Vue.js : composants

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

Vue.js

Remarque

- Pour une meilleure restructuration du projet, utilisons les composants.
- Application **Vue.js** = { composants }

Vue.js

Déplaçons le contenu de la balise `<div id='app'>` **dans une nouvelle section** `template`

```
const App = Vue.createApp({
    data() {
        return {
            // les attributs précédents
        };
    },
    methods: {
            // les méthodes précédentes
    },
    template : `
        <!--
            déplacer ici le contenu de la balise <div id='app'> ici entre backquote
        -->
    `
});

App.mount('#app');
```

Vue.js

Déplaçons le contenu de la balise `<div id='app'>` **dans une nouvelle section** `template`

```
const App = Vue.createApp({
    data() {
        return {
            // les attributs précédents
        };
    },
    methods: {
            // les méthodes précédentes
    },
    template : `
        <!--
            déplacer ici le contenu de la balise <div id='app'> ici entre backquote
        -->
    `
});

App.mount('#app');
```

Remarque

Un composant **Vue.js** = Objet **JavaScript** (composant inline).

Vue.js

Nouveau contenu d'`index.html`

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Cours Vue.js</title>
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <div id='app'>
    </div>
    <script src="https://unpkg.com/vue@next"></script>
    <script src="script.js">
    </script>
</body>

</html>
```

Vue.js

Nouveau contenu d'`index.html`

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Cours Vue.js</title>
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <div id='app'>
    </div>
    <script src="https://unpkg.com/vue@next"></script>
    <script src="script.js">
    </script>
</body>

</html>
```

Relancez l'application et vérifiez que tout fonctionne comme avant.

Vue.js

Créons un fichier `HomeView.js` dans `components` pour les données concernant le composant et gardons le reste dans `script.js`

```
export default {
    data() {
        return {
            // les attributs précédents
        };
    },
    methods: {
            // les méthodes précédentes
    },
    template : `
        <!-- les balises précédentes  -->
    `
};
```

Vue.js

Dans `script.js` (le point d'entrée), importons puis déclarons le composant

```
import HomeView from "./components/HomeView.js";

const App = Vue.createApp({
    components: {
        'home-view': HomeView
    }
});

App.mount('#app');
```

Dans `index.html` nous pouvons désormais utiliser le composant

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale
      =1.0">
    <title>Cours Vue.js</title>
    <link rel="stylesheet" href="style.css">
</head>

<body>
    <div id='app'>
        <home-view></home-view>
    </div>
    <script src="https://unpkg.com/vue@next"></script>
    <script src="script.js" type="module">
    </script>
</body>

</html>
```

Vue.js

Remarque

Si le composant est directement attaché au **DOM** (dans une page **HTML**) , il est recommandé de l'écrire en **Kebab Case**.

Vue.js

Remarque

Si le composant est directement attaché au **DOM** (dans une page **HTML**) , il est recommandé de l'écrire en **Kebab Case**.

Relancez l'application et vérifiez que tout fonctionne comme avant.

Vue.js

Remarque

Dans ce cours, nous considérons le projet `cours-vue-cli`.

Vue.js

Remarque

Dans ce cours, nous considérons le projet `cours-vue-cli`.

Rappel : pour créer un projet Vue.js avec CLI

```
vue create cours-vue-cli
```

Vue.js

Arborescence d'un projet **Vue.js**

- `node_modules` : contenant les fichiers nécessaires de la librairie **Node.js** pour un projet **Vue.js**
- `src` : contenant les fichiers sources de l'application
- `package.json` : contenant l'ensemble de dépendance de l'application
- `public` : contenant le point d'entrée de l'application `index.html`
- `vue.config.js` : fichier chargeant les services de **Vue-CLI**

Vue.js

Que contient `src` ?

- `assets` : unique dossier accessible aux visiteurs et contenant logo, CSS, images, sons...
- `components` : contient un composant : `HelloWorld.vue`
- `main.js` : fichier référencé par `index.html` et permettant de charger l'application **Vue.js** dans l'élément ayant l'identifiant `app`
- `App.vue` : composant principal référencé par `main.js` et utilisant les deux composants `HelloWorld.vue` et `TheWelcome.vue`

Vue.js

Rappel

- Application **Vue.js** = { composants }
- Théoriquement : un composant = { code **HTML** + code **CSS** + code **JS** }

Vue.js

Composant **SFC** : **S**ingle **F**ile **C**omponent

- Amélioration des composants inline
- Composant défini dans un fichier `.vue`
- En pratique

```
<template>
    ...
</template>

<script>
    ...
</script>

<!-- Add scoped attribute to limit CSS to this component only -->
<style scoped>
    ...
</style>
```

Vue.js

Deux types de composant

- Prédéfinis : comme le composant racine `App` et les composants intégrés fournis par **Vue.js** tels que `<transition>`, `<component>`...
- Personnalisés : à définir par le développeur et dont le nom doit être composé de plusieurs mots (`HelloWorld` par exemple)

Vue.js

Deux types de composant

- Prédéfinis : comme le composant racine `App` et les composants intégrés fournis par **Vue.js** tels que `<transition>`, `<component>`...
- Personnalisés : à définir par le développeur et dont le nom doit être composé de plusieurs mots (`HelloWorld` par exemple)

Remarque

Pour utiliser un composant, il faut l'importer.

Vue.js

Deux types d'importation

- Locale : le composant ne pourra être utilisé que dans le composant qui l'importe.
- Globale : le composant peut être utilisé dans toute l'application

Vue.js

Deux types d'importation

- Locale : le composant ne pourra être utilisé que dans le composant qui l'importe.
- Globale : le composant peut être utilisé dans toute l'application

Terminologie

- **Composant parent** : composant qui importe un autre composant.
- **Composant enfant** : composant importé par un autre composant.

Vue.js

Dans `App.vue`, gardons l'utilisation de `<HelloWorld/>` dans `template` et supprimons (ou commentons) son importation et déclaration dans `script`

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld/>
</template>

<script>
export default {
  name: 'App',
  components: {
//  HelloWorld
  }
}
</script>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

Vue.js

Importons et déclarons globalement le composant `HelloWorld` dans `main.js`

```
import { createApp } from 'vue'
import App from './App.vue'
import HelloWorld from './components/HelloWorld.vue'

createApp(App)
.component('HelloWorld', HelloWorld)
.mount('#app')
```

Vue.js

Importons et déclarons globalement le composant `HelloWorld` dans `main.js`

```
import { createApp } from 'vue'
import App from './App.vue'
import HelloWorld from './components/HelloWorld.vue'

createApp(App)
.component('HelloWorld', HelloWorld)
.mount('#app')
```


Remarques

- Le composant `HelloWorld` peut être utilisé par tous les composants de l'application sans l'importer localement.
- Possibilité d'importer globalement plusieurs composants.

Vue.js

Considérons aussi le code simplifié suivant pour `HelloWorld.vue`

```
<template>
  <div class="hello">
    Hello world
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
}
</script>

<style scoped>
</style>
```

Vue.js

Considérons aussi le code simplifié suivant pour `HelloWorld.vue`

```
<template>
  <div class="hello">
    Hello world
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
}
</script>

<style scoped>
</style>
```


Explication

- La partie `template` contient le code **HTML** à afficher.
- La partie `script` exporte le composant pour pouvoir l'utiliser ailleurs : l'attribut `name` permet de définir le nom à utiliser pour l'importation.

Vue.js

Relancer l'application et vérifier que `Hello world` s'affiche au démarrage de l'application.

Vue.js

Remettons dans `main.js` le code initial (en supprimant l'importation et la déclaration de `HelloWorld`)

```
import { createApp } from 'vue'
import App from './App.vue'

createApp(App).mount('#app')
```

Vue.js

Dans `App.vue`, importons localement `HelloWorld` et déclarons le dans `components`

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld/>
</template>

<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    HelloWorld
  }
}
</script>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

Vue.js

Relancer l'application et vérifier que `Hello world` s'affiche au démarrage de l'application.

Vue.js

Rappelons les 2 modes de liaison template/script d'un même composant

- One way binding
  - Interpolation {{ ... }}
  - Attribute binding `v-bind`
  - Event binding `v-on`
- Two way binding : `v-model`

Vue.js

Déclarons la fonction `data` dans la partie `script` du composant `HelloWorld`

```
export default {
  name: 'HelloWorld',
  data() {
    return {
      msg: "Hello world"
    }
  }
}
```

Vue.js

Déclarons la fonction `data` dans la partie `script` du composant `HelloWorld`

```
export default {
  name: 'HelloWorld',
  data() {
    return {
      msg: "Hello world"
    }
  }
}
```


Utilisons l'interpolation pour afficher le contenu de `msg` dans la partie `template` du même composant

```
<div class="hello">
    {{ msg }}
</div>
```

Vue.js

Relancer l'application et vérifier que `Hello world` s'affiche au démarrage de l'application.

Vue.js

Formes d'interaction entre composants parent-enfant

- Ajouter le sélecteur d'un premier composant dans le template d'un deuxième composant
  - on appelle le premier composant : composant fils
  - on appelle le deuxième composant : composant parent
- Plusieurs formes de transmission de données par
  - Du parent vers l'enfant : via la balise `slot` (transclusion) ou la propriété `props`
  - De l'enfant vers le parent : via la méthode `$emit()`

Vue.js

Vue.js nous permet d'utiliser une balise auto-fermante ou une balise fermante pour nos composants (`template` de `App.vue`)

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld/>
  <HelloWorld></HelloWorld>
</template>
```

Vue.js

Vue.js nous permet d'utiliser une balise auto-fermante ou une balise fermante pour nos composants (`template` de `App.vue`)

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld/>
  <HelloWorld></HelloWorld>
</template>
```


Relancer l'application et vérifier que `Hello world` s'affiche deux fois au démarrage de l'application.

Vue.js

Et si le composant parent (ici `App`) envoie des données entre les balises ouvrante et fermante de `HelloWorld`

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld/>
  <HelloWorld>Wick</HelloWorld>
</template>
```

Vue.js

Et si le composant parent (ici `App`) envoie des données entre les balises ouvrante et fermante de `HelloWorld`

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld/>
  <HelloWorld>Wick</HelloWorld>
</template>
```

Constat

Le contenu ne s'affiche pas.

Vue.js

Et si le composant parent (ici `App`) envoie des données entre les balises ouvrante et fermante de `HelloWorld`

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld/>
  <HelloWorld>Wick</HelloWorld>
</template>
```


Constat

Le contenu ne s'affiche pas.

Question

Comment récupérer et afficher les données envoyées par le composant parent ?

Vue.js

Pour récupérer des données envoyées par le composant parent entre les balises ouvrante et fermante, on utilise la balise `slot` (contenu du `template` du composant `HelloWorld`)

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot/>
  </div>
</template>
```

Vue.js

Pour récupérer des données envoyées par le composant parent entre les balises ouvrante et fermante, on utilise la balise `slot` (contenu du `template` du composant `HelloWorld`)

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot/>
  </div>
</template>
```


Question

Comment définir un contenu par défaut pour le cas où le parent n'envoie pas de données ?

Vue.js

Pour définir un contenu par défaut pour la balise `slot` (contenu du `template` du composant `HelloWorld`)

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot>Doe</slot>
  </div>
</template>
```

Vue.js

Pour définir un contenu par défaut pour la balise `slot` (contenu du `template` du composant `HelloWorld`)

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot>Doe</slot>
  </div>
</template>
```


Relancer l'application et vérifier que `Hello Doe` et `Hello Wick` s'affichent au démarrage de l'application.

Vue.js

Question

Comment envoyer plusieurs données du parent vers l'enfant entre les balises ouvrante et fermante ?

Vue.js

Question

Comment envoyer plusieurs données du parent vers l'enfant entre les balises ouvrante et fermante ?

Pour cela, il faut les séparer dans des templates différents et utiliser `v-slot` (contenu du `template` du composant `App`)

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld/>
  <HelloWorld>
    <template v-slot:nom>Wick</template>
    <template v-slot:age>45</template>
  </HelloWorld>
</template>
```

Vue.js

Pour récupérer les données envoyées par le parent, on utilise la balise `slot` avec l'attribut `name` (contenu du `template` du composant `HelloWorld`)

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot name="nom">Doe</slot>,
    you have <slot name="age">0</slot> years old.
  </div>
</template>
```

Vue.js

L'écriture précédente peut être simplifiée avec les identifiants (commençant par #)

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld/>
  <HelloWorld>
    <template #nom>Wick</template>
    <template #age>45</template>
  </HelloWorld>
</template>
```

Vue.js

Rien à changer dans `template` **de** `HelloWorld`

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot name="nom">Doe</slot>,
    you have <slot name="age">0</slot> years old.
  </div>
</template>
```

Vue.js

On peut aussi utiliser un identifiant `default` avant de simplifier la récupération

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld/>
  <HelloWorld>
    <template #default>Wick</template>
    <template #age>45</template>
  </HelloWorld>
</template>
```

Vue.js

Le `default` sera récupéré sans avoir besoin d'utiliser `name` dans `slot`

```
<template>
  <div class="hello">
    {{ msg }}
  </div>
  <div>
    Hello <slot>Doe</slot>,
    you have <slot name="age">0</slot> years old.
  </div>
</template>
```

Exercice : **primeur-produit**

- Première partie
  - Créez deux composants `PrimeurComponent` et `ProduitComponent` : `PrimeurComponent` est le composant parent des composants `ProduitComponent`
  - Le composant `PrimeurComponent` a un attribut `produits` : à déclarer dans la fonction `data` (voir ci-dessous).
  - Utilisez `v-for` pour créer autant de composants `ProduitComponent` que d'éléments dans le tableau `produits` : chaque composant `produit` reçoit le nom qu'il doit afficher.
- Deuxième partie
  - Dans `PrimeurComponent`, ajoutez un bouton `ajouter` et trois zones de saisie : une pour le nom, une pour le prix et une pour la quantité.
  - En cliquant sur le bouton `ajouter`, un nouveau composant `produit` s'ajoute (s'affiche) au composant (dans la page) avec le nom saisi par l'utilisateur.

Attribut à déclarer dans la fonction `data` dans `primeur.vue`

```
produits: [
    { nom: "banane", prix: 3, quantite: 10 },
    { nom: "fraise", prix: 10, quantite: 20 },
    { nom: "poivron", prix: 5, quantite: 10 }
]
```

Vue.js

Récapitulons

Grâce à la balise `<slot/>`, on récupère une valeur envoyée par le parent entre la balise ouvrante et fermante de l'enfant.

Vue.js

Récapitulons

Grâce à la balise `<slot/>`, on récupère une valeur envoyée par le parent entre la balise ouvrante et fermante de l'enfant.

Question

Et si nous voulions passer des valeurs comme valeur d'attribut ?

Vue.js

Récapitulons

Grâce à la balise `<slot/>`, on récupère une valeur envoyée par le parent entre la balise ouvrante et fermante de l'enfant.

Question

Et si nous voulions passer des valeurs comme valeur d'attribut ?

Réponse

Nous pourrons utiliser la propriété `props`.

Vue.js

Ajoutons un attribut `ville` à la balise `HelloWorld` dans le `template` de `App.vue`

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld ville="Marseille"/>
  <HelloWorld>
    <template v-slot:nom>Wick</template>
    <template v-slot:age>45</template>
  </HelloWorld>
</template>
```

Vue.js

Pour récupérer la valeur de la propriété `ville`, il faut commencer par la déclarer dans une section `props` de `script` du composant `HelloWorld`

```
<script>
export default {
  name: 'HelloWorld',
  data() {
    return {
      msg: "Hello world"
    }
  },
  props: ['ville']
}
</script>
```

Vue.js

Pour afficher la valeur de la propriété `ville`, on utilise l'interpolation dans la partie `template`

```
<template>
  <div class="hello">
    {{ msg }} from {{ ville }}
  </div>
  <div>
    Hello <slot name="nom">Doe</slot>,
    you have <slot name="age">0</slot> years old.
  </div>
</template>
```

Vue.js

On peut aussi typer la `props`

```
<script>
export default {
  name: 'HelloWorld',
  data() {
    return {
      msg: "Hello world"
    }
  },
  props: {
    ville: String
  }
}
</script>
```

Vue.js

Les types autorisés sont les types prédéfinis en **JavaScript**

- Number
- String
- Boolean
- Array
- Date
- ...

Vue.js

On peut aussi spécifier une valeur par défaut pour la `props`

```
<script>
export default {
  name: 'HelloWorld',
  data() {
    return {
      msg: "Hello world"
    }
  },
  props: {
    ville: {
      type: String,
      default: 'Paris'
    }
  }
}
</script>
```

Vue.js

Exercice : **primeur-produit**

Reprenons l'exercice précédent et affichons le prix et la quantité disponible de chaque produit en utilisant `props`.

Vue.js

Objectif

Transmettre de données depuis un composant enfant vers un composant parent.

Vue.js

Dans le `template` de `HelloWorld`, ajoutons le contenu suivant

```
<div>
    <label for="nom">Nom</label>
    <input type="text" id="nom" v-model="nom">
    <button @click="$emit('sendData', this.nom)">
      Envoyer</button>
</div>
```

Vue.js

Dans le `template` de `HelloWorld`, ajoutons le contenu suivant

```
<div>
    <label for="nom">Nom</label>
    <input type="text" id="nom" v-model="nom">
    <button @click="$emit('sendData', this.nom)">
      Envoyer</button>
</div>
```


Sans oublier de déclarer `nom` comme attribut de la fonction `data`

```
nom: null
```

Vue.js

Explication

- En cliquant sur le bouton, le composant enfant (`HelloWorld`) émet un évènement à son parent (`App`).
- Par conséquent, l'évènement `sendData` (à définir dans la balise `HelloWorld` de `App` déclenchera l'exécution d'une fonction à définir dans `App`..
- Le nom sera envoyé comme paramètre.

Vue.js

Dans le `template` de `App`, ajoutons un évènement de type `sendData` (écrit en `kebab case`) dans `HelloWorld` qui, une fois déclenché, la fonction `afficherBonjour()` sera exécutée

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <p>Bonjour {{ nom ?? "Doe" }}</p>
  <HelloWorld @send-data="afficherBonjour" ville="Marseille"/>
  <!-- <HelloWorld  />
  <HelloWorld>
    <template v-slot:nom>Wick</template>
    <template v-slot:age>45</template>
  </HelloWorld> -->
  <PrimeurComponent />
</template>
```

Vue.js

N'oublions pas de définir `nom` dans `data` et `afficherBonjour` dans `methods`

```
<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    HelloWorld,
  },
  data() {
    return {
      nom: null
    }
  },
  methods: {
    afficherBonjour(nom) {
      this.nom = nom;
    }
  }
}
</script>
```

Vue.js

Dans le `template` de `HelloWorld`, nous pouvons aussi remplacer

```
<button @click="$emit('sendData', this.nom)">Envoyer</button>
```

Vue.js

Dans le `template` de `HelloWorld`, nous pouvons aussi remplacer

```
<button @click="$emit('sendData', this.nom)">Envoyer</button>
```

Par

```
<button @click="envoyer">Envoyer</button>
```

Vue.js

Dans le `template` de `HelloWorld`, nous pouvons aussi remplacer

```
<button @click="$emit('sendData', this.nom)">Envoyer</button>
```

Par

```
<button @click="envoyer">Envoyer</button>
```

Et utiliser `$emit` dans `envoyer` (à définir dans `script`)

```
methods: {
  envoyer() {
    this.$emit('sendData', this.nom)
  }
}
```

Vue.js

Exercice : **primeur-produit**

- Dans `PrimeurComponent`, déclarez un attribut `total` dans `data`.
- Dans `ProduitComponent`, ajoutez une zone de saisie et un bouton.
- En choisissant une quantité et appuyant sur le bouton, le total sera mis à jour et le bouton sera désactivé.

Vue.js

Exercice 2

- Considérons deux composants `ClasseComponent` et `EleveComponent`.
- Chaque composant `EleveComponent` aura un champ texte pour saisir une note et un bouton pour envoyer la valeur au composant `ClasseComponent`.
- Le bouton sera désactivé après envoi.
- Chaque fois que le composant `ClasseComponent` reçoit la note d'un `EleveComponent`, il recalcule la moyenne et il l'affiche.
- Il y aura autant de `EleveComponent` (composant enfant) dans `ClasseComponent` (composant parent) que d'éléments dans le tableau `noms` (voir ci-dessous).

Attribut `noms` à déclarer dans `data` de `ClasseComponent`

```
noms: ['Wick', 'Hoffman', 'Abruzzi']
```

Vue.js

Exercice **clavier-touche** (Simulation d'un clavier virtuel)

- Créez deux composants `ClavierComponent` et `ToucheComponent`.
- Le composant `ClavierComponent` a un attribut `lettres` (voir ci-dessous)
- le composant `ToucheComponent` a un attribut `value` recevant une valeur du tableau `lettres` (un composant fils `ToucheComponent` pour chaque valeur du tableau `lettres`).
- Chaque `ToucheComponent` affiche la lettre qu'il a reçue sur un bouton.
- En cliquant sur ce bouton, la lettre s'affiche (à la suite des autres) dans une balise `textarea` définie dans le composant `ClavierComponent`.

Contenu du tableau `lettres`

```
lettres: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', '
  m', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

Vue.js

Lifecycle hooks

- Tout composant `Vue.js` a un cycle de vie qui commence à la création.
- `Vue.js` nous a préparé une méthode pour chaque phase du cycle de vie : **Lifecycle hooks**.

Vue.js

Différentes méthodes (hooks) de cycle de vie d'un composant **Vue.js** (dans l'ordre)

- `setup` (**Vue.js 3**) : appelé avant la création du composant et après la résolution de toutes les `props` (à voir dans un prochain chapitre).
- `beforeCreate` : appelée avant la création du composant.
- `created` : appelée après la création du composant.
- `beforeMount` : appelée avant chaque que le composant ne soit attaché au **DOM**.
- `mounted` : appelée après attachement du composant et tous ses composants enfants au **DOM**.
- `beforeUpdate` : appelée avant une modification d'un élément du composant ou de ses enfants.
- `updated` : appelée après modification.
- `beforeUnmount` : appelée avant destruction du composant.
- `unmounted` : appelée après destruction du composant.

Vue.js

Ajoutons toutes les méthodes `hooks` dans la partie `script` de `HelloWorld.vue` (ne les placez pas dans `methods`

```
beforeCreate() {
    console.log('before create')
},
created() {
    console.log('created')
},

beforeMount() {
    console.log('before mount')
},
mounted() {
    console.log('mounted')
},

beforeUpdate() {
    console.log('before update')
},
updated() {
    console.log('updated')
},

beforeUnmount() {
    console.log('before unmount')
},
unmounted() {
    console.log('unmounted')
},
```

Vue.js

Allez à la route associée au composant `CycleVieComponent` et vérifier l'affichage des messages suivants

```
before create
created
before mount
mounted
```

Vue.js

Allez à la route associée au composant `CycleVieComponent` et vérifier l'affichage des messages suivants

```
before create
created
before mount
mounted
```


Saisissez une valeur dans la zone de saisie et vérifiez l'apparition des deux messages suivants

```
before update
updated
```

Vue.js

Question 1

Quelle est la différence entre `created` et `mounted` ?

Vue.js

Question 1

Quelle est la différence entre `created` et `mounted` ?

`created`

- S'exécute au tout début du cycle de vie,
- S'exécute une seule fois,
- Impossible de manipuler le **DOM**,
- Utilisé généralement pour récupérer des données à partir d'une API backend.

`mounted`

- S'exécute après `created`,
- Peut-être exécutée plusieurs fois,
- La manipulation de **DOM** est possible,
- N'est pas utilisée pour la récupération de données.

Vue.js

`ref`

- Attribut spécial pour les éléments **HTML** définis dans un composant.
- Permettant d'obtenir une référence directe sur un élément **HTML**.

Dans le `template` de `HelloWorld`, définissons une référence sur la zone de saisie

```
<template>
  <div class="hello">
    {{ msg }} from {{ ville }}
  </div>
  <div>
    Hello <slot name="nom">Doe</slot>,
    you have <slot name="age">0</slot> years old.
  </div>
  <div>
    <label for="nom">Nom</label>
    <input type="text" id="nom" v-model="nom" ref="name">
    <button @click="$emit('sendData', this.nom)">Envoyer</button>
  </div>
</template>
```

Dans le `template` de `HelloWorld`, définissons une référence sur la zone de saisie

```
<template>
  <div class="hello">
    {{ msg }} from {{ ville }}
  </div>
  <div>
    Hello <slot name="nom">Doe</slot>,
    you have <slot name="age">0</slot> years old.
  </div>
  <div>
    <label for="nom">Nom</label>
    <input type="text" id="nom" v-model="nom" ref="name">
    <button @click="$emit('sendData', this.nom)">Envoyer</button>
  </div>
</template>
```

Objectif

Placer le curseur et attacher un `placeholder` à la zone de saisie.

Vue.js

Ajoutons la méthode `mounted` pour placer le curseur au chargement du DOM

```
mounted() {
    this.$refs.name.placeholder = 'Votre nom';
    this.$refs.name.focus();
}
```

Vue.js

Ajoutons la méthode `mounted` pour placer le curseur au chargement du DOM

```
mounted() {
    this.$refs.name.placeholder = 'Votre nom';
    this.$refs.name.focus();
}
```


Constat

`$refs` est un objet contenant toutes références.

Vue.js

Ajoutons la méthode `mounted` pour placer le curseur au chargement du DOM

```
mounted() {
    this.$refs.name.placeholder = 'Votre nom';
    this.$refs.name.focus();
}
```


Constat

`$refs` est un objet contenant toutes références.

Remarque

La méthode `created` ne pourra pas être utilisé car le composant ne sera pas encore attaché au **DOM**.

Vue.js

Créons un composant `ReactiveValue` avec le code suivant

```
<template>
    <p>{{ valeur1 }} + {{ valeur2 }} = {{ resultat }}</p>
</template>

<script>
export default {
    name: 'ReactiveValue',
    data() {
        return {
            valeur1: 2,
            valeur2: 3,
            resultat: 0
        }
    },
    created() {
            this.resultat = this.valeur1 + this.valeur2
    },
    mounted() {
        setInterval(() => this.valeur2 += 10, 5000);
    },
}
</script>
```

Vue.js

Créons un composant `ReactiveValue` avec le code suivant

```
<template>
    <p>{{ valeur1 }} + {{ valeur2 }} = {{ resultat }}</p>
</template>

<script>
export default {
    name: 'ReactiveValue',
    data() {
        return {
            valeur1: 2,
            valeur2: 3,
            resultat: 0
        }
    },
    created() {
            this.resultat = this.valeur1 + this.valeur2
    },
    mounted() {
        setInterval(() => this.valeur2 += 10, 5000);
    },
}
</script>
```

Question

Que sera le résultat de ce code ?

Résultat attendu

- Au chargement de la page `2 + 3 = 5`
- 5 secondes plus tard, 3 sera remplacé par 13
- Contenu attendu : `2 + 13 = 15`

Résultat obtenu

- Au chargement de la page `2 + 3 = 5`
- 5 secondes plus tard, 3 est remplacé par 13
- Contenu affiché : `2 + 13 = 5`

Résultat attendu

- Au chargement de la page `2 + 3 = 5`
- 5 secondes plus tard, 3 sera remplacé par 13
- Contenu attendu : `2 + 13 = 15`

Résultat obtenu

- Au chargement de la page `2 + 3 = 5`
- 5 secondes plus tard, 3 est remplacé par 13
- Contenu affiché : `2 + 13 = 5`

Conclusion

Quand `valeur1` ou `valeur2` change, `resultat` ne se met pas à jour.

Résultat attendu

- Au chargement de la page `2 + 3 = 5`
- 5 secondes plus tard, 3 sera remplacé par 13
- Contenu attendu : `2 + 13 = 15`

Résultat obtenu

- Au chargement de la page `2 + 3 = 5`
- 5 secondes plus tard, 3 est remplacé par 13
- Contenu affiché : `2 + 13 = 5`

Conclusion

Quand `valeur1` ou `valeur2` change, `resultat` ne se met pas à jour.

Deux solutions

- Première solution : utiliser les watchers (`watchEffect`)
- Deuxième solution : utiliser `computed`

Première solution : utiliser `watchEffect` (aucun changement dans le `template`)

```
<script>
import { watchEffect } from 'vue'

export default {
    name: 'ReactiveValue',
    data() {
        return {
            valeur1: 2,
            valeur2: 3,
            resultat: 0
        }
    },
    created() {
        watchEffect(() => {
            this.resultat = this.valeur1 + this.valeur2
        })
    },
    mounted() {
        setInterval(() => this.valeur2 += 10, 5000);
    },
}
</script>
```

Première solution : utiliser `watchEffect` (aucun changement dans le `template`)

```
<script>
import { watchEffect } from 'vue'

export default {
    name: 'ReactiveValue',
    data() {
        return {
            valeur1: 2,
            valeur2: 3,
            resultat: 0
        }
    },
    created() {
        watchEffect(() => {
            this.resultat = this.valeur1 + this.valeur2
        })
    },
    mounted() {
        setInterval(() => this.valeur2 += 10, 5000);
    },
}
</script>
```

Remarque

On utilise `watchEffect` pour relancer le calcul après chaque modification constatée.

Deuxième solution : utiliser `computed` (aucun changement dans le `template`)

```
<script>
import { computed } from 'vue'

export default {
    name: 'ReactiveValue',
    data() {
        return {
            valeur1: 2,
            valeur2: 3,
            resultat: 0
        }
    },
    created() {
        this.resultat = computed(() => this.valeur1 + this.valeur2)
    },
    mounted() {
        setInterval(() => this.valeur2 += 10, 5000);
    },
}
</script>
```

Deuxième solution : utiliser `computed` (aucun changement dans le `template`)

```
<script>
import { computed } from 'vue'

export default {
    name: 'ReactiveValue',
    data() {
        return {
            valeur1: 2,
            valeur2: 3,
            resultat: 0
        }
    },
    created() {
        this.resultat = computed(() => this.valeur1 + this.valeur2)
    },
    mounted() {
        setInterval(() => this.valeur2 += 10, 5000);
    },
}
</script>
```

Remarque

On utilise `computed` pour relancer le calcul après chaque modification constatée.

On peut aussi utiliser `computed` ainsi : sans déclarer `resultat` dans `data`

```
<template>
    <p>{{ valeur1 }} + {{ valeur2 }} = {{ resultat }}</p>
</template>

<script>
export default {
    name: 'ReactiveValue',
    data() {
        return {
            valeur1: 2,
            valeur2: 3,
        }
    },
    computed: {
        resultat() {
            return this.valeur1 + this.valeur2
        }
    },
    mounted() {
        setInterval(() => this.valeur2 += 10, 5000);
    },
}
</script>
```

Vue.js

	watch	watchEffect	computed
Observe une seule valeur	✓		
A accès à la valeur précédente	✓		
S'exécute immédiatement		✓	✓
S'exécute après changement	✓	✓	✓
Ne prend pas de paramètre		✓	✓