✅ Interpolation (Mustache Syntax)

Name: `{{ }}` is called **Mustache Syntax**.
It is used to **insert dynamic data** in HTML.

🧪 Example:

```
<template>
  <h1>{{ greet }}</h1>
</template>

<script setup>
const greet = "Hello, Vue!";
</script>
```

📝 **Explanation**: `{{ greet }}` displays the value of the `greet` variable.

🔷 What is `ref()`?

`ref()` is used to create a **reactive primitive value** (like numbers, strings, booleans). It wraps the value in a special object and gives it a `.value` property that Vue tracks for changes.

✅ Example: Using `ref()` for a counter

```
<template>
  <h1>This is counter {{ count }}</h1>
  <button @click="Plus">Plus</button>
  <button @click="Minus">Minus</button>
</template>

<script setup>
import { ref } from "vue"; // Importing ref from Vue

const count = ref(0); // count is a reactive reference (starts with 0)

function Plus() {
  count.value = count.value + 1; // Access or update ref values using .value
}

function Minus() {
  count.value = count.value - 1;
}
</script>
```

🧠 Explanation:

- `ref(0)` makes `count` reactive.
- You **must use `.value`** to access or update it in the `<script>`.
- In the `<template>`, you can use `count` **without `.value`** – Vue automatically unwraps it for you.

🔷 What is `reactive()`?

`reactive()` is used to create a **reactive object** (like a JavaScript object with multiple properties). You **don't need `.value`** for each property.

✅ Example: Using `reactive()` for multiple numbers

```
<template>
  <h1>This is Num1 = {{ count.num1 }}</h1>
  <h1>This is Num2 = {{ count.num2 }}</h1>
  <button @click="Plus">Plus</button>
  <button @click="Minus">Minus</button>
</template>

<script setup>
import { reactive } from "vue"; // Importing reactive from Vue

const count = reactive({
  num1: 0,
  num2: 0
}); // count is a reactive object

function Plus() {
  count.num1 = count.num1 + 1;
  count.num2 = count.num2 - 1;
}

function Minus() {
  count.num1 = count.num1 - 1;
  count.num2 = count.num2 + 1;
}
</script>
```

🧠 Explanation:

- `reactive({})` makes the whole object reactive.
- You can update `count.num1`, `count.num2` directly.
- No `.value` is needed.
- Best for **grouping related values together**.

🔍 Key Differences Between `ref()` and `reactive()`

Feature	`ref()`	`reactive()`
Used For	Primitives (number, string, etc.)	Objects and arrays
Access in script	`.value` required	Direct access (e.g., `obj.key`)
Access in template	No `.value` needed	No `.value` needed
Nesting	Good for single values	Good for multiple values
Example	`const count = ref(0)`	`const obj = reactive({ a: 1 })`

💡 When to Use What?

- Use `ref()` when you're working with a **single value** like a number, boolean, or string.
- Use `reactive()` when you're dealing with an **object with multiple properties**.

✅ v-html (Raw HTML Output)

Name: `v-html` is a directive to **render raw HTML**.

🧪 Example:

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

<script setup>
const rawContent = "<strong>This is bold</strong>";
</script>
```

📝 **Warning**: Be careful with `v-html` to avoid **XSS (security)** issues.

✅ v-bind (Full Syntax)

Name: `v-bind` binds an **attribute** to a variable.

🧪 Example:

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

<script setup>
const imgUrl = "https://via.placeholder.com/150";
</script>
```

✅ : (Shorthand for v-bind)

🧪 Example:

```
<template>
  <img :src="imgUrl" />
</template>

<script setup>
const imgUrl = "https://via.placeholder.com/150";
</script>
```

📝 **Tip**: `:src="imgUrl"` is the same as `v-bind:src="imgUrl"`.

Style Binding

✅ . : (*Style Binding)

🧪 Example with `v-bind`:

```
<template>
  <p v-bind:style="{ color: textColor }">Styled Text</p>
</template>

<script setup>
const textColor = "blue";
</script>
```

🧪 Example with `:style`:

```
<template>
  <p :style="{ color: textColor }">Styled Text</p>
</template>
```

✅ v-model (Full Syntax)

Name: `v-model` is used for **two-way binding** between data and input fields.

🧪 Example:

```
<template>
  <input v-model="username" />
  <p>Hello, {{ username }}</p>
</template>

<script setup>
import { ref } from 'vue';
const username = ref('');
</script>
```

📝 **Explanation**: What the user types is automatically saved in `username`.

🧪 Example: Form Handling

```
<template>
    <form @submit.prevent="onFormSubmit">
      <label>User Name {{MyFormData.user}}</label><br/>
      <input v-model="MyFormData.user" type="text" placeholder="User Name">
<br/>

      <label>Password {{MyFormData.pass}}</label><br/>
      <input v-model="MyFormData.pass" type="text" placeholder="Password">
<br/>

      <button type="submit">Submit</button> <br/>
    </form>
</template>
<script setup>

  import {reactive} from "vue";

  const MyFormData=reactive({
    user:"",
    pass:""
  })

  function onFormSubmit(){
    console.log(MyFormData)
  }


</script>
```

✅ v-for (List Rendering)

Name: `v-for` is used to **loop through arrays or objects**.

🧪 Example:

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

<script setup>
```

```
const fruits = ["Mango", "Apple", "Orange"];
</script>
```

✅ v-if / v-else / v-else-if

Name: `v-if` is used for **conditional rendering** (hide/show based on conditions).

🧪 Example:

```
<template>
  <p v-if="loggedIn">Welcome!</p>
  <p v-else>Please log in.</p>
</template>

<script setup>
const loggedIn = false;
</script>
```

🧪 Example: v-else-if: Adds Additional Conditional Renderings

```
<template>
  <h1 v-if="marks<=100 && marks>=80">A+</h1>
  <h1 v-else-if="marks<80 && marks>=70">A</h1>
  <h1 v-else-if="marks<70 && marks>=60">A</h1>
  <h1 v-else>Please Login</h1>
</template>

<script setup>
  const marks =75
</script>
```

✅ v-show (vs v-if)

Name: `v-show` is also for conditionals but only hides the element using CSS.

🧪 Example:

```
<template>
  <p v-show="showText">This text is conditionally visible.</p>
</template>
```

```
<script setup>
const showText = true;
</script>
```

📝 `v-if` removes from DOM, `v-show` only hides it.

✅ Event Handling - @click

Name: `@click` is shorthand for `v-on:click`
Used to handle **user events** like clicking, typing, etc.

🧪 Example 1 (shorthand):

```
<template>
  <button @click="sayHi">Click Me</button>
</template>

<script setup>
function sayHi() {
  alert("Hi there!");
}
</script>
```

🧪 Example 2 (full syntax):

```
<template>
  <button v-on:click="sayHi">Click Me</button>
</template>
```

🧪 Example 3 (Others events):

```

<template>
  <h1>Events</h1>
  <button @click="onClick">Click Event</button>

  <form @submit.prevent="onFormSubmit">
    <button type="submit">Form Submit Event</button>
  </form>

  <button @click.right="onRightClick">Right Click Event</button>
  <button @click.left="onLeftClick">Left Click Event</button>

  <input @change="onChange" placeholder="on change event">
  <input @keydown="onChange" placeholder="on key down event">
  <input @keyup="onChange" placeholder="on key up event">

  <input @focus="onChange" placeholder="on foucs event">

  <input @focusin="onChange" placeholder="on foucs in event">
  <input @focusout="onChange" placeholder="on foucs out event">







</template>


<script setup>

function onClick(){
  alert("I am click event")
}

function onRightClick(){
  alert("I am right click event")
}

function onLeftClick(){
  alert("I am left click event")
}


function onChange(){
  alert("I am on Change Event")
}


function onFormSubmit(){
  alert("I am click event")
}

</script>
```

✅ Computed Properties

🧪 Example:

```
<template>
  <p>Full Name: {{ fullName }}</p>
</template>

<script setup>
import { ref, computed } from 'vue';

const first = ref("John");
const last = ref("Smith");

const fullName = computed(() => `${first.value} ${last.value}`);
</script>
```

✅ Class Binding

🧪 Example with `v-bind`:

```
<template>
  <p v-bind:class="{ active: isActive }">This is dynamic</p>
</template>

<script setup>
const isActive = true;
</script>
```

🧪 Example with `:class`:

```
<template>
  <p :class="{ active: isActive }">This is dynamic</p>
</template>
```