Vue.js Core Concepts

Data Binding: One-Way

One-way data binding means data flows in a single direction: from the component's data to the UI. When the data changes, the UI updates, but not the other way around.

- **Directive**: `v-bind:` or the shorthand `:`.
- **Use Case**: Displaying data, binding attributes.

```
<div id="app">
  <!-- Binds the 'title' attribute to the 'message' data property -->
  <span :title="message">
    Hover me to see the dynamic title!
  </span>
</div>
<script>
new Vue({
  el: '#app',
  data: {
    message: 'Page loaded on: ' + new Date().toLocaleString()
  }
});
</script>
```

One-Way Binding: Class and Style

You can use `v-bind` to dynamically toggle classes or apply inline styles.

- **Object Syntax**: For binding multiple classes or styles at once.
- **Array Syntax**: For applying a list of classes.

```
<div id="app">
  <!-- Bind a class object. 'active' is applied if isActive is true. -->
  <div :class="{ active: isActive, 'text-danger': hasError }">Classy</div>

  <!-- Bind a style object. -->
  <div :style="{ color: activeColor, fontSize: fontSize + 'px' }">Stylish</div>
</div>
<script>
new Vue({
  el: '#app',
  data: {
    isActive: true,
    hasError: false,
    activeColor: 'blue',
    fontSize: 20
  }
});
</script>
```

Data Binding: Two-Way

Two-way data binding creates a synchronized link between data and the UI. If the user changes the input, the data updates. If the data changes, the input updates.

- **Directive**: `v-model`.
- **Use Case**: Form inputs like `<input>`, `<textarea>`, and `<select>`.

```
<div id="app">
  <input v-model="message" placeholder="Edit me">
  <p>The message is: {{ message }}</p>
</div>
<script>
new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
});
</script>
```

Two-Way Binding: `v-model` Modifiers

`v-model` has modifiers to change its behavior.

- **.lazy**: Updates the data on `change` events instead of `input`. (e.g., after blurring the input)
- **.number**: Automatically typecasts the input value to a number.
- **.trim**: Automatically trims whitespace from the input.

```
<div id="app">
  <!-- Updates only after you click away -->
  <input v-model.lazy="lazyMessage" placeholder="Lazy update">

  <!-- Input is treated as a number -->
  <input v-model.number="age" type="number" placeholder="Age">

  <!-- Whitespace is trimmed -->
  <input v-model.trim="trimmedMessage" placeholder="Trimmed">
</div>
```

Computed Properties

Computed properties let you create new data based on existing data. They are **cached** and only re-calculate when their dependencies change, making them very efficient.

- **Use Case**: For complex logic in your template. For example, reversing a string or filtering a list.

```
<div id="app">
  <p>Original: "{{ message }}"</p>
  <p>Reversed: "{{ reversedMessage }}"</p>
</div>
<script>
new Vue({
  el: '#app',
  data: { message: 'Hello' },
  computed: {
    // This computed getter reverses the message
    reversedMessage: function () {
      return this.message.split('').reverse().join('');
    }
  }
});
</script>
```

Computed Properties vs. Methods

Why use a computed property instead of a method?

- **Caching**: Computed properties are **cached**. They only re-run when a dependency changes. A method, on the other hand, is **always** re-run every time the component re-renders.
- **Performance**: Use computed properties for expensive operations to avoid re-calculating on every render. Use methods when you don't need caching or when you need to pass arguments.

```
<!-- Computed: efficient, cached -->
<p>{{ reversedMessage }}</p>

<!-- Method: re-runs on every render -->
<p>{{ reverseMessageMethod() }}</p>
```

Computed Properties: Setters

While computed properties are getters by default, you can also provide a **setter** for when you need to update the underlying data.

- **Use Case**: When you want to change the source data based on a computed property's new value.

```
<div id="app">
  <input v-model="fullName"> <!-- This now works both ways! -->
</div>
<script>
new Vue({
  el: '#app',
  data: { firstName: 'John', lastName: 'Doe' },
  computed: {
    fullName: {
      get: function() { return this.firstName + ' ' + this.lastName; },
      set: function(newValue) {
        var names = newValue.split(' ');
        this.firstName = names[0];
        this.lastName = names[names.length - 1];
      }
    }
  }
});
</script>
```

Watchers

A **watcher** is a feature that lets you perform an action when a specific data property changes.

- **Use Case**: Use watchers for **asynchronous** or **expensive** operations, like making an API call when data changes.
- Unlike computed properties, watchers don't return a value.

```
<div id="app">
  <p>Type something:</p>
  <input v-model="valueToWatch">
  <p>{{ feedback }}</p>
</div>
<script>
new Vue({
  el: '#app',
  data: { valueToWatch: '', feedback: '' },
  watch: {
    // This function runs whenever 'valueToWatch' changes
    valueToWatch: function(newValue, oldValue) {
      this.feedback = `Typing...`;
      // You could add a debounce here to make an API call
    }
  }
});
</script>
```

Watchers: Advanced Options

Watchers can take an options object for more control.

- **deep**: To detect changes inside objects or arrays, set `deep: true`.
- **immediate**: To run the watcher callback immediately on component load (before any changes), set `immediate: true`.

```
watch: {
  someObject: {
    // This handler will be called on load and on deep mutations
    handler(newValue, oldValue) {
      console.log('Object changed!');
    },
    deep: true,
    immediate: true
  }
}
```

Components, Templates & Props

What is a Component?

Components are **reusable Vue instances** with a name. They are the core building blocks of a Vue application, allowing you to create a tree of nested components.

- **Encapsulation**: Each component has its own data, logic, and template.
- **Reusability**: Write a component once and use it anywhere.

```
// Register a global component called 'my-button'
Vue.component('my-button', {
  data: function() {
    return {
      count: 0
    }
  },
  template: '<button @click="count++">You clicked me {{ count }} times.</button>'
});
```

```
<!-- Use it in your HTML -->
<div id="app">
  <my-button></my-button>
  <my-button></my-button>
</div>
```

Global vs. Local Registration

There are two ways to register components:

- **Global Registration**: Using `Vue.component()`. These components can be used anywhere in your application, in any other component's template.
- **Local Registration**: Registered inside a component's `components` option. These components are only available within the scope of the component that registers them.

Best Practice: Prefer **local registration**. It makes dependencies explicit and is better for performance and code organization. Use global registration only for very common base components (like buttons, inputs, etc.).

Local Registration Example

Here, `component-b` is registered locally and is only available inside `component-a`.

```
const ComponentB = {
  template: '<p>I am Component B!</p>'
}

const ComponentA = {
  components: {
    'component-b': ComponentB // Locally register ComponentB
  },
  template: `
    <div>
      <h1>I am Component A</h1>
      <component-b></component-b>
    </div>
  `
}

new Vue({
  el: '#app',
  components: {
    'component-a': ComponentA // Register ComponentA in the root instance
  }
})
```

Component Data Must Be a Function

The `data` option in a component **must be a function** that returns an object.

Why? If `data` were just an object, all instances of that component would share the **same data object**. By using a function, each component instance gets its own, unique copy of the data object, preventing state from being shared unintentionally.

```
// Correct: data is a function
data: function() {
  return {
    message: 'This is unique to each component instance'
  }
}

// WRONG: data is an object (this will cause bugs!)
// data: {
//   message: 'This message is shared by ALL component instances'
// }
```

Component Templates

A component's UI is defined by its `template` option.

- **Single Root Element**: A component's template **must** have a single root element. A `<div>` is often used to wrap the content.
- **Template Options**:
  i. **Inline Strings**: `template: '<div>...</div>'` (Good for small templates).
  ii. **Template Literals**: `template: ...`` (Good for multi-line templates).
  iii. **`x-templates`**: Using a script tag with `type="text/x-template"`.
  iv. **Single File Components (`.vue` files)**: The modern standard. Combines template, script, and style in one file. Requires a build step (e.g., with Vue CLI).

What are Props?

Props (short for properties) are custom attributes you can register on a component. They are the primary way to pass data **from a parent component down to a child component**.

- **One-Way Data Flow**: Data flows down from parent to child. A child should not mutate a prop directly.

```
// The child component declares the props it accepts
Vue.component('blog-post', {
  props: ['title', 'author'],
  template: '<h3>{{ title }} by {{ author }}</h3>'
});
```

```
<!-- The parent passes data to the child's props -->
<div id="app">
  <blog-post title="My Vue Journey" author="Jane Doe"></blog-post>
  <blog-post title="Advanced Vue" author="John Smith"></blog-post>
</div>
```

Prop Validation

It's a best practice to specify validation for your props. This helps catch common errors and makes your components easier to use.

- **Object Syntax**: Instead of an array of strings, use an object for validation.
- **Validators**: `type`, `required`, `default`, `validator` function.

```
Vue.component('user-profile', {
  props: {
    name: {
      type: String,       // Must be a String
      required: true      // This prop is mandatory
    },
    age: {
      type: Number,       // Must be a Number
      default: 18         // Default value if not provided
    },
    status: {
      // Custom validation function
      validator: function(value) {
        return ['active', 'inactive', 'pending'].includes(value);
      }
    }
  },
  template: '<div>...</div>'
});
```

Application State Management

Application State

State is the data that drives your application—it's the single source of truth.

- **Local State**: Data managed inside a single component.
- **Global State**: Data shared across many components.

For simple apps, you can pass data from parent to child using **props**.

For larger, more complex apps, a state management library like **Vuex** is the standard. It creates a central "store" for all your application's state, making it easier to manage and predict.

State: Parent-to-Child (Props)

Props are the most basic way to pass data from a parent component to a child component. Data flows down.

```
// Child Component
Vue.component('child-component', {
  props: ['message'], // Declare the prop
  template: '<p>{{ message }}</p>'
});

// Parent Vue Instance
new Vue({
  el: '#app',
  data: {
    parentMessage: 'Hello from the parent!'
  }
});
```

```
<!-- In your HTML -->
<div id="app">
  <child-component :message="parentMessage"></child-component>
</div>
```

State: Child-to-Parent ($emit)

A child component can't directly change a parent's data. Instead, it should **emit an event** that the parent can listen for. This maintains the one-way data flow principle.

```
// Child Component
Vue.component('child-component', {
  template: '<button @click="$emit(\'update-parent\', \'New message!\')">Update</button>'
});

// Parent Vue Instance
new Vue({
  el: '#app',
  data: { message: 'Old message' },
  methods: {
    handleUpdate(newMessage) {
      this.message = newMessage;
    }
  }
});
```

```
<div id="app">
  <p>{{ message }}</p>
  <child-component @update-parent="handleUpdate"></child-component>
</div>
```

The Problem with `props` and `$emit`

In large applications, you might have deeply nested components that all need access to the same piece of state.

- **Prop Drilling**: You have to pass the prop down through every level of the component tree, even through components that don't need it.
- **Complex Event Chains**: To update state from a deep child, you have to emit an event chain (`$emit`, `$emit`, `$emit`) all the way back up to the parent.

This becomes very difficult to manage.

Global State: Vuex

Vuex is the official state management library for Vue. It provides a **centralized store** for all the components in an application.

- **Centralized**: A single object contains all your application-level state.
- **Predictable**: Changes to the state must happen in a predictable way (through mutations).
- **Traceable**: Devtools allow you to track every state change.

This solves the problem of prop drilling and complex event chains. Any component can access the store's state or trigger actions to change it.