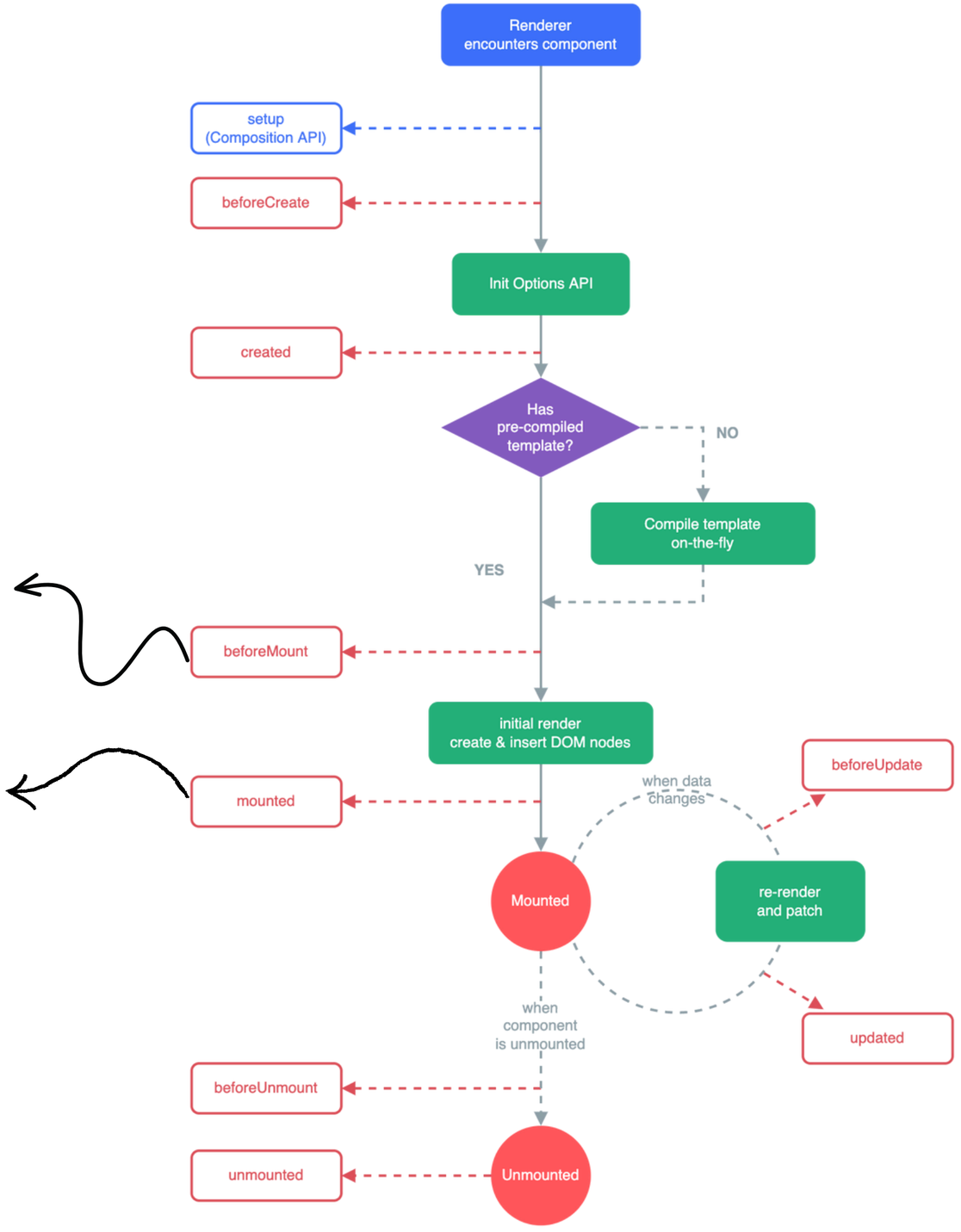
beforeUpdate: runs immediately after a *change* to your component occurs and before it has been re-rendered.

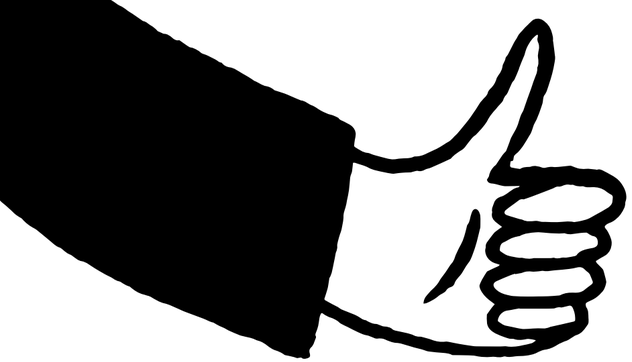
updated: runs immediately *after* the beforeUpdate hook and re-renders your component with new data changes.

Understanding component lifecycle hooks

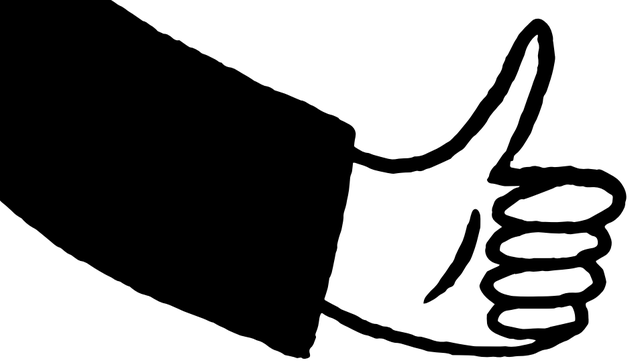
beforeUnMount: This is fired directly before unmounting your component instance. The component will still be functional until the unmounted hook is called, allowing you to *stop event listeners and subscriptions to data to avoid memory leaks*.

unmounted: All the virtual DOM elements and event listeners have been *cleaned up* from your Vue instance. This hook allows you to communicate that to anyone or any element that needs to know this has been done.

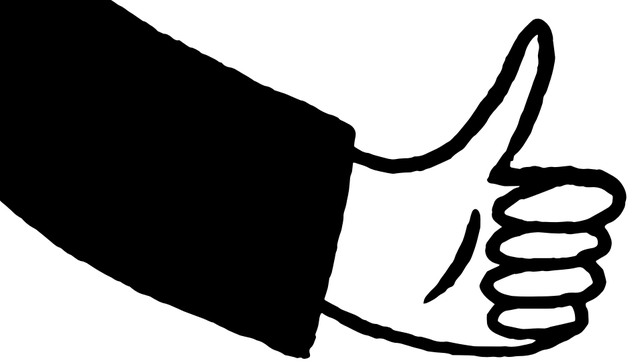




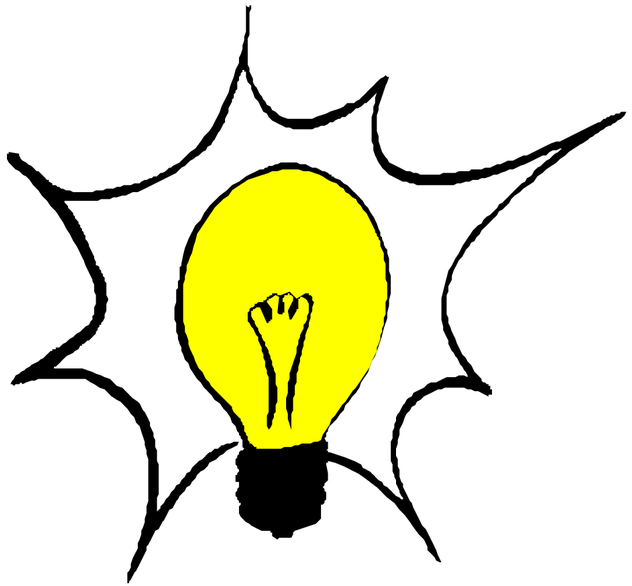
Exercise 1.10 – using a Vue lifecycle to control data



Exercise 1.11 – importing SCSS into a scoped component



Exercise 1.12 – styling Vue components using CSS modules



Questions?