

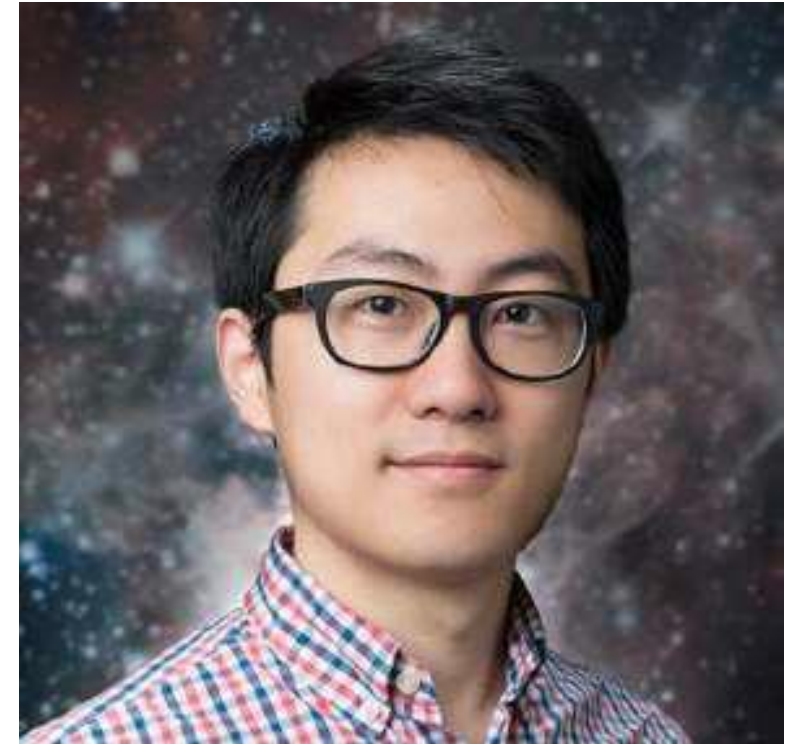
Unit-3 & Unit-4 Vue.js



The Founder

Evan You

- Previously worked as a Creative Technologist at **Google**
- Core Dev at **Meteor**
- From 2016 working **full-time** on **Vue.JS** framework.



How to choose the right framework?

- How **mature** are the frameworks / libraries?
- How **extensive and helpful** are their corresponding communities?
- How **easy** is it to **find developers** for each of the frameworks?
- What are the **basic programming concepts** of the frameworks?
- How easy is it to use the frameworks **for small or large applications**?
- What does the **learning curve** look like for each framework?
- What kind of **performance** can you expect from the frameworks?
- Where can you have a **closer look under the hood**?
- How **can you start developing** with the chosen framework?

Which big companies use Vue.js?

- GitLab
- Behance
- Livestorm
- Fox News
- Alibaba

Why Vue.js?

- Easy to start
- Vue template file component
- Virtual DOM
- Scoped css
- Built-in Transitions & Animation support
- Documentation
- Community
- Ecosystem

Ecosystem.

- vue-router - Single-page application routing
- vuex - Large-scale state management
- vue-cli - Project scaffolding
- vue-loader - Single File Component (*.vue file) loader for webpack
- vue-rx - RxJS integration
- vue-devtools - Browser DevTools extension
- vue-server-renderer - Server-side rendering support
- vue-class-component - TypeScript decorator for a class-based API

What is Vue.js?

- Vue (pronounced /vju:/, 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.
- On the other hand, Vue is also perfectly capable of powering sophisticated Single-Page Applications
- Progressive framework is a framework that you can insert into your project as you feel the need for it

Vue.js

- Vue is more simple and flexible
- Vue allows you make just specific parts of your application. You learn just what is necessary for the problem you are dealing with.
- Or if it is necessary and you have time, you can learn more and make a full complex front-end application 100% Vue.
- Only thing you need to know is HTML, CSS and JavaScript

Installing Vue.js

- You can install core library via CDN (Content Delivery Network)
- If you want to always follow up the latest version of Vue, use unpkg:
 - <https://unpkg.com/vue> - Always redirect to the latest version of Vue!
 - [https://unpkg.com/vue@\[version\]/dist/vue.min.js](https://unpkg.com/vue@[version]/dist/vue.min.js) - to specific version.
- You can install core library using Bower or npm



Vue.js

Vs



- **Vue** uses Javascript, **Angular** relies on TypeScript.
- **Vue** is pretty easy to learn, **Angular's** learning curve is much steeper
- **Vue** is indicated if you are working alone or have a small team, **Angular** is for really large applications.

What is a Single Page Application (SPA)?

- A single page application or SPA is a web application or a website that provides users a very fluid, reactive, and fast experience similar to a desktop application.
- A single page application contains a menu, buttons, and blocks on a single page. When a user clicks on any of them, it dynamically rewrites the current page rather than loading entire new pages from a server. That's the reason behind its reactive fast speed.
- Vue is basically developed for frontend development, so it has to deal with a lot of HTML, JavaScript, and CSS files. Vue.js facilitates users to extend HTML with HTML attributes called directives.
- Vue.js provides built-in directives and a lot of user-defined directives to enhance functionality to HTML applications.

Features of Vue.js

- Virtual DOM

- VueJS makes the use of virtual DOM, which is also used by other frameworks such as React, Ember, etc.
- The changes are not made to the DOM, instead a replica of the DOM is created which is present in the form of JavaScript data structures.
- The final changes are then updated to the real DOM, which the user will see changing. This is good in terms of optimization, it is less expensive and the changes can be made at a faster rate.

- Data Binding

- The data binding feature helps manipulate or assign values to HTML attributes, change the style, assign classes with the help of binding directive called **v-bind** available with VueJS.

- Components

- Vue.js Components are one of the important features of this framework. They are used to extend basic HTML elements to encapsulate reusable code. You can create reusable custom elements in Vue.js applications that can be later reused in HTML.

- Templates

- Vue.js provides HTML-based templates that can be used to bind the rendered DOM with the Vue instance data. All Vue templates are valid HTML that can be parsed by specification-compliant browsers and HTML parsers.
- Vue.js compiles the templates into Virtual DOM render functions. Vue renders components in virtual DOM memory before updating the browser.
- Vue can also calculate the minimum number of components to re-render and apply the minimum amount of DOM manipulations when you change the application's state.

- ## Reactivity

- Vue provides a reactivity system that uses plain JavaScript objects and optimizes re-rendering. In this process, each component keeps track of its reactive dependencies, so the system knows precisely when, and which components to re-render.

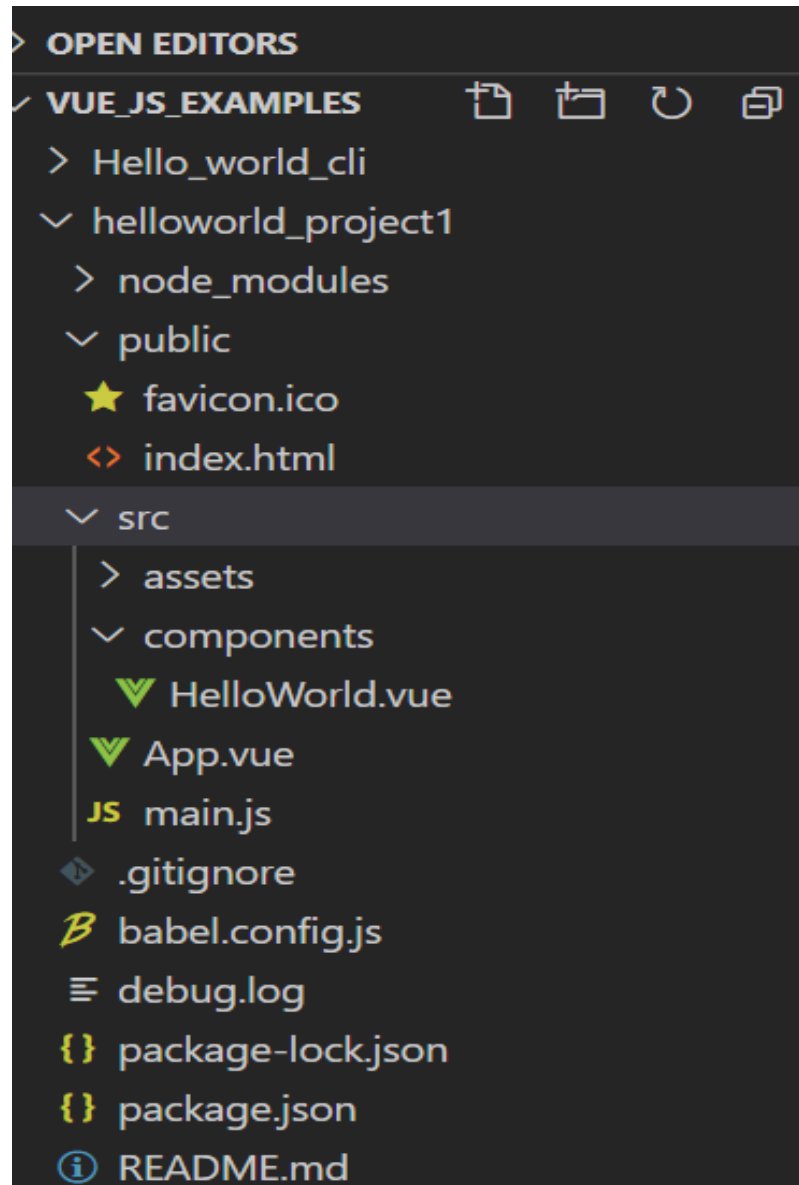
- ## Routing

- Navigation between pages is performed with the help of vue-router. You can use the officially-supported vue-router library for your Single Page Application or if you only need a simple routing and do not want to use the full-featured router library, you can do this by dynamically rendering a page-level component.

- ## Transitions

- Vue allows you to use different transition effects when the items are inserted, updated, or removed from the DOM.

Structure of the Project



Name	Description
public/index.html	This is the HTML file that is loaded by the browser. It has an element in which the application is displayed and a script element that loads the application files.
src/main.js	This is the JavaScript file that is responsible for configuring the Vue.js application. It is also used to register any third-party packages that the application relies on.
src/App.vue	This is the Vue.js component, which contains the HTML content that will be displayed to the user, the JavaScript code required by the HTML, and the CSS that styles the elements. Components are the main building blocks in a Vue.js Application
src/assets/logo.png	The assets folder is used to store static content, such as images. In a new project, it contains a file called logo.png, which is an image of the Vue.js logo.

Vue Component.

```
<!-- Basic Vue template - ./app.vue -->
<template>
  <div id="app">
    
    <HelloWorld msg="Welcome to Your Vue.js App"/>
  </div>
</template>
<script>
import HelloWorld from './components/HelloWorld.vue'
export default {
  name: 'app',
  components: {
    HelloWorld
  }
}
</script>
<style scoped>
div {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  color: #2c3e50;
}
</style>
```

Vue component VS Basic template.

```
<!-- Basic Vue template - ./app.vue -->
<template>
<div id="app">

<HelloWorld msg="Welcome to Your Vue.js
App"/>
</div>
</template>
<script>
import HelloWorld from './components/HelloWorld.vue'
export default {
name: 'app',
components: {
HelloWorld
}
}
</script>
<style scoped>
div {
font-family: 'Avenir', Helvetica, Arial, sans-serif;
color: #2c3e50;
}
</style>
```

```
!<!DOCTYPE html>
<html>
<head>
<script src="https://npmcdn.com/vue/dist/vue.js"></script>
</head>
<body>
<div id="app">
{{ msg }}
</div>
<script>
new Vue({
el: '#app',
data: {
msg: 'Hello World'
}
})
</script>
</body>
</html>
```

```
<!-- Basic Vue template - ./app.vue -->
```

```
<template>
```

```
<div id="app">
```

```

```

```
<HelloWorld msg="Welcome to Your Vue.js
```

```
App"/>
```

```
</div>
```

```
</template>
```

```
<script>
```

```
import HelloWorld from './components/HelloWorld.vue'
```

```
export default {
```

```
  name: 'app',
```

```
  components: {
```

```
    HelloWorld
```

```
  }
```

```
}
```

```
</script>
```

```
<style scoped>
```

```
div {
```

```
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
```

```
  color: #2c3e50;
```

```
}
```

```
</style>
```

View Template

Script

Style

Vue.JS

- A progressive framework for building interfaces
- Mvvm pattern with focus on the viewmodel, connecting View and model with two-way reactive data-binding
- Core values: reactivity, componentization, modularity,
- Simplicity and stability
- Simple API (easy to learn)
- Lightweight
- Rapid grow community

Core Features of Vue.js

- Reactivity
- Declarative Rendering
- Data Binding
- Directives
- Loops & Conditionals
- Component Encapsulation
- Event Handling
- Computed Properties & Watchers
- Transitioning Effects
- Custom Filters

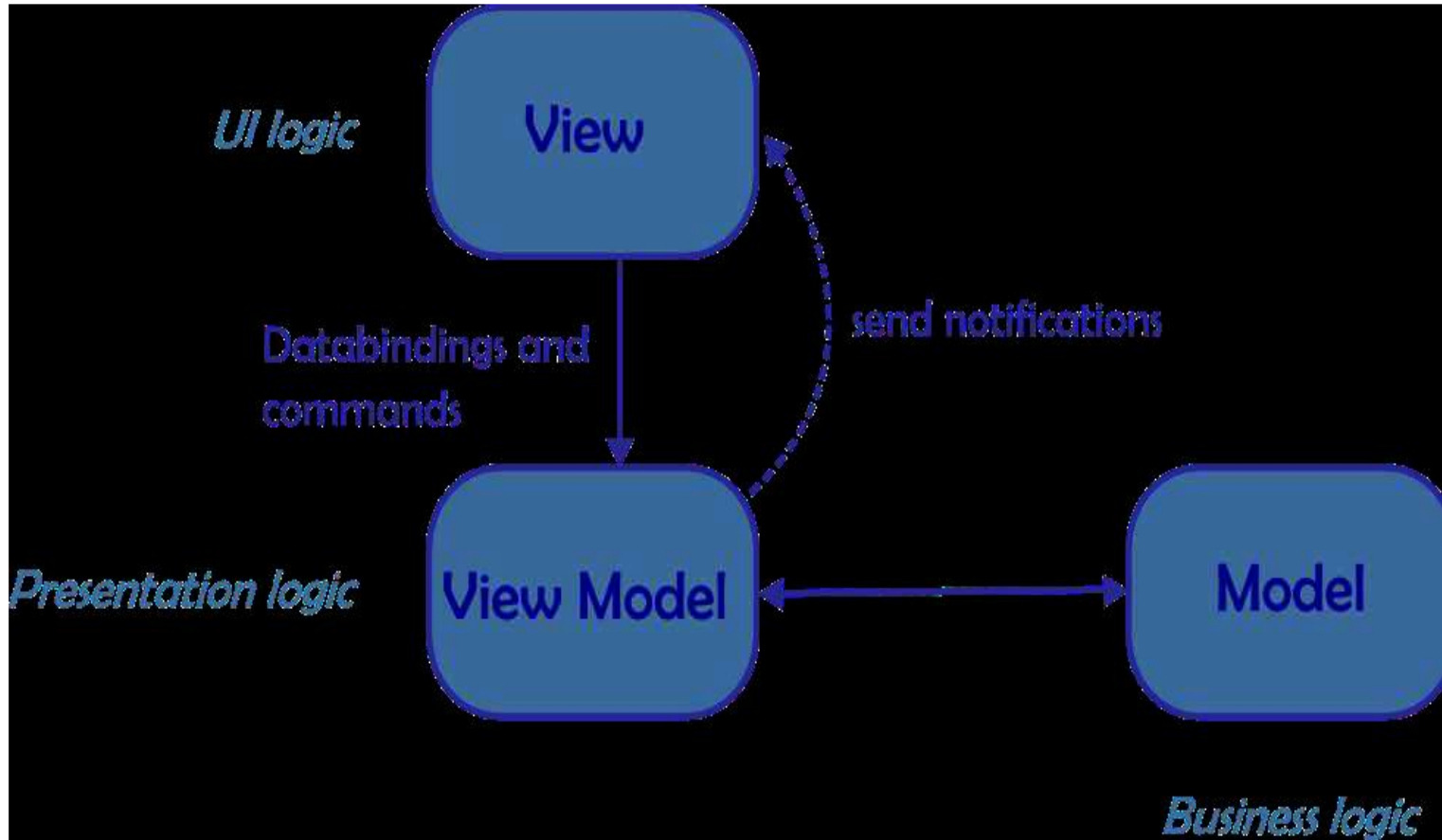
MVVM

- MVVM is not a design pattern it is an architectural pattern
- MVVM stands for Model-View-ViewModel
- Introduced by Microsoft in 2006.
- Also known as Presentation Model.
- MVVM guides you how to organize and structure your code to write maintainable, testable and extensible applications.

What is MVVM?

- **MVVM** Is a programming framework. Split it out: M-V-VM
- **M**: Model (data model layer)
- **V**: View (view interface layer)
- **VM**: ViewModel (View Model Layer)

ARCHITECTURAL DIAGRAM



- **Model:** It simply holds the data and has nothing to do with any of the business logic.
- **ViewModel:** It acts as the link/connection between the Model and View and makes stuff look pretty.
- **View:** It simply holds the formatted data and essentially delegates everything to the Model.

Model

- Non-visual classes that encapsulate the application's data and business logic.
- Can't See View Model or View.
- Should not contain any use case–specific or user task–specific behavior or application logic.
- Notifies other components of any state changes.
- May provide data validation and error reporting

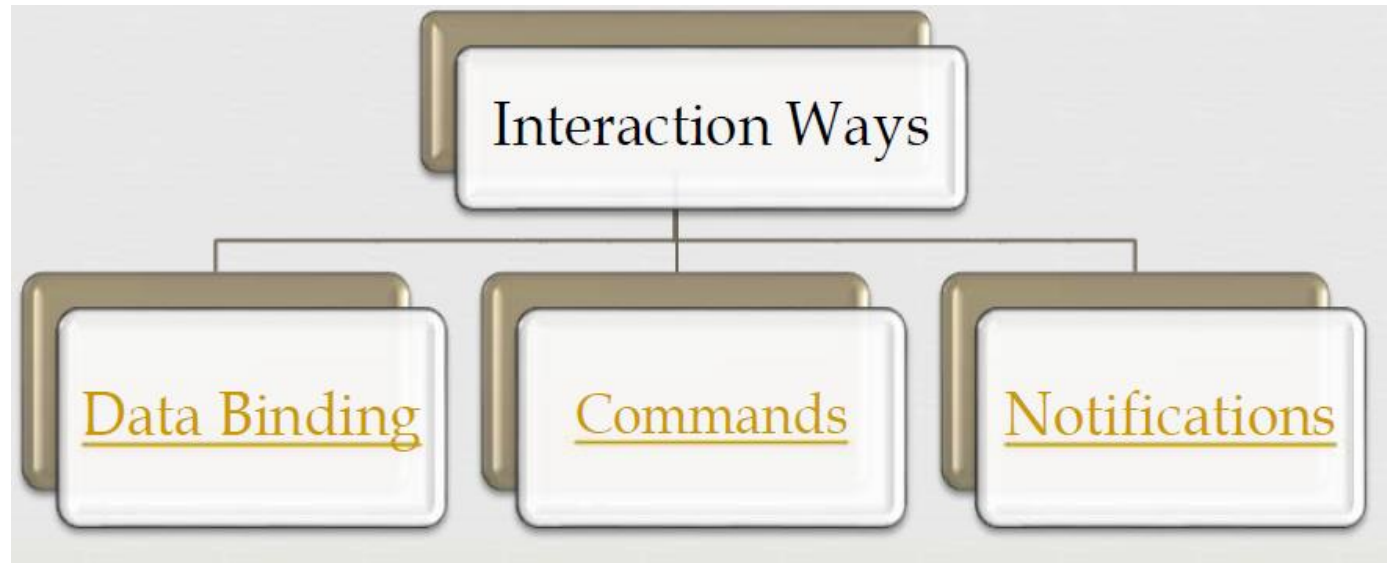
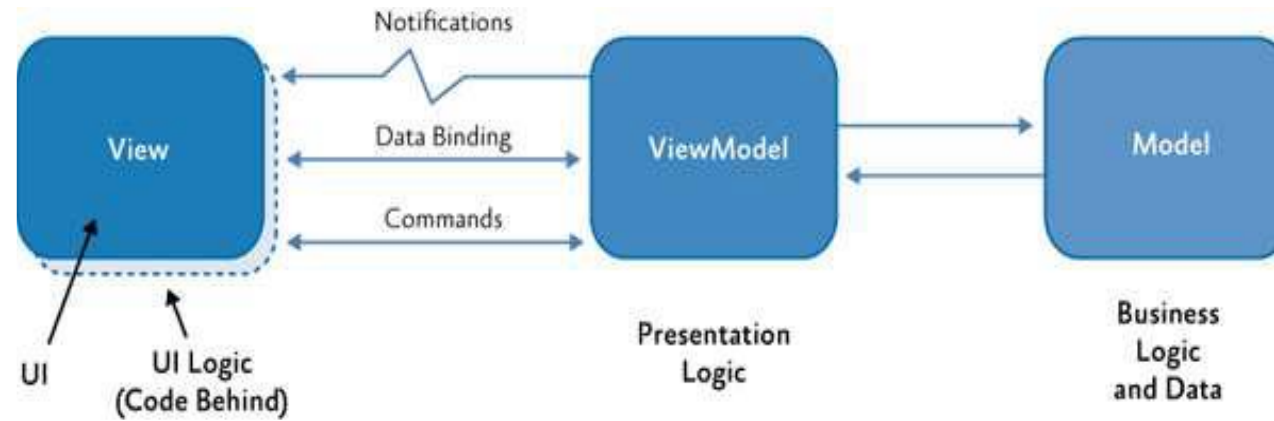
ViewModel

- Non-visual class encapsulates the presentation logic required.
- Can't See View (no direct Reference).
- Coordinates the view's interaction with the model.
- May provide data validation and error reporting.
- Notifies the view of any state changes.

View

- Visual element defines the controls and their visual layout and styling.
- Can see all others components.
- Defines and handles UI visual behavior, such as animations or transitions.
- Code behind may contain code that requires direct references to the specific UI controls.

Classes Interaction



Loosely Coupled

- The View knows the ViewModel but the ViewModel does not know the View.
- You can very easily replace the View without affecting the ViewModel.
- This is very useful in Developer/Designer teams where the Developer improves the ViewModel and the Designer enhances the View.

PROS & CONS

- Cons

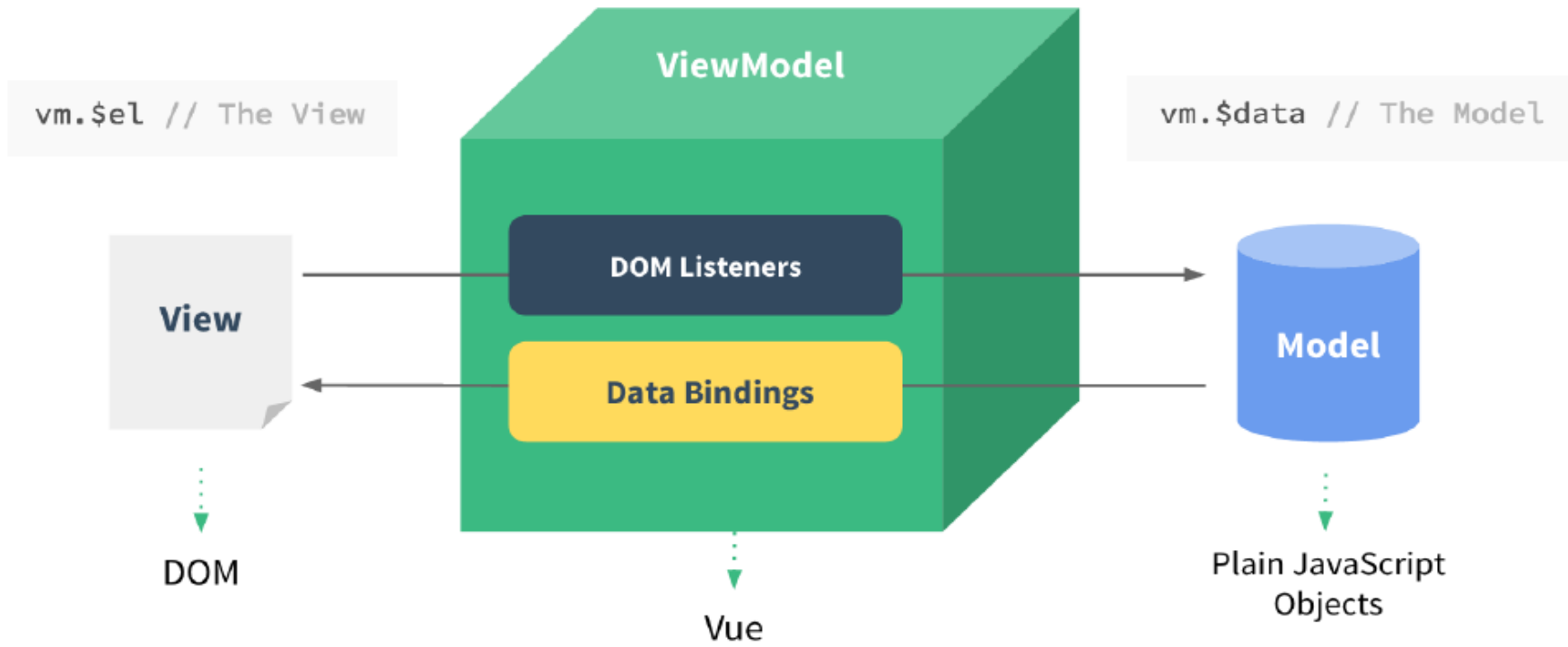
- Ease of Maintenance
- Ease of Testing
- Flexibility
- Re Usability
- Low Degree of coupling

- Pros

- Not Suitable for Simple Applications (Need best coding practices and standards)
- Not a standardized one , Everyone follows their own flavours

The Core Vue.js Framework

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



Creating a Vue.js application

- One of the best parts of using Vue.js is that it requires little overhead to get started.
- Add a script tag referencing the Content Delivery Network (CDN) for the library to your page and you are ready to get going!
- Empty HTML file

```
<html lang="en">
<head>
<title>Getting to Know Vue.js</title>
</head>
<body>
</body>
</html>
```

To take this empty HTML file to a working Vue.js app, we need to add three things:

- An HTML element where we “mount” our app
- A `<script>` reference to Vue.js on the CDN
- A `<script>` element in which we create our app

- We will start with a place to mount the app.
- We will use a <div> with an id of app.
- For the second one, we will use the development version of Vue.js at <https://cdn.jsdelivr.net/npm/vue/dist/vue.js>.
- The final one will be a JavaScript <script> element that we will use for all our JavaScript to get started.

```
<!-- Div to Mount App -->
```

```
<div id="app">
```

```
</div>
```

```
<!-- Reference to Vue.js library -->
```

```
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

```
<!-- Script Element for our first App -->
```

```
<script>
```

```
</script>
```

- The next step is to add some template syntax to our app's `<div>` to bind some data to it.
- To do this we use what is commonly called *mustache* syntax. It consists of two curly braces surrounding the property name we want to inject the data from, such as in `{{ propertyName }}`.

```
<!-- Div to Mount App -->
```

```
<div id="app">
```

```
  {{ propertyName }}
```

```
</div>
```

- Now we just need to create the app.
- In the empty `<script>` element we created, we are going to add a new instance of Vue.js, called `new Vue()`.

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

```
<script>  
  var vm = new Vue({  
    // options  
  });  
</script>
```

- To tell our instance of Vue.js where to mount the options object, we pass in a property called el.

The Element Option

- Inside of our Vue instance we can pass an option that specifies which HTML Element we want to attach to (Mount to).

```
<script>
  var vm = new Vue({
    el: '#app'
  });
</script>
```

- Here, we are telling our Vue instance to mount on an element with an ID of **app**.
- In the application created it won't have much functionality if we don't have any data, so it makes sense for our instance to have data option.

The Data Option

- We can set or define data that we want to use within our application by defining a **data** object inside the Vue instance created.

```
<script>
  var vm = new Vue({
    el: '#app'
    data: {
      name: 'AWP Class'
    }
  });
</script>
```

- Anything we define inside of our **data** object will be available for us to use in our application. We can use store strings, integers, arrays, boolean values, and even other objects.
- As you can see in the above example we added some data to our Vue instance. We added a key called **name** which contains a string with the value of **AWP Class**.
- Lastly, there are few hooks or functions that we can add to our Vue instance that can be called during various lifecycles of our instance.

Lifecycle Hooks

- How can we run some functionality after our Vue instance is **created**?
It can be done just by using **created** function/hook.

```
<script>
  var vm = new Vue({
    el: '#app'
    data: {
      name: 'AWP Class'
    },
    created: function(){
      console.log('Our new Vue instance has been created');
    }
  });
</script>
```

- There are few more important hooks that are used in instances which are **mounted, updated, and destroyed**.
- **Created:** called when the Vue instance is created.
- **Mounted:** called when the ‘#app’ element is mounted to the document.
- **Updated:** called anytime when the Vue instance or any data gets updated.
- **Destroyed:** called when the Vue instance is destroyed.

Instance Examples

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

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

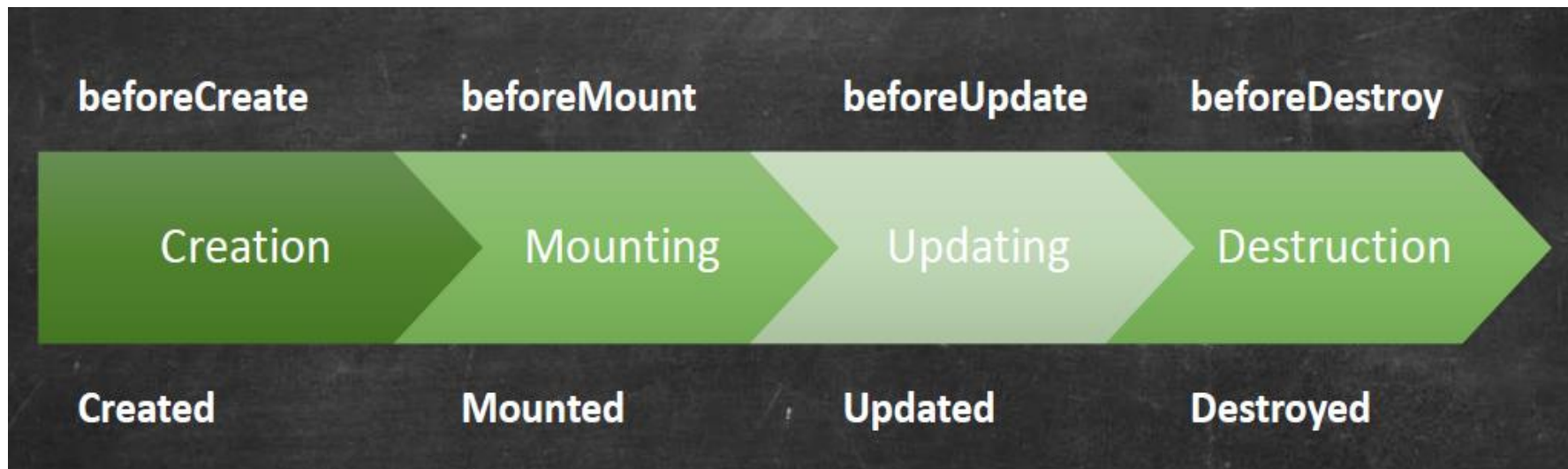
```
var vm = new Vue({
  data: {
    a: 1
  },
  created: function () {
    // `this` points to the vm instance
    console.log('a is: ' + this.a)
  }
})
// -> "a is: 1"
```

```
var vm = new Vue({
  data: {
    // declare message with an empty value
    message: ''
  },
  template: '<div>{{ message }}</div>'
})
// set `message` later
vm.message = 'Hello!'
```

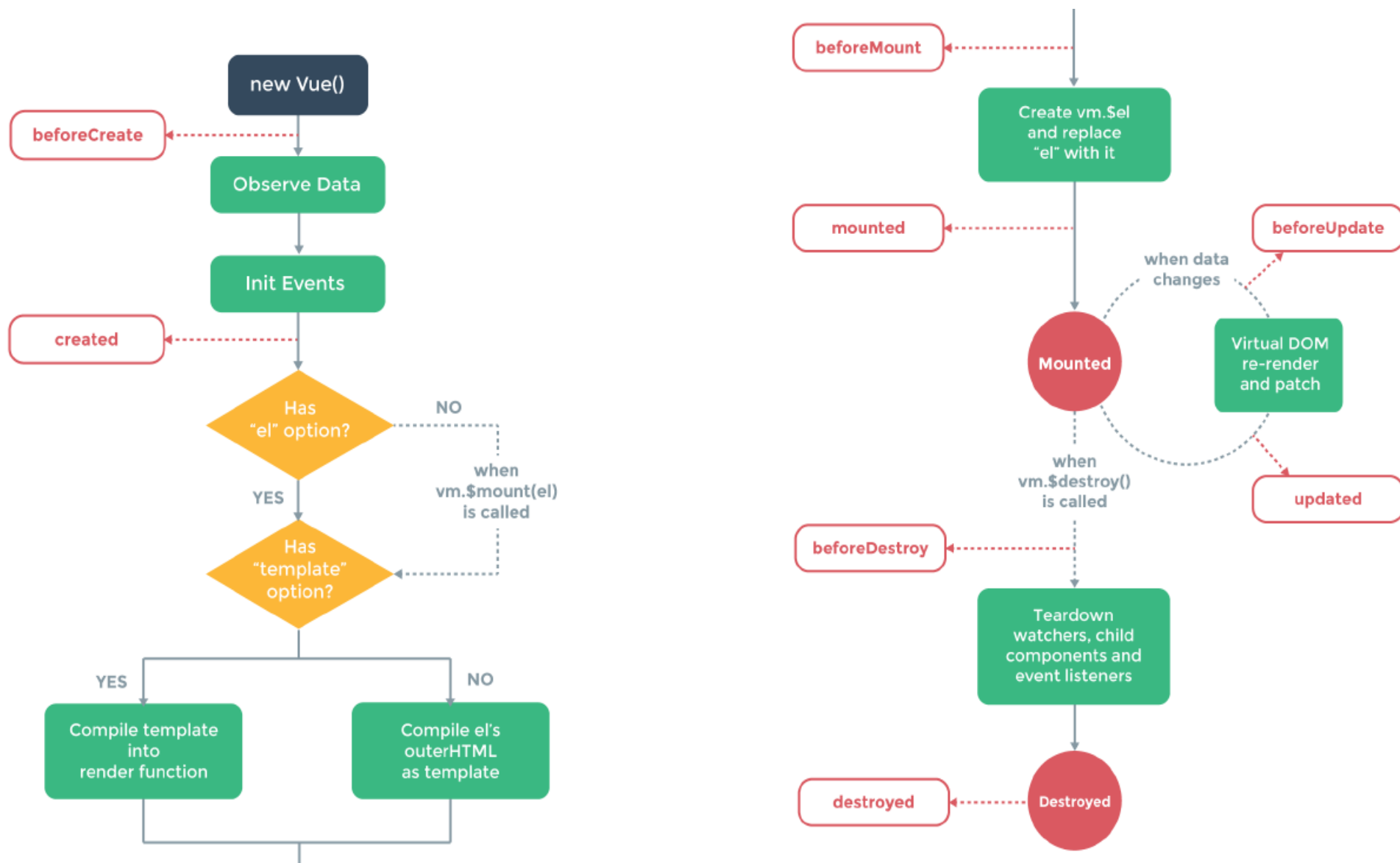
What are Lifecycle methods?

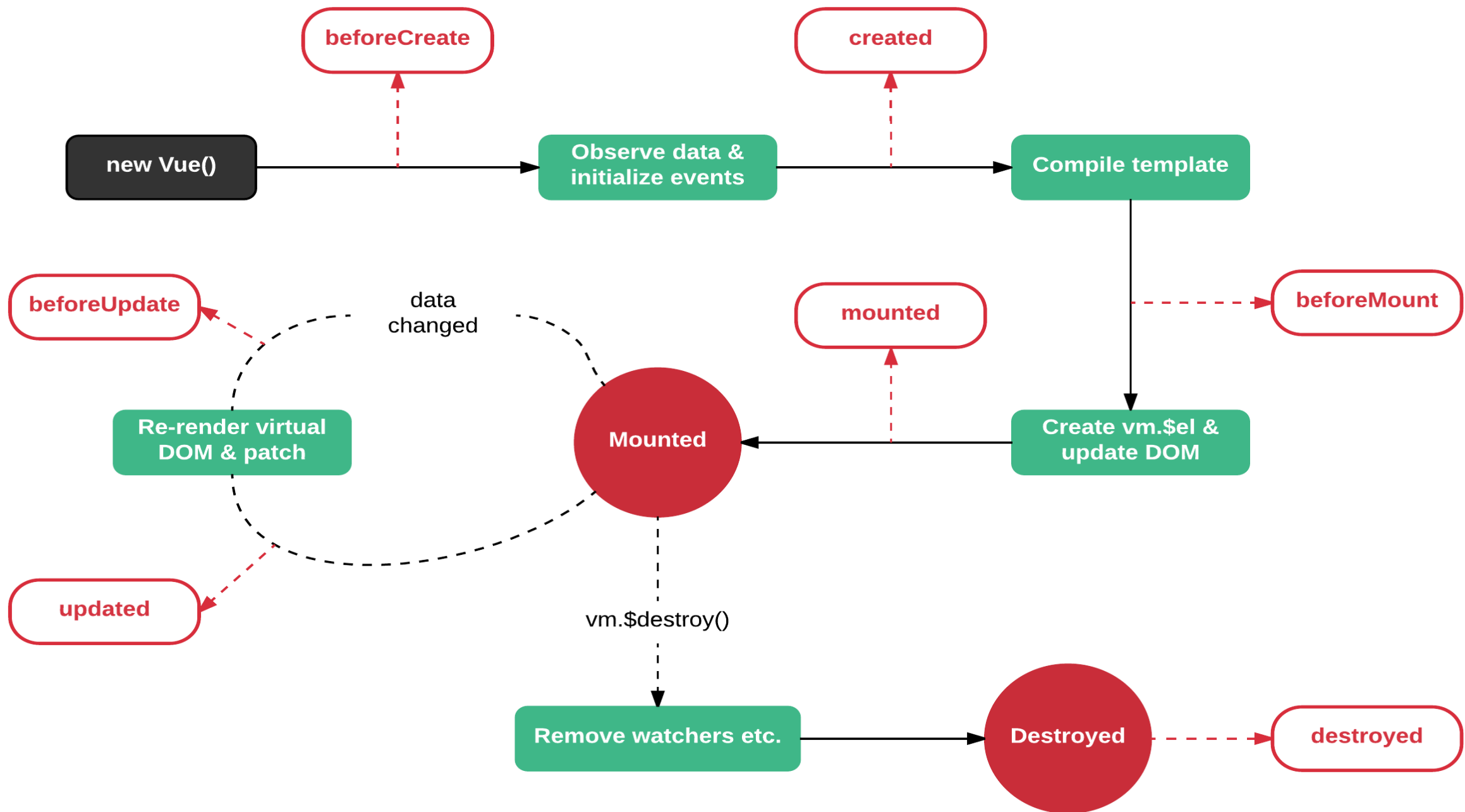
- Lifecycle methods serve as a viewpoint into how our built components work behind the scenes.
- You often need to know when your component is created, added to the DOM, updated or destroyed.
- They provide you methods that enable you trigger different actions at different junctures of a component's lifecycle.
- They also auto-bind to your component so that you can use the component's state and methods.
- Actions of lifecycle methods can be broken down into four categories:
 - Creation (Initialization)
 - Mounting (DOM Insertion)
 - Updating (Diff & Re-render)
 - Destruction (Teardown)

Lifecycle Diagram



Instance Lifecycle





Lifecycle Diagram - Creation

- Creation hooks are the very first hooks that run in your component.
- They allow you to perform actions before your component has even been added to the DOM.
- Unlike any of the other hooks, creation hooks are also run during server-side rendering.
- Use creation hooks if you need to set things up in your component both during client rendering and server rendering.
- You will **not** have access to the DOM or the target mounting element (`this.$el`) inside of creation hooks.

Lifecycle Diagram > Creation > beforeCreate

- The beforeCreate hook runs at the very initialization of your component. data has not been made reactive, and events have not been set up yet.

```
new Vue({
  data: {
    x: 'Vue Lifecycle hooks'
  },
  beforeCreate: function () {
    // `this` points to the view model instance
    console.log('x is: ' + this.x)
  }
})
// x is: undefined
```

Lifecycle Diagram > Creation > created

- In the created hook, you will be able to access reactive data and events are active. Templates and Virtual DOM have not yet been mounted or rendered.

```
new Vue({
  data: {
    x: 'Vue Lifecycle hooks'
  },
  created: function () {
    // `this` points to the view model instance
    console.log('x is: ' + this.x)
  }
})
// x is: Vue Lifecycle hooks
```


Lifecycle Diagram - Mounting

- Mounting hooks are often the most-used hooks, for better or worse.
- They allow you to access your component immediately before and after the first render.
- They do not, however, run during server-side rendering.
- **Use if:** You need to access or modify the DOM of your component immediately before or after the initial render.
- **Do not use if:** You need to fetch some data for your component on initialization. Use `useEffect` (or `useEffect + useLayoutEffect` for keep-alive components) for this instead, especially if you need that data during server-side rendering.

Lifecycle Diagram > Mounting > beforeMount

- The beforeMount hook runs right before the initial render happens and after the template or render functions have been compiled. Most likely **you'll never need** to use this hook.
- Remember, it doesn't get called when doing server-side rendering.

```
<script>
export default {
  beforeMount() {
    console.log(`this.$el doesn't exist yet, but it will soon!`)
  }
}
</script>
```

Lifecycle Diagram > Mounting > mounted

- In the mounted hook, you will have full access to the reactive component, templates, and rendered DOM (via. this.\$el). Mounted is the most-often used lifecycle hook.
- The most frequently used patterns are fetching data for your component (use created for this instead,) and modifying the DOM, often to integrate non-Vue libraries.

```
<template>
  <p>I'm text inside the component.</p>
</template>

<script>
export default {
  mounted() {
    console.log(this.$el.textContent) // I'm text inside the component
  }
}
</script>
```

Lifecycle Diagram - Updating

- Updating hooks are called whenever a reactive property used by your component changes, or something else causes it to re-render.
- They allow you to hook into the watch-compute-render cycle for your component.
- **Use if:** You need to know when your component re-renders, perhaps for debugging or profiling.
- **Do not use if:** You need to know when a reactive property on your component changes. Use **computed properties** or **watchers** for that instead.

Lifecycle Diagram > Updating > beforeUpdate

- The beforeUpdate hook runs after data changes on your component and the update cycle begins, right before the DOM is patched and re-rendered.
- It allows you to get the new state of any reactive data on your component before it actually gets rendered.

```
<script>
export default {
  data() {
    return {
      counter: 0
    }
  },

  beforeUpdate() {
    console.log(this.counter) // Logs the counter value every second
  },

  created() {
    setInterval(() => {
      this.counter++
    }, 1000)
  }
}
</script>
```

Lifecycle Diagram > Updating > updated

- The updated hook runs after data changes on your component and the DOM re-renders. If you need to access the DOM after a property change, here is probably the safest place to do it.

```
<template>
  <p ref="dom-element">{{counter}}</p>
</template>
<script>
export default {
  data() {
    return {
      counter: 0
    }
  },

  updated() {
    // Fired every second, should always be true
    console.log(+this.$refs['dom-element'].textContent === this.counter),

    created() {
      setInterval(() => {
        this.counter++
      }, 1000)
    }
  }
}
</script>
```

Lifecycle Diagram - Destruction

- Destruction hooks allow you to perform actions when your component is destroyed, such as cleanup or analytics sending. They fire when your component is being torn down and removed from the DOM.

Lifecycle Diagram > Destruction > beforeDestroy

- beforeDestroy is fired right before teardown. Your component will still be fully present and functional. If you need to cleanup events or reactive subscriptions, beforeDestroy would probably be the time to do it.

```
export default {  
  data() {  
    return {  
      awesomeMethod: 'I leak memory if not cleaned up!'  
    }  
  },  
  beforeDestroy() {  
    // Perform the teardown procedure for awesomeMethod (our case,  
    effectively nothing)  
    this.awesomeMethod = null  
    delete this.awesomeMethod  
  }  
}
```


Lifecycle Diagram > Destruction > destroyed

- By the time you reach the destroyed hook, there's pretty much nothing left on your component. Everything that was attached to it has been destroyed. You might use the destroyed hook to do any last-minute cleanup or inform a remote server that the component was destroyed.

```
export default {  
  destroyed() {  
    console.log(this) // There's practically nothing here!  
    AnalyticsService.bomb('Target acquired.')  
  }  
}
```

Lifecycle Diagram > Other Hooks

- There are two other hooks, activated and deactivated. These are for keep-alive components, a topic that is outside the scope of this presentation / syllabus.
- Suffice it to say that they allow you to detect when a component that is wrapped in a `<keep alive></keepalive>` tag is toggled on or off. You might use them to fetch data for your component or handle state changes, effectively behaving as `created` and `beforeDestroy` without the need to do a full component rebuild.

Instance Lifecycle Hooks - Warning!

- Don't use **arrow functions** on an options property or callback, such as
 - `created: () => console.log(this.a)` or `vm.$watch('a', newValue => this.myMethod())`.
- Since arrow functions are bound to the parent context, this will not be the Vue instance as you'd expect, often resulting in errors such as
 - Uncaught TypeError: Cannot read property of undefined or Uncaught TypeError: `this.myMethod` is not a function.

Recap

- The syntax for Vue is very simple. As an example we can output the value of name by using {{ }} curly braces.

```
<div id="example">
```

```
<p>{{ hello }}</p>
```

```
</div>
```

- Remember, we defined **hello** in our data object in the previous classes.
- Since our **hello** variable is wrapped inside of these {{ }} curly braces it will now output the value of **hello**, which is **Hello to AWP CLASS!**.
- Also, notice that the div has an id of app. This is important for our app to function correctly. This is the element that we binded to our Vue instance.
- Anytime we want to output data we can simply wrap them in {{ }}.

Vue.js Template

- In the Vue.js Instance concept, we have learned how to get an output in the form of text content on the screen.
- Now, we will learn how to get an output in the form of an HTML template on the screen.
- Vue.js uses an HTML-based template syntax that facilitates Vue.js developers to declaratively bind the rendered DOM to the underlying Vue instance's data.
- All Vue.js templates are valid HTML that can be parsed by spec-compliant browsers and HTML parsers.

Interpolations

- Text
- Raw HTML
- Attributes
- Using Javascript Expressions

Text

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

`Message: {{ msg }}`

- The mustache tag will be replaced with the value of the **msg** property on the corresponding data object. It will also be updated whenever the data object's **msg** changes.
- You can also perform one-time interpolations that do not update on data change by using the **v-once directive**, but keep in mind this will also affect any other bindings on the same node:

`This will never change: {{ msg }}`

Raw HTML

- The double mustache syntax shown above never interprets the data as HTML but as plain text. When we want to output the real HTML, we will need to use the v-html directive:

<p>Using mustaches: {{product}} </p>

<p>Using v-html directive: </p>

- The contents of the span will be replaced with the value of the product property, interpreted as plain HTML - data bindings are ignored.
- You cannot use v-html to compose template partials, because Vue is not a string-based templating engine.
- Rather, components are preferred as the fundamental unit for UI reuse and composition.

Attributes

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

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

- In the case of boolean attributes, where their mere existence implies true, v-bind works a little differently. In this example:

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

- If disabled has the value of null, undefined, or false, the disabled attribute will not even be included in the rendered <button> element.

Using JavaScript Expressions

- Thus far we have been binding to simple property keys in our templates only.
- But vue.js actually supports the full power of JavaScript expressions inside all data bindings:

```
{{ value + 1 }}  
{{ satisfied ? 'YES' : 'NO' }}  
{{ sentence.split('').reverse().join('') }}  
<ul v-bind:id="'product-'+id"></ul>
```

- The expressions above will be evaluated as JavaScript in the data scope of the owner Vue instance.

Directives and Data Binding

- A directive is some special token in the markup that tells the library to do something to a DOM element.
- A Vue.js directive can only appear in the form of a prefixed HTML attribute that takes the following format:

```
<element  
  prefix-directiveId="[argument:] expression [| filters...]">  
</element>
```

- Directive - A directive is a special attribute that adds dynamic data and functionality to our HTML elements. These attributes are typically prefixed with a “v-”. For example.

```
<div v-text="message"></div>
```

- Here the prefix is **v** which is the default. The directive ID is **text** and the expression is **message**. This directive instructs Vue.js to update the div’s `textContent` whenever the `message` property on the Vue instance changes.

Directives

- Reactive Directives

- V-text
- V-html
- V-show
- V-class
- V-attr
- V-style
- V-model
- V-if
- V-repeat

- Literal Directives

- V-transition
- V-ref
- V-el
 - Empty Directives
 - V-pre
 - V-cloak

Shorthand Syntax

- Shorthands allow us to add Vue functionality with an alternate syntax. Many directives have a shorthand syntax that we can use to make our application a little more readable.
- As an example instead of the v-bind directive:

```
<hl v-bind : id=" header_id"><h1>
```

- We can prefix our attribute with : and get the exact same functionality:

```
<hl : id=" header_id"><h1>
```

Filters

- Vue.js allows you to define filters that can be used to apply common text formatting.
- Filters are usable in two places: mustache interpolations and v-bind expressions.
- Filters should be appended to the end of the JavaScript expression, denoted by the “pipe” symbol:

`<p>ID {{ name | Upper }}</p>`

- The filters property of the component is an object. A single filter is a function that accepts a value and returns another value.
- The returned value is the one that’s actually printed in the Vue.js template.
- The filter, of course, has access to the component data and methods.

Why are Vue Filters important?

- **Enhancement of the presentation:** Vue JS is laser focused on the view layer of your application so it only makes sense that tools that enable you exert full control of the view layer is readily available for you to use at all times.
- **It is reusable:** you can declare a filter to be globally available and then you just use it in any desired component of choice inside the project. This is really important for efficiency and use of resources.
- **Data formatting:** Filters were built to equip you the developer with powers to be able to format your data at the view level. You know for instance when you make a HTTP get request, you can specify how the data should look in the template level. With filters, you can go a step further to still format that in the DOM itself.

- Filters are actually a feature provided by Vue.js that lets you apply common text formatting to your data.
- Filters do NOT change the data itself but rather change the output to the browser by returning a *filtered* version of that data.
- We can register filters in two different ways: **Globally and Locally**. The global gives you access to your filter across all your components, unlike the local which only allows you to use your filter inside the component it was defined in.
- Filters are simple JavaScript functions, they take the value to be transformed as the first parameter, but you can also pass in as many other arguments as you will need to return the formatted version of that value.

Local Filters

- Here is what a Local filter looks like:

```
filters: {  
  capitalize: function (value) {  
    if (!value) return ' '  
    value = value.toString()  
    return value.charAt(0).toUpperCase() + value.slice(1)  
  }  
}
```

Global filters

- Define a filter globally before creating the Vue instance:

```
Vue.filter('capitalize', function (value) {  
  if (!value) return ''  
  value = value.toString()  
  return value.charAt(0).toUpperCase() + value.slice(1)  
})  
new Vue({  
  // ...  
})
```

- When the global filter has the same name as the local filter, the local filter will be preferred.

Conditional Rendering With Vue.js

- The ability to show or hide elements based on conditions is a fundamental feature of any frontend framework. Vue.js provides us with a set of core directives to achieve this effect: **v-if**, **v-else**, **v-else-if** and **v-show**.
- **v-if**
 - The v-if directive adds or removes DOM elements based on the given expression.
- **v-else**
 - As the name v-else suggests, this directive is used to display content only when the expression adjacent v-if resolves to false.
- **v-else-if**
 - v-else-if can be used when we need more than two options to be checked. This will ensure that only one of the chained items in the else-if chain will be visible.

- **v-show**

- Very similar to v-if, the v-show directive can also be used to show and hide an element based on an expression.
- The main difference between the two is that,
 - v-if - Only renders the element to the DOM if the expression passes.
 - v-show - Renders all elements to the DOM and then uses the CSS **display** property to hide elements if the expression fails.
 - v-show - Does not support v-else, v-else-if
- Usually, v-show has a performance advantage if the elements are switched on and off frequently, while the v-if has the advantage when it comes to initial render time.

List Rendering With Vue.js

- 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:

HTML code

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

JavaScript

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

Result

- Foo
- Bar

- The v-for directive is used to render a list of items based on a data source. The directive can be used on a template element and requires a specific syntax along the lines of:

v-for = "*item* in *items*"



alias



data
collection

- We have full access to parent scope properties inside the v-for blocks. Vue-for supports a second argument which is used as an index of the current item.

```
<ul id="app-2">
  <li v-for="(item, index) in items">
    {{ parentMessage }} - {{ index }} - {{ item.message }}
  </li></ul>
var app2 = new Vue({
  el: '#app-2',
  data: {
    parentMessage: 'Parent',
    items: [
      { message: 'Foo' },
      { message: 'Bar' }
    ]
  }
})
```

Output

- Parent - 0 - Foo
- Parent - 1 - Bar

- ‘Of’ can be used as a delimiter instead of ‘in’:

```
<div v-for="item of items"></div>
```


v-for with an Object

- v-for can be used 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'
    }
  }
})
```

Output

- How to do lists in Vue
- Jane Doe
- 2016-04-10

- You can also provide a second argument for the property's name (a.k.a. key):

```
<div v-for="(value, name) in object">  
  {{ name }}: {{ value }}  
</div>
```

- And another for the index:

```
<div v-for="(value, name, index) in object">  
  {{ index }}. {{ name }}: {{ value }}  
</div>
```

Outputs

title: How to do lists in Vue
author: Jane Doe
publishedAt: 2016-04-10

0. title: How to do lists in Vue
1. author: Jane Doe
2. publishedAt: 2016-04-10

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>
```

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>
```

- **Reactive directives**: These directives can bind themselves to a property on the Vue instance, or to an expression which is evaluated in the context of the instance. When the value of the underlying property or expression changes, the **Update()** function of these directives will be called asynchronously on next tick.
- **Literal directives**: treat their attribute value as a plain string; they do not attempt to bind themselves to anything. All they do is executing the **bind()** function once. Literal directives can also accept mustache expressions inside their value.
- **Empty directives**: do not require and will ignore their attribute value.

Array Mutation methods in Vue.js

- Vue wraps an observed array's mutation methods so they will also trigger view updates.
- The wrapped methods are:
 - Push()
 - Pop()
 - Shift()
 - Unshift()
 - Sort()
 - Reverse()

- **Vue.js Array Push:** You can use `.push` method to add an element in array or object.
- **Vue.js Array pop:** Array Pop function is used to remove the last element from array and it returns that element.
- **Vue.js Array Shift:** This method removes an element from the beginning of array.
- **Vue.js Array Unshift:** This method adds the new elements at the beginning of array.
- **Vue.js Sort Array:** There are many ways to sort array objects in JavaScript.
- **Vue.js Array Reverse:** This method reverses the order of elements in array.

Replacing an Array in Vue.js

- Mutation methods, as the name suggests, mutate the original array they are called on.
- In comparison, there are also non-mutating methods, e.g. `filter()`, `concat()` and `slice()`, which do not mutate the original array but **always return a new array**.
When working with non-mutating methods, you can replace the old array with the new one.

- **Vue.Js Array Concat:** We can join two or more arrays using JavaScript concat method in Vue.js.
- **Vue.js Array Filter:** This method is used to create an array which is filled with elements that pass the test based on function passed.
- **Vue.js String Slice:** This method is basically used to extract a part of string from a string. We can use the native slice function to extract a part of string from a string.
- **Vue.js String Split:** JavaScript split function basically converts a string into array of substring. We can use the native JavaScript split function to convert a string into array of substring

Custom Directives in Vue

- Vue.js allows you to register custom directives, essentially enabling you to teach Vue new tricks on how to map data changes to DOM behavior.
- Vue allows you to create your own custom directive where you can then specify the action you want.
- They can come in handy in very large Vue applications where some actions abstracted to only a HTML element is needed throughout a Vue component.
- You can register a global custom directive with the **Vue.directive(id, definition)** method, passing in a directive id followed by a **definition object**.

Consider the following directive:

```
Vue.directive('focus', {  
  // When the bound element is inserted into the DOM...  
  inserted: function (el) {  
    // Focus the element  
    el.focus()  
  }  
})
```

- This example registers a new global custom directive into the main Vue instance.
- A custom directive is defined by a JavaScript literal object implementing a set of functions.
- These functions are called hooks by Vue JS and are standard to any custom directive.

Hook Functions

- All hook functions provided by Vue Js for building custom directives are optional.
- Hook functions are there to help you customize and provide the needed functionality for the directive at certain stages of the directive life cycle.
- There are five available:
 - bind
 - inserted
 - update
 - componentUpdate
 - unbind

- **bind:** This function is called once when the directive is bound to the underlying element. Think of it as a **one-time setup** hook.
- **inserted:** This is called when the underlying element is inserted into the parent node. This doesn't mean the element is inserted into the live DOM but rather its context is now known and part of a well-defined tree of nodes.
- **update:** This function is called after the containing component's VNode has updated, but possibly before its children have updated.
- **componentUpdate:** This is called after the containing component's VNode and the VNodes of its children have updated.
- **unbind:** This function is called only once when the directive is unbound from the element.
- Vue JS engine passes the same set of input parameters to all hook functions.

Binding Function Parameters

- Each and every hook function receives the same set of input parameters defined as follows.
- **el:** This parameter represents the element that this custom directive is applied to. It can be any valid HTML element.
- **binding:** This input parameter is an object containing the following properties:
 - **name:** The name of the directive without the **v-** prefix. For instance, using a custom directive as **v-focus** yields a name of **focus**.
 - **value:** The value passed to the directive. For instance, using the **v-slot="prop"** directive yields a value of **prop**.
 - **oldValue:** This field is only available inside **update()** and **componentUpdate()** hook functions. It contains the previous value of the directive, before the update.
 - **expression:** This field represents the expression of the binding as a string literal. For instance, using the custom directive **v-add="1+1"** yields an expression of **"1+1"**.

- **arg:** This field represents the argument (if any) that's passed to the directive. There can be only one argument passed. For instance, using the **v-slot:default** directive yields an argument of default.
- **modifiers:** This field is an object containing modifiers that could change and control the behavior of the directive if they are set. Think of modifiers as flags you set on the directive. If a modifier is set, it will have a value of **true**, if not set, it won't even be visible to the directive. For example, using the directive **v-on:click.prevent** yields a modifier of { **prevent: true** } object.
- **vnode:** The virtual node produced by Vue's compiler.
- **oldVnode:** The previous virtual node, only available in the **update()** and **componentUpdated()** hooks.

```
Vue.directive('focus', {  
  // When the bound element is inserted into the DOM...  
  inserted: function (el) {  
    // Focus the element  
    el.focus()  
  }  
})
```

- The **inserted()** hook function accepts the **el** input parameter. This parameter represents the HTML element where this custom directive is applied.
- Inside the function, the **focus()** function is called on the element itself.
- Overall, when the element with the custom directive is added to its parent node, this function runs and makes the element in the focus state.
- How do you apply this custom directive inside a component? Every custom directive should be prefixed by the letter **v-**. In this case, assuming we are adding this custom directive to an input control, then it follows like this:

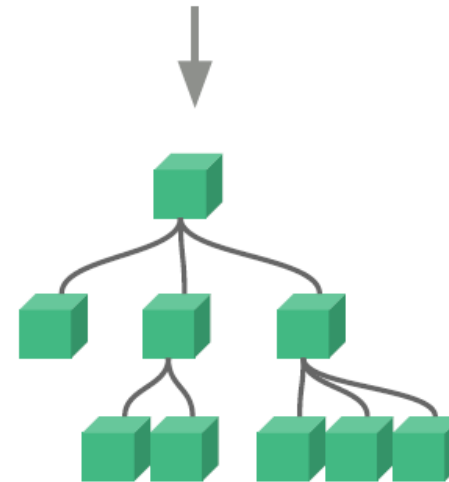
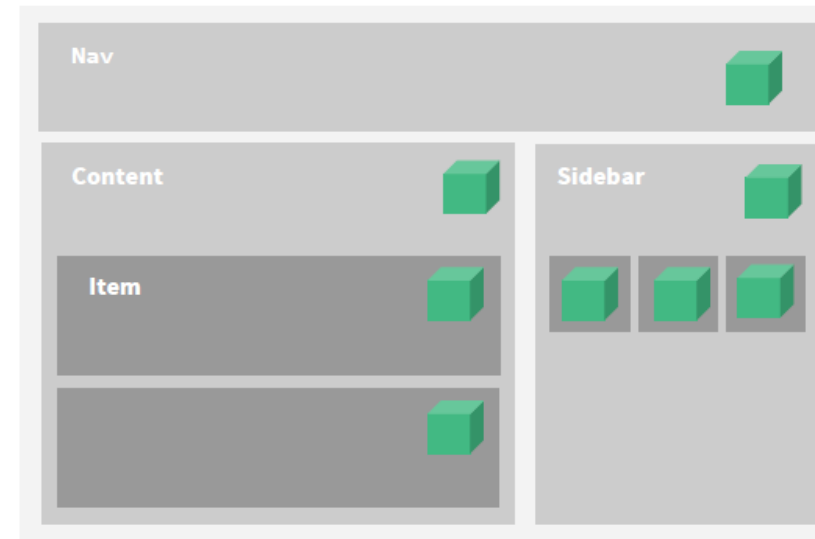
`<input v-focus>`

Components

- Components are one of the most powerful features of Vue.js.
- They help you extend basic HTML elements to encapsulate reusable code.
- A component is a self-contained, reusable, most often single-responsibility, piece of UI logic.

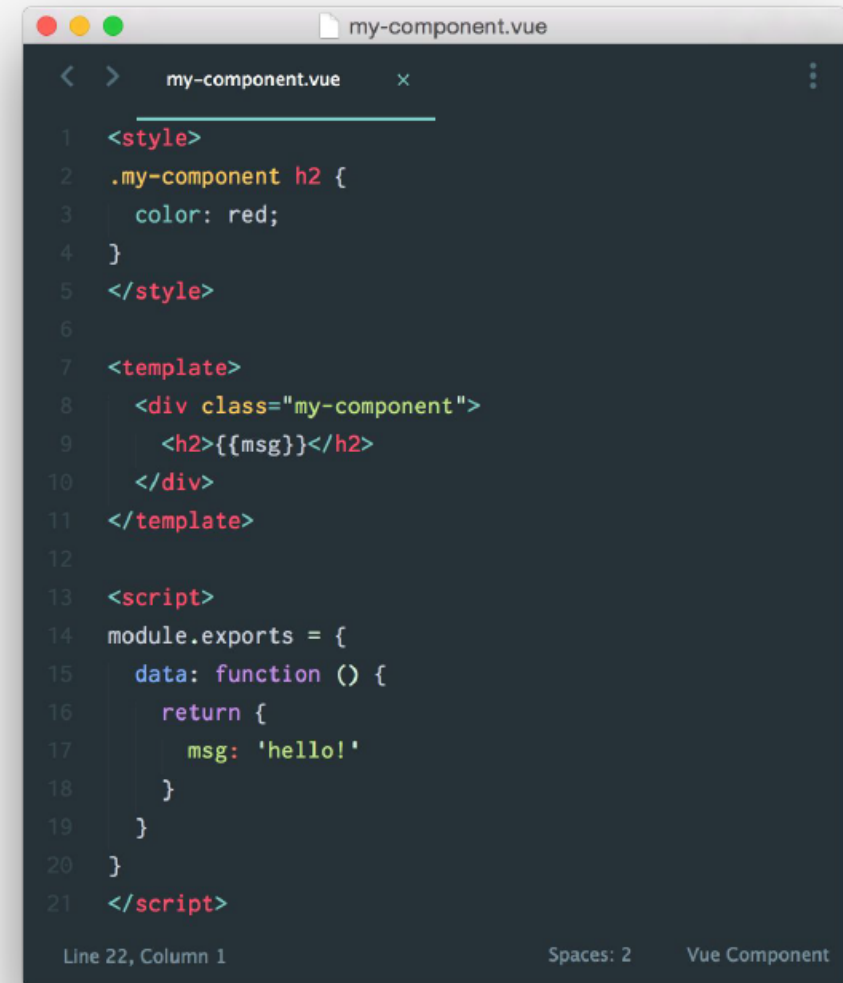
Components

- Small
- Self-contained
- Often reusable



Components

- Single File Components
- Reusability
 - Props
 - Content Slots
 - Dynamic Components



```
1 <style>
2 .my-component h2 {
3   color: red;
4 }
5 </style>
6
7 <template>
8   <div class="my-component">
9     <h2>{{msg}}</h2>
10   </div>
11 </template>
12
13 <script>
14 module.exports = {
15   data: function () {
16     return {
17       msg: 'hello!'
18     }
19   }
20 }
21 </script>
```

Line 22, Column 1 Spaces: 2 Vue Component

A Basic Component

- Let's start off by creating a very simple component. We can register this globally by using the **Vue.component(tagName, options)** like so:

```
Vue.component('simple-component', {  
  // All of my components options  
})
```

```
// followed by my vue instance:  
var vue = new Vue({  
  ...  
})
```

Creating your First Component

- The component which we will be developing will be a simple reusable rating counter that will illustrate how VueJS components work.

```
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<div id="app">
</div>
<script>
Vue.component('rating-counter', {
  data() {
    return {
      count: 0
    }
  },
  template: `<div>
    <button v-on:click="count--">Thumbs Down</button>
    {{ count }}
    <button v-on:click="count++">Thumbs Up</button>
  </div>`
})
var app = new Vue({
  el: '#app'
})
</script>
```

Use the Component

- Let's try the new component out. In our ratingcounter.html created, update the app div as follows:

```
<div id="app">  
  <rating-counter></rating-counter>  
  <rating-counter></rating-counter>  
  <rating-counter></rating-counter>  
</div>
```

Using Component Props

- Props are VueJS's method for adding custom properties to the component created. [example_component6_props1.html](#)

```
Vue.component('rating-counter', {  
  props: ['title'],  
  data() {  
    return {  
      count: 0  
    }  
  },  
  template: `

<h1>{{ title }}</h1>  
    <button v-on:click="count--">Thumbs Down</button>  
    {{ count }}  
    <button v-on:click="count++">Thumbs Up</button>  
  </div>`  
})


```

- 1) In ratingcounter.html, update your Vue.component() as shown in previous slide.
- 2) Just pass title as an attribute to your component's opening tag. Whatever you pass there would be accessible by your component.

```
<div id="app">  
  <rating-counter title="Rating 1"></rating-counter>  
  <rating-counter title="Rating 2"></rating-counter>  
  <rating-counter title="Rating 3"></rating-counter>  
</div>
```

Sharing Data Between Components and the Parent App

- Just as a Vue component we can keep track of data used in that component, the Vue app itself can also maintain its own data object. [example_component6_sharing_data.html](#)
- The parent app's data is set on lines 29-34. The app now keeps track of an object called parentHeader.
- The data from this object is rendered on line 4.
- On line 13, we've added another prop to the component, called parent.
- On lines 6-8, the value for this prop is assigned with the v-bind:parent attribute. By using the v-bind syntax, you're telling VueJS to bind the parent attribute of the component to whichever data property you supply, in this case the parentHeader object.
- On lines 21 and 23, the on-click actions for each button will increment or decrement the globalCount property of the parent prop, which corresponds to the globalCount property of the parentHeader object in your app's data.
- Because props are reactive, changing this data from the component will cascade the changes back to the parent, and to all other components that reference it.

Using Slots

- Slots are another very clever way to pass data from parent to components in VueJS. [example_component6_slots.html](#)
- Instead of using attributes as you did before, you can pass data within the component's opening and closing HTML tags.

Nesting Slots

- The most important feature of slots might be the ability to use components within components. This is especially useful when creating structure for your apps. [example_component6_Nested_slots.html](#)
- Instead of using attributes as you did before, you can pass data within the component's opening and closing HTML tags.

Named Slots

- To allow even more structure, you can use multiple slots in a template by using slot naming. Lets overengineer our simple example a little bit to see how it works. [example_component6_Named_slots.html](#)

Using Component Events

- Events in components are essential for component-to-parent communication in VueJS. Whenever a component needs to communicate back to its parent, this is the safe way of doing that.

[example_component6_com_events.html](#)

- In the v-on:click event of our component (line 13), we can see that this time we're using VueJS's \$emit method instead of changing variables manually. The \$emit() method takes a custom event name as an argument (increment in this example).
- This fires an empty increment event, which our parent is subscribed to on line 7. Notice the v-on:increment="parentHeader.globalCount++" attribute; this is our event subscriber.
- The subscriber can call a method of the VueJS app, or in this example, just directly use the increment JavaScript ++ operator to increase the counter.
- Because the component no longer directly manipulates the parent's globalCount data, the parent prop for the component can be removed.

Passing a Parameter with Events

- [example_component6_passing_events.html](#) example introduces a second parameter of the `$emit()` function (lines 13-14).
- Instead of simply incrementing by one, our parent event subscriber can make use of this argument with the `$event` variable (line 7).
- The template for the component now has a Thumbs Down button again, and the argument for this button's event is -2 (line 13).