WEB PROGRAMMING 0601632 2

BY DR BUNDIT THANASOPON

WHAT IS VUE.JS?

https://vuejs.org/v2/guide/index.html#what-is-vue-js

- Vue (pronounced /vjuː/, like **view**) is a **progressive framework** for building user interfaces.

- The core library is focused on the **view** layer only, and is easy to pick up and integrate with other libraries or existing projects.

- **Free videos:** https://www.vuemastery.com/courses/intro-to-vue-js/vue-instance/

LET'S GET STARTED

- CDN
  - Include this in the header section of your html file
    - `<script src="https://cdn.jsdelivr.net/npm/vue@2.6.12/dist/vue.js"></script>`
- NPM
  - NPM is the recommended installation method when building large scale applications with Vue.
    - $ npm install vue
- CLI
  - Vue provides an **official CLI** for quickly scaffolding ambitious Single Page Applications.
  - **Website:** https://cli.vuejs.org/guide/

THE VUE INSTANCE

CREATING A VUE INSTANCE

- Every Vue application starts by creating a new **Vue instance** with the **Vue** function:

```
var vm = new Vue({
  // options
})
```

- When you create a Vue instance, you pass in an "**options object**".
- For reference, you can also browse the full list of options in the **API reference**.

DATA

- When a Vue instance is created, it adds all the properties found in its **data** object to Vue's **reactivity system**. When the values of those properties change, the view will "react", updating to match the new values.

- When this data changes, the view will re-render.

```
data: {
  newTodoText: '',
  visitCount: 0,
  hideCompletedTodos: false,
  todos: [],
  error: null
}
```

DECLARATIVE RENDERING

- At the core of Vue.js is a system that enables us to declaratively render data to the DOM using straightforward template syntax:

The data and the DOM are now linked, and everything is now **reactive**.

- !!! It should be noted that properties in data are only **reactive** if they existed when the instance was created.

```
<div id="app">
  {{ message }}
</div>
```

```
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

INTERPOLATIONS

INTERPOLATIONS

- Text
  - The most basic form of data binding is text interpolation using the "Mustache" syntax (double curly braces):

```
<span>Message: {{ msg }}</span>
```

  - You can also perform one-time interpolations that do not update on data change by using the **v-once directive**

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

- Raw HTML

```
HTML
<p>Using mustaches: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

Using mustaches: This should be red.
Using v-html directive: This should be red.

```
rawHtml = '<span style="color: red;">Hello</span>'
```

INTERPOLATIONS

- Attributes
  - Mustaches cannot be used inside HTML attributes. Instead, use a **v-bind directive**:

```
<div v-bind:id="dynamicId"></div>

<button v-bind:disabled="isButtonDisabled">Button</button>
```

- Using JavaScript Expressions
  - Vue.js actually supports the full power of JavaScript expressions inside all data bindings:

```
{{ number + 1 }}

{{ message.split('').reverse().join('') }}

<div v-bind:id="'list-' + id"></div>
```

V-BIND DIRECTIVE

```
<div id="app-2">
  <span v-bind:title="message">
    Hover your mouse over me for a few seconds
    to see my dynamically bound title!
  </span>
</div>
```

```
var app2 = new Vue({
  el: '#app-2',
  data: {
    message: 'You loaded this page on ' + new Date().toLocaleString()
  }
})
```

- **Directives** are prefixed with v- to indicate that they are special attributes provided by Vue.
- They apply special reactive behavior to the rendered DOM.

SHORTHANDS

- **v-bind Shorthand**

```
<!-- full syntax -->
<a v-bind:href="url"> ... </a>

<!-- shorthand -->
<a :href="url"> ... </a>
```

- **v-on Shorthand**

```
<!-- full syntax -->
<a v-on:click="doSomething"> ... </a>

<!-- shorthand -->
<a @click="doSomething"> ... </a>
```

CONDITIONAL RENDERING

CONDITIONALS

- **v-if**
  - Conditionally render the element based on the truthy-ness of the expression value.
- **v-else**
  - **Restriction:** previous sibling element must have v-if or v-else-if.
- **v-else-if**
  - **Restriction:** previous sibling element must have v-if or v-else-if.
- **v-show**
  - Toggles the element's display CSS property based on the truthy-ness of the expression value.

```
<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>
```

CONDITIONAL GROUPS WITH V-IF ON <TEMPLATE>

- Because v-if is a directive, it has to be attached to a single element.
- But what if we want to toggle more than one element? In this case we can use **v-if** on a <template> element, which serves as an invisible wrapper. The final rendered result will not include the <template> element.

```
<template v-if="ok">
  <h1>Title</h1>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</template>
```

CONTROLLING REUSABLE ELEMENTS WITH KEY

- Vue tries to render elements as efficiently as possible, often re-using them instead of rendering from scratch. Beyond helping make Vue very fast, this can have some useful advantages. For example, if you allow users to toggle between multiple login types:

Then switching the **loginType** in the code above will not erase what the user has already entered.

Since both templates use the same elements, the **<input>** is not replaced - just its placeholder.

```
<template v-if="loginType === 'username'">
  <label>Username</label>
  <input placeholder="Enter your username">
</template>
<template v-else>
  <label>Email</label>
  <input placeholder="Enter your email address">
</template>
```

CONTROLLING REUSABLE ELEMENTS WITH KEY

- This isn't always desirable though, so Vue offers a way for you to say, "These two elements are completely separate - don't re-use them."
- Add a **key** attribute with unique values:

```
<template v-if="loginType === 'username'">
  <label>Username</label>
  <input placeholder="Enter your username" key="username-input">
</template>
<template v-else>
  <label>Email</label>
  <input placeholder="Enter your email address" key="email-input">
</template>
```

LIST RENDERING

MAPPING AN ARRAY TO ELEMENTS WITH V-FOR

- We can use the **v-for** directive to render a list of items based on an array.
- The **v-for** directive requires a special syntax in the form of **item in items**, where **items** is the source data array and **item** is an **alias** for the array element being iterated on:

```
<ul id="example-1">
  <li v-for="item in items" :key="item.message">
    {{ item.message }}
  </li>
</ul>
```

```
var example1 = new Vue({
  el: '#example-1',
  data: {
    items: [
      { message: 'Foo' },
      { message: 'Bar' }
    ]
  }
})
```

V-FOR WITH AN OBJECT

- You can also use v-for to iterate through the properties of an object.

```
<ul id="v-for-object" class="demo">
  <li v-for="value in object">
    {{ value }}
  </li>
</ul>
```

```
new Vue({
  el: '#v-for-object',
  data: {
    object: {
      title: 'How to do lists in Vue',
      author: 'Jane Doe',
      publishedAt: '2016-04-10'
    }
  }
})
```

LIMITATIONS - ARRAY

- Due to limitations in JavaScript, Vue **cannot** detect the following changes to an array:
  - When you directly set an item with the index,
    - e.g. vm.items[indexOfItem] = newValue
  - When you modify the length of the array,
    - e.g. vm.items.length = newLength

- To deal with limitation 1:
  - vm.items.splice(indexOfItem, 1, newValue)
- To deal with limitation 2:
  - vm.items.splice(newLength)

ARRAY CHANGE DETECTION

- Mutation Methods - Vue wraps an observed array's mutation methods so they will also trigger view updates. The wrapped methods are:
  - **push()**
  - **pop()**
  - **shift()**
  - **unshift()**
  - **splice()**
  - **sort()**
  - **reverse()**

LIMITATIONS - OBJECT

- **Vue cannot detect property addition or deletion!**
- Vue does not allow dynamically adding new root-level reactive properties to an already created instance. For example:

```
var vm = new Vue({
  data: {
    a: 1
  }
})
// `vm.a` is now reactive

vm.b = 2
// `vm.b` is NOT reactive
```

However, it's possible to add reactive properties to a nested object using the Vue.set(object, propertyName, value) method.

```
Vue.set(vm.someObject, 'b', 2)
```

MORE WITH V-FOR

- **v-for** with a Range – **v-for** can also take an integer. In this case it will repeat the template that many times.

```
<div>
  <span v-for="n in 10">{{ n }} </span>
</div>
```

Result:

1 2 3 4 5 6 7 8 9 10

- **v-for** on a <template> – Similar to template **v-if**, you can also use a <template> tag with **v-for** to render a block of multiple elements. For example:

```
<ul>
  <template v-for="item in items">
    <li>{{ item.msg }}</li>
    <li class="divider" role="presentation"></li>
  </template>
</ul>
```

V-FOR WITH V-IF

Note!!!
it's **not** recommended to use v-if and v-for together.

- When they exist on the same node, **v-for** has a higher priority than **v-if**. That means the v-if will be run on each iteration of the loop separately.

```
<li v-for="todo in todos" v-if="!todo.isComplete">
  {{ todo }}
</li>
```

Bad

```
<ul v-if="todos.length">
  <li v-for="todo in todos">
    {{ todo }}
  </li>
</ul>
<p v-else>No todos left!</p>
```

Good

- The example above only renders the todos

that are not complete.

CLASS & STYLE BINDINGS

BINDING HTML CLASSES OBJECT SYNTAX

- We can pass an object to **v-bind:class** to dynamically toggle classes:

```
<div v-bind:class="{ active: isActive }"></div>
```

- You can have multiple classes toggled by having more fields in the object.
- In addition, the **v-bind:class** directive can also co-exist with the plain class attribute. So given the following template:

```
<div
  class="static"
  v-bind:class="{ active: isActive, 'text-danger': hasError }"
></div>
```

```
data: {
  isActive: true,
  hasError: false
}
```

BINDING HTML CLASSES ARRAY SYNTAX

- We can pass an **array** to **v-bind:class** to apply a list of classes:

```
<div v-bind:class="[activeClass, errorClass]"></div>
```

```
data: {
  activeClass: 'active',
  errorClass: 'text-danger'
}
```

- If you would like to also toggle a class in the list conditionally, you can do it with a ternary expression:

```
<div v-bind:class="[isActive ? activeClass : '', errorClass]"></div>
```

```
<div v-bind:class="[{ active: isActive }, errorClass]"></div>
```

BINDING INLINE STYLES - OBJECT SYNTAX

- The object syntax for **v-bind:style** is pretty straightforward - it looks almost like CSS, except it's a JavaScript object. You can use either camelCase or kebab-case (use quotes with kebab-case) for the CSS property names:

```
<div v-bind:style="styleObject"></div>
```

```
data: {
  styleObject: {
    color: 'red',
    fontSize: '13px'
  }
}
```

```
<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>
```

```
data: {
  activeColor: 'red',
  fontSize: 30
}
```

BINDING INLINE STYLES - ARRAY SYNTAX

- The array syntax for **v-bind:style** allows you to apply multiple style objects to the same element

```
<div v-bind:style="[baseStyles, overridingStyles]"></div>
```