Introduction of Vue :

Vue is a progressive JavaScript framework for building user interfaces (UIs) and single-page applications (SPAs). It is designed to be incrementally adoptable and integrates smoothly with other libraries or existing projects. Vue builds on standard HTML, CSS, and JavaScript, offering a declarative and component-based programming model to help developers create applications efficiently—whether simple or complex.

```
JavaScript :
import { createApp, ref } from 'vue'

createApp({
  setup() {
    return {
      count: ref(0)
    }
  }
}).mount('#app')

template :
<div id="app">
  <button @click="count++">
    Count is: {{ count }}
  </button>
</div>
```

Core Features Demonstrated

1. Declarative Rendering

- Vue extends standard HTML with a template syntax.
- This allows you to declaratively define what the HTML output should look like.
- The HTML is dynamically rendered based on the JavaScript state (data).
- Example: If a variable message = "Hello" changes to message = "Hi", the UI updates automatically.

2. Reactivity

- Vue provides a reactive system that tracks changes in your JavaScript data.
- When the data/state changes, Vue automatically updates the DOM to reflect those changes.
- No need to manually manipulate the DOM—Vue handles it for you efficiently.

Single-File Components (SFCs)

Single-File Components (SFCs) are .vue files that contain a component's template, JavaScript logic, and CSS styles all in one file.

This structure makes Vue components self-contained, organized, and easy to maintain.

```
<template>
  <!-- HTML structure of the component -->
  <div>
    <h1>{{ message }}</h1>
    <button @click="changeMessage">Click Me</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: 'Hello from Vue!'
    };
  },
  methods: {
    changeMessage() {
      this.message = 'You clicked the button!';
    }
  }
};
</script>

<style scoped>
/* Styles that apply only to this component */
h1 {
  color: teal;
}
</style>
```

- `<template>`: Defines the HTML layout of the component.
- `<script>`: Contains the JavaScript logic (data, methods, lifecycle hooks).
- `<style>`: Adds CSS styles specific to this component. The scoped attribute ensures styles apply only within this component.

Types of API styles

There are two types : Options API and Composition API

1. Options API

define your component by using options like data, methods, computed, watch, etc. This api style is easy to understand.

```
<script>
export default {
  data() {
    return {
```

```
      count: 0
    };
  },
  methods: {
    increment() {
      this.count++;
    }
  },
  computed: {
    doubled() {
      return this.count * 2;
    }
  }
};
</script>
```

2. Composition API

Vue's setup() function to compose logic using reactive primitives like ref, reactive, computed, watch, etc.

This API style used for larger and more complex components.

Composition API can be written in two different styles :

i. Using `<script setup>`

A compiler-enhanced syntax that makes Composition API more concise and readable.No need for export default or return from setup() — everything is automatically available in the template.

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

const count = ref(0);
const increment = () => {
  count.value++;
};
const doubled = computed(() => count.value * 2);
</script>

<template>
  <div>
    <p>Count: {{ count }}</p>
    <p>Doubled: {{ doubled }}</p>
    <button @click="increment">Increase</button>
  </div>
</template>
```

ii. Using setup() function

Standard way to define Composition API logic. Write a setup() function inside the export default block and return everything you want to expose to the template.

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

export default {
  setup() {
    const count = ref(0);
    const increment = () => {
      count.value++;
    };
    const doubled = computed(() => count.value * 2);

    return {
      count,
      increment,
      doubled
    };
  }
};
</script>

<template>
  <p>Count: {{ count }}</p>
  <p>Doubled: {{ doubled }}</p>
  <button @click="increment">+1</button>
</template>
```

Options API and Composition API

Feature	Options API	Composition API
Readability	Easier for beginners	More abstract, needs practice
Logic Organization	Scattered in options	Grouped by feature
TypeScript Support	Decent	Excellent
Code Reuse	Mixins (less ideal)	Composables (cleaner)
Flexibility	Less flexible	Highly flexible

Note :

- If you're just getting started, go with Options API.
- If you're building for the long run, learning and using the Composition API is totally worth it.