

# Vue Basics

---

Prof. Emanuele Panizzi

<https://sfc.vuejs.org/>

```
<template>  
<h1>  
  Hello World!  
</h1>  
</template>
```

- The `<template>` in Vue.js contains what will be rendered in the html page

- We want to create a counter that the user can increment or decrement, step by step
- composed of a “-” button, the counter value, and a “+” button

## Looks like this

```
<template>
<button>
  -
</button>
0
<button>
  +
</button>
</template>
```



(`<button>` is an html element)

## The Counter

- the counter should increase or decrease when the user presses the “+” or “-” button, respectively
- we need a variable there

```
<template>
<button>
  -
</button>
{{ counter }}
<button>
  +
</button>
</template>
```

- double braces `{{}}` can contain any js expression

## Definition of the counter variable

```
<script>
export default {
  data() {      // function that returns an object
    return {    // containing all the
      counter: 0 // variable definitions
    }
  }
}
</script>
```

- variables can be used:
  - in the `<template>`, e.g. `{{ counter }}`
  - in the `<script>`, e.g. `this.counter`

- variables defined by *data()* are called 'the reactive state'
- if their values change, the template is re-rendered
- like in the *observer* pattern
  
- in our example, if counter is incremented, the template is re-rendered and the new value is shown



- On button click we want to increment/decrement the counter
- we listen to the DOM click event, using the `@click` attribute<sup>1</sup>

```
<template>
<button @click="counter--"> - </button>
{{ counter }}
<button @click="counter++"> + </button>
</template>
```

---

<sup>1</sup>these Vue attributes are called *directives*

- the value of a directive can be any js expression
- e.g. *counter++*
- e.g. a function call like *incr()*

```
<template>  
<button @click="decr()"> - </button>  
{{ counter }}  
<button @click="incr()"> + </button>  
</template>
```

## Methods

- a function call refers to a method to be defined in the script

```
...  
  counter: 0,  
}  
,  
methods: {  
  incr() {  
    this.counter++  
  },  
  decr() {  
    this.counter--  
  }  
}  
...
```



## Negative numbers in red

This is static style (always red):

```
<template>  
<button @click="decr()"> - </button>  
<span style="color:red">{{ counter }}</style>  
<button @click="incr()"> + </button>  
</template>
```

## Negative numbers in red

This is dynamic style (style changes if variable *dCol* changes):

```
<template>
<button @click="decr()"> - </button>
<span :style="dCol">{{ counter }}</style>
<button @click="incr()"> + </button>
</template>
```

- note the `:` notation to express the binding
- *dCol* variable must be defined in the reactive state
- when *dCol* changes, the `<template>` is re-rendered

## Definition and update of dCol

```
data() {  
  return {  
    counter: 0,  
    dCol: 'color:black';  
  }  
},
```

-4+

```
methods: {  
  incr() {  
    this.counter++  
    if (this.counter >= 0) {  
      this.dCol = 'color:black;';  
    }  
  },  
  decr() {  
    this.counter--  
    if (this.counter < 0) {  
      this.dCol = 'color:red;';  
    }  
  }  
}
```

```
// ... in incr():
  if (this.counter >= 0) {
    this stepperStyle =
      'color:black;font-size:'+this.counter+'em;';

// ... in decr():
  if (this.counter < 0) {
    this stepperStyle =
      'color:red;font-size:'+this.counter+'em;';
```

- please note: *dCol* renamed *stepperStyle*
- *em* is a size unit in CSS

## How it is rendered

- 4 +

- 2 +



## Use a computed property

- keep the `incr()` and `decr()` functions focused on their task
- add a *computed property* to handle the color and size changes

```
methods: {  
  incr() { this.counter++; },  
  decr() { this.counter--; }  
},  
computed: {  
  stepperStyle() {  
    let color = this.counter < 0 ? 'red' : 'black';  
    return 'color:' + color + '; font-size:' +  
      Math.abs(this.counter) + 'em';  
  }  
}
```

- variable-like syntax, e.g.

```
<span :style="stepperStyle">{{ counter }}</span>
```

- reactively computed from other properties and reactive state
- automatically updates when its dependencies change

## Create Vue Component

- put all the above stuff in a file, call it *Stepper.vue*
- import *Stepper.vue* in the main file *App.vue*
- declare the component in the app's export:

```
export default {  
  components: {  
    Stepper  
  },  
}
```

- use the newly created element `<Stepper>`

# Stepper.vue

App.vue Stepper.vue × +

```
1 v <!-- STEPPER COMPONENT -->
2 v <script>
3 v export default {
4 v   data() {
5 v     return {
6       counter: 0,
7     }
8   },
9 v   methods: {
10 v     incr() {
11       this.counter++;
12     },
13 v     decr() {
14       this.counter--;
15     }
16   },
17 v   computed: {
18 v     stepperStyle() {
19       let color = this.counter < 0 ? 'red' : 'black';
20       return 'color:'+color+'; font-size:'+Math.abs(this.counter)+'em';
21     }
22   },
23 }
24
25 </script>
26
27 v <template>
28 v   <button @click="decr">
29     -
30   </button>
31 v   <span :style="stepperStyle">{{ counter }} </span>
32 v   <button @click="incr">
33     +
34   </button>
35 </template>
```



The image shows a code editor interface with two tabs: 'App.vue' (active) and 'Stepper.vue'. The 'App.vue' tab contains the following code:

```
1 <script>
2 import Stepper from './Stepper.vue';
3 export default {
4   components: {
5     Stepper
6   }
7 }
8 </script>
9
10 <template>
11 <h1>
12   My page
13 </h1>
14 <Stepper />
15 <br>
16 <Stepper />
17 </template>
```

On the right side of the editor, there are three tabs: 'Import Map', 'PREVIEW', and 'JS'. The 'PREVIEW' tab is active and displays the rendered output of the code, which is a large heading 'My page' followed by two small, empty rectangular buttons, each with a minus sign on the left and a plus sign on the right.

## Component attributes

- pass arguments to components, as attributes

```
<!-- in the app -->  
<Stepper title="My stepper">
```

```
// ... in the <script>  
  },  
  props: ['title']  
}
```

```
<!-- ... in the <template> -->  
<span style='background-color:yellow;'  
  {{ title }}:  
</span>
```

### My page

My stepper: - 2 +

My other stepper: - -1 +

## Many components

- we can use many `<Stepper>` components
- each one is instantiated separately, no conflicts
  
- let's use many steppers to count people that enters or exit rooms after the lessons started
- assume we have four lessons: wasa, deep learning, foundations, and cyber

## Data definition in the app's <script>

```
<script>
import Stepper from './Stepper.vue';
  export default {
    components: {
      Stepper
    },
    data() {
      return {
        courses: ['deep', 'foundations', 'cyber', 'wasa']
      }
    }
  }
</script>
```



## v-for: directive in the app's <Stepper> element

```
<template>
  <h1>
    Room entry/exit
  </h1>
  <Stepper v-for="course in courses" :title="course" />
</template>
```

### Room entry/exit

deep: -1 +

foundations: -3 +

cyber: - +

wasas: -2 +

- HTML forms can collect user input
- Vue can bind input data to variables using the *v-model* directive
- Let's create a way to add other courses to our list

## In the app's <template>

```
<!-- ... in the app's template -->
```

Add a course:

```
<input v-model="newCourse"
  placeholder="new course name">
<button
  :disabled="newCourse.length == 0"
  @click="add()">
  ADD
</button>
```

## Room entry/exit

deep: - +

foundations: - +

cyber: - +

was: - +

Add a course:

## Room entry/exit

deep: - +

foundations: - +

cyber: - +

was: - +

Add a course:

## Room entry/exit

deep: - +

foundations: - +

cyber: - +

was: - +

HCI: - +

Add a course:

## Logic to add new courses (<script>)

```
// ... in the app's script
data() {
  return {
    courses: ['deep', 'foundations', 'cyber', 'wasa'],
    newCourse: ""
  }
},
methods: {
  add() {
    if (this.newCourse !== "") {
      this.courses.push(this.newCourse);
    }
    this.newCourse = "";
  }
}
}
```

## Link to the playground of this lesson

[shorturl.at/doH17](https://shorturl.at/doH17)

- <https://vuejs.org>