

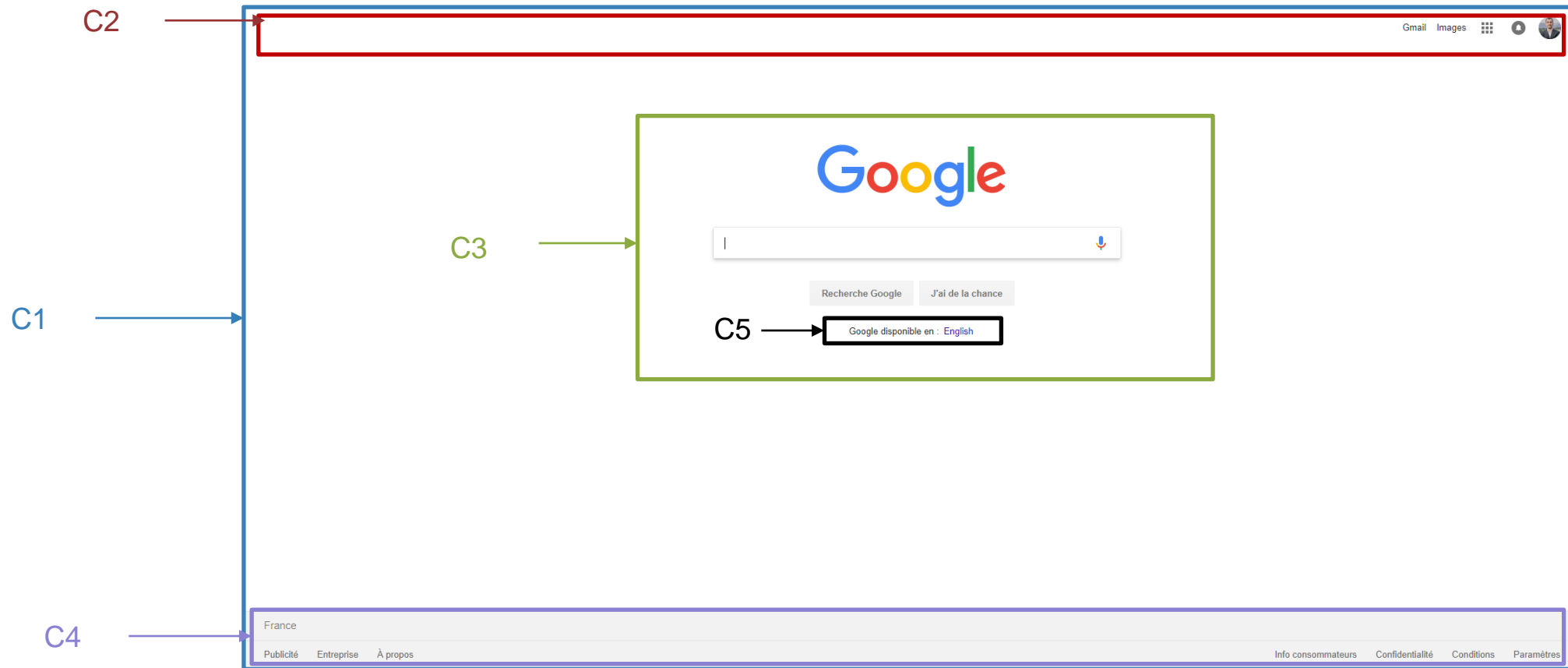
Les composants

Animé par Mazen Gharbi

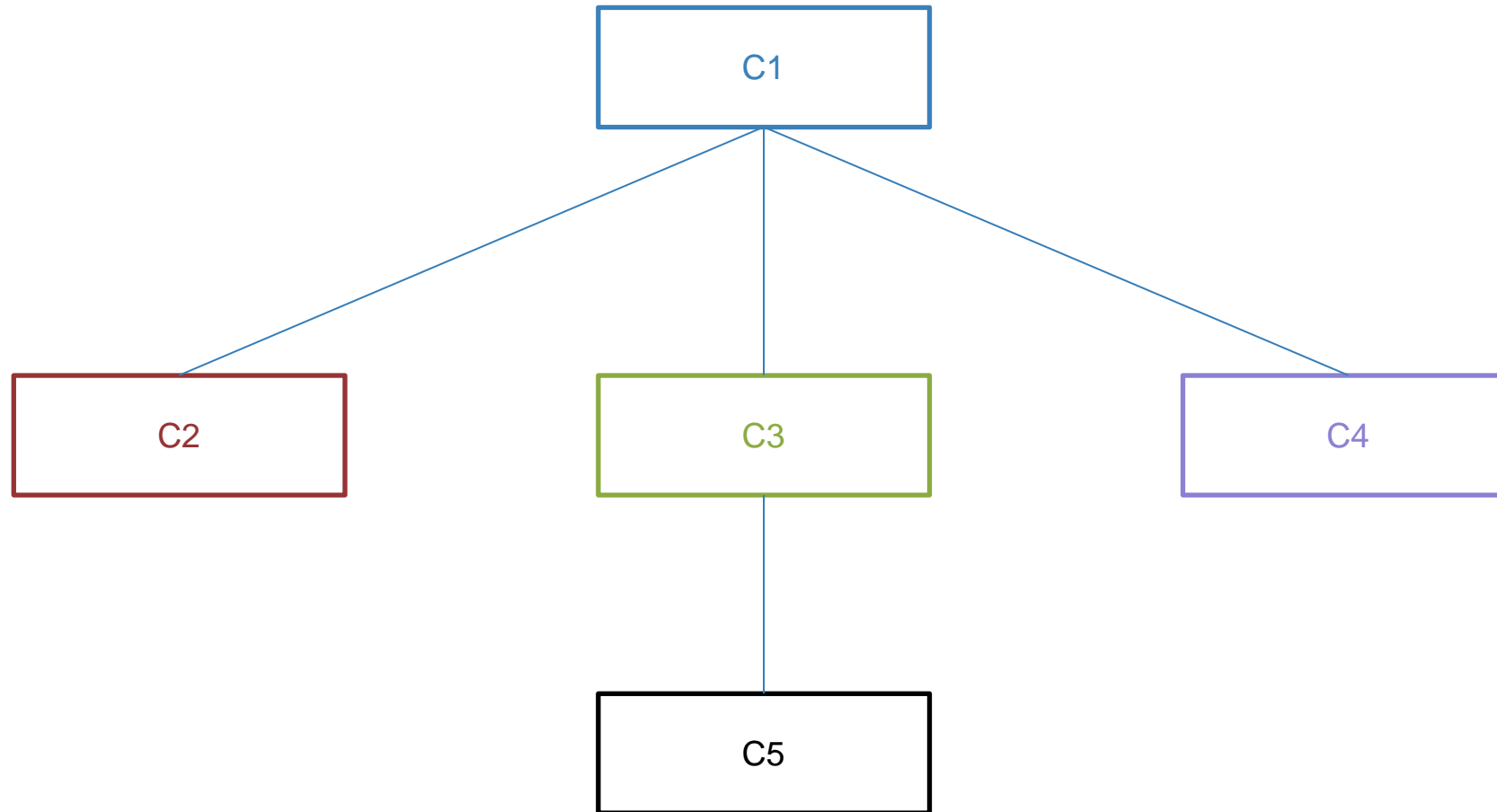
Qu'est ce qu'un composant ?

- ▷ Gère une vue ou une partie d'une vue ;
- ▷ Imaginez votre application comme une arborescence de composants ;
- ▷ Les composants permettent une meilleure décomposition de l'application, facilitent le refactoring et le testing ;
- ▷ Chaque composant est une nouvelle instance de Vue et est, de ce fait, isolé des autres par défaut.

Séparation par composants



Séparation par composants



Notre premier composant

- ▷ Créons un composant « mcdm-menu » !
 - › *dash-case pour le nom de fichier, PascalCase pour le nom du composant*
 - › *Il est recommandé d'appliquer un préfixe aux noms de vos composants*
 - › *les fichiers composants peuvent avoir l'extension .vue ou .js au choix*
- ▷ Une façon d'enregistrer un composant consiste à utiliser la méthode « component » de Vue

Notre premier composant

► Une fois le composant créé, il est nécessaire de l'appeler sous forme de web-component (avec son identifiant) dans l'élément père

```
APP.component('mcdm-menu', {  
  template: `  
    <div>  
      Bonjour, veuillez trouver le menu du jour ci-dessous  
      <hr/>  
    </div>`  
});
```

Bonjour, veuillez trouver le menu du jour ci-dessous

Un composant doit avoir un seul élément racine !

index.js

```
<div id="app">  
  <mcdm-menu></mcdm-menu>  
</div>
```

index.html

Réutilisabilité

- ▷ L'avantage, c'est qu'on va pouvoir appeler ce bout de code où on le souhaite et autant de fois qu'on le veut

```
<div id="app">
  <mcdm-menu></mcdm-menu>
  <mcdm-menu></mcdm-menu>
  <mcdm-menu></mcdm-menu>
  <mcdm-menu></mcdm-menu>
</div>
```

index.html

Résultat :

Bonjour, veuillez trouver le menu du jour ci-dessous

Bonjour, veuillez trouver le menu du jour ci-dessous

Bonjour, veuillez trouver le menu du jour ci-dessous

Bonjour, veuillez trouver le menu du jour ci-dessous

Marquer le découpage entre vue & modèle

[Testez ce code !](#)

- ▷ On passe pour le moment une chaîne de caractère ce qui empêche la coloration syntaxique du navigateur
- ▷ **On préférera cette nouvelle écriture :**

```
<div id="app">  
  <mcdm-menu></mcdm-menu>  
</div>
```

Le moteur WEB ne comprend pas cette écriture et va ignorer cette balise.
On l'utilise donc simplement comme stockage

index.html

```
<script type="text/template" id="template-mcdm-menu">  
  <div>  
    Bonjour, veuillez trouver le menu du jour ci-dessous  
    <hr/>  
  </div>  
</script>
```

Bonjour, veuillez trouver le menu du jour ci-dessous
<hr/>

Variable renvoyé par createApp

```
↓  
APP.component('mcdm-menu', {  
  template: `#template-mcdm-menu`  
});
```

index.js

Passage de propriétés

- ▷ Nos composants sont flexibles !
- ▷ Nous avons la possibilité d'envoyer des propriétés aux composants enfants afin de les configurer
- ▷ Dans notre cas, nous tenterons d'envoyer le tableau des plats
- ▷ Dans la plupart des cas, **il est conseillé d'avoir un niveau de granularité important quant aux paramètres**
 - › Préférable au fait de passer un gros objet contenant toutes les infos
 - › Ainsi, on optimise la réutilisabilité

Passage de propriétés

- ▷ Le composant enfant prépare les paramètres à recevoir :

```
APP.component('mcdm-menu', {  
  props: ['menu'],  
  template: `#template-mcdm-menu`  
})
```

index.js

- ▷ Puis on passe le paramètre

- › Il est nécessaire d'ajouter « v-bind: » car si la variable se met à jour, on veut que l'enfant se mette à jour

```
<div id="app">  
  <mcdm-menu v-bind:menu="menuToday"></mcdm-menu>  
</div>
```


index.html

Passage de propriétés

- ▷ Sans le « v-bind: », VueJS considèrera la valeur passer comme une chaine de caractère et non comme une variable à interpréter
- ▷ **Attention**, si vous utilisez des noms en upperCase dans les props, la propriété devra être en dash-case lors du passage de paramètre !
 - › Les attributs HTML sont case insensitives ([lire plus](#))

```
APP.component('mcdm-menu', {  
  props: ['monMenu'],  
  template: `#template-mcdm-menu`  
});
```

`<mcdm-menu v-bind:mon-menu="menuToday"></mcdm-menu>`



Typage des propriétés

- ▷ Un code bien cadré est la condition sine qua non pour une application pérenne
- ▷ On a la possibilité de forcer le passage d'un paramètre et de forcer un type en particulier :
 - › Il est conseillé de toujours spécifier le type des propriétés

```
APP.component('mcdm-menu', {  
  props: {  
    menu: {  
      type: Array,  
      required: true  
    },  
    length: Number  
  },  
  template: `#template-mcdm-menu`  
});
```

index.js

Vider la carte des menus

- ▷ Nous allons ajouter un bouton « *Vider* » dans le composant enfant
- ▷ Lors du click, on voudra vider le contenu du tableau menu !

Bonjour, veuillez trouver le menu du jour ci-dessous

-
- [0] - Filet de saumon - 15€
 - [1] - Salade de fruit - 9€
 - [2] - Salade niçoise - 120€
-

Simple

▷ On va donc simplement récupérer la propriété du père et recréer un nouveau tableau :

```
...  
methods: {  
  clear() {  
    this.menu = [];  
  }  
},  
...
```

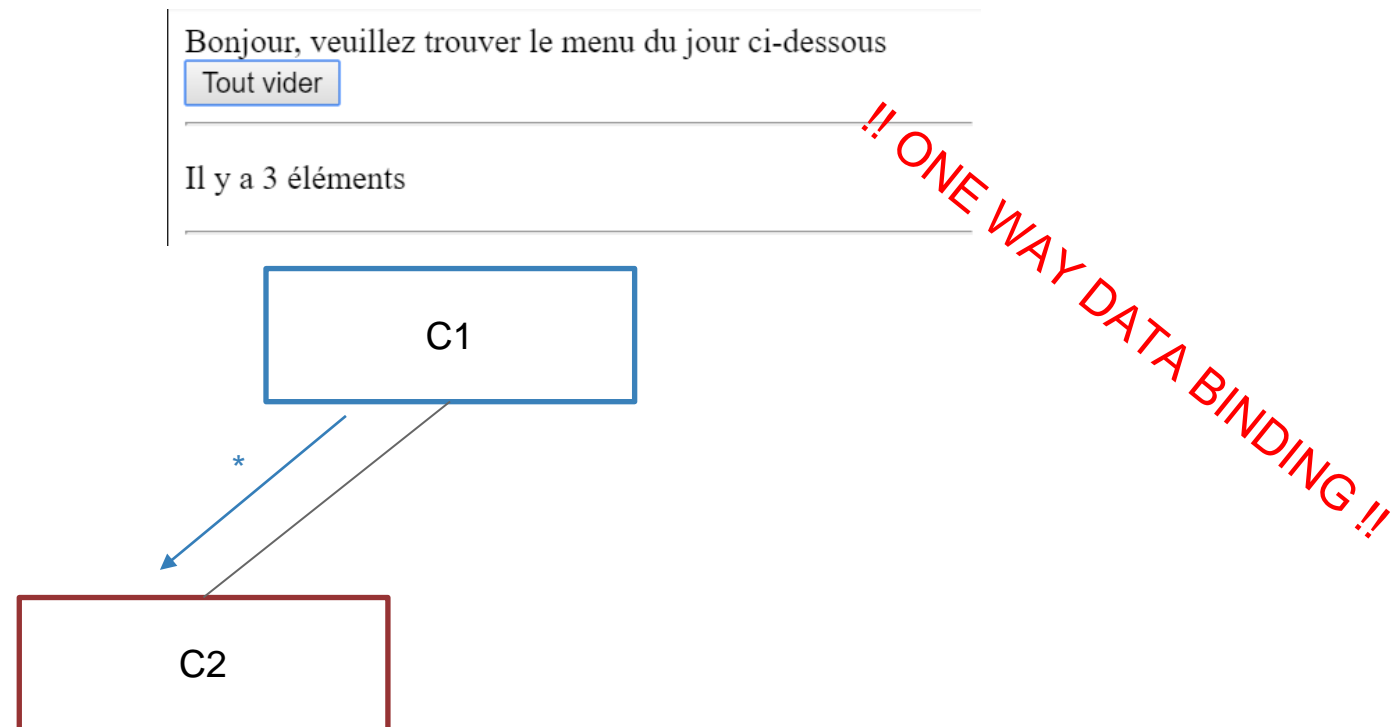
index.js

```
...  
<div>  
  Bonjour, veuillez trouver le menu du jour ci-dessous  
  <button @click="clear()">Tout vider</button>  
...
```

index.html

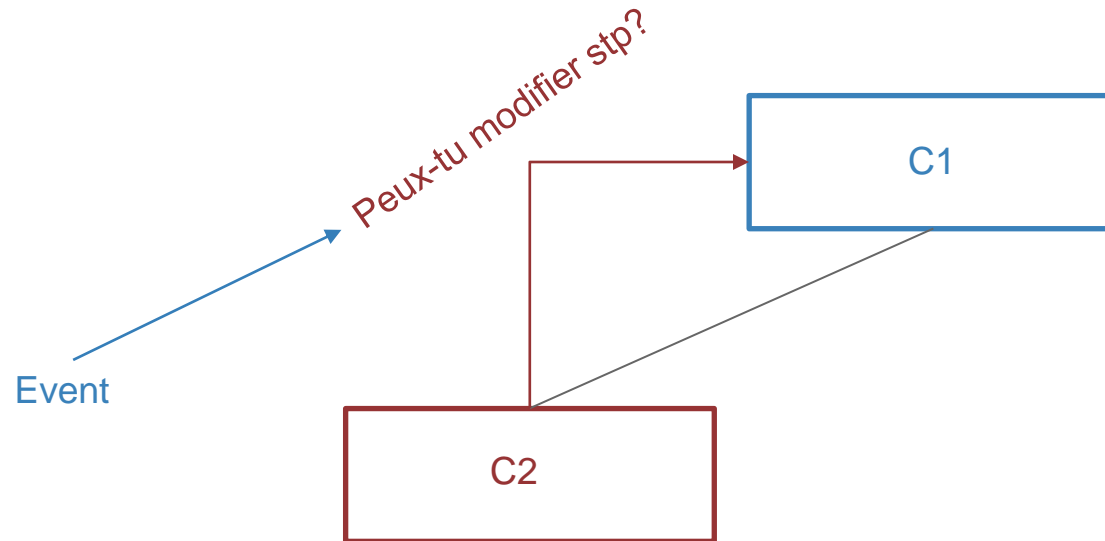
Problème

- ▷ Au click, on remarque que ~~le tableau a bien été supprimé mais que la taille affichée est toujours à 3 !~~
 - › Une erreur est levée !



Pas de two-way data binding !

- ▷ Les variables sont mises à jour du haut vers le bas, mais l'inverse est « impossible » !
- ▷ Il serait pratique d'avoir un cas de figure comme celui-ci :



Implémentation

- ▷ Notre problématique était de faire communiquer un enfant avec le père afin que le père modifie la variable
- › Le one-way data-binding fera en sorte que l'enfant soit notifié à la modif.

Modèle de l'enfant :

```
emitClear() {  
  this.$emit('clear');  
}
```

Vue de l'enfant :

```
<div>  
  Bonjour, veuillez trouver le menu du jour ci-dessous  
  <button @click="emitClear">Tout vider</button>  
</div>
```

Modèle du père :

```
onClear() {  
  this.menuToday = []  
}
```

Vue du parent :

```
<mcdm-menu  
  @clear="onClear">  
</mcdm-menu>
```

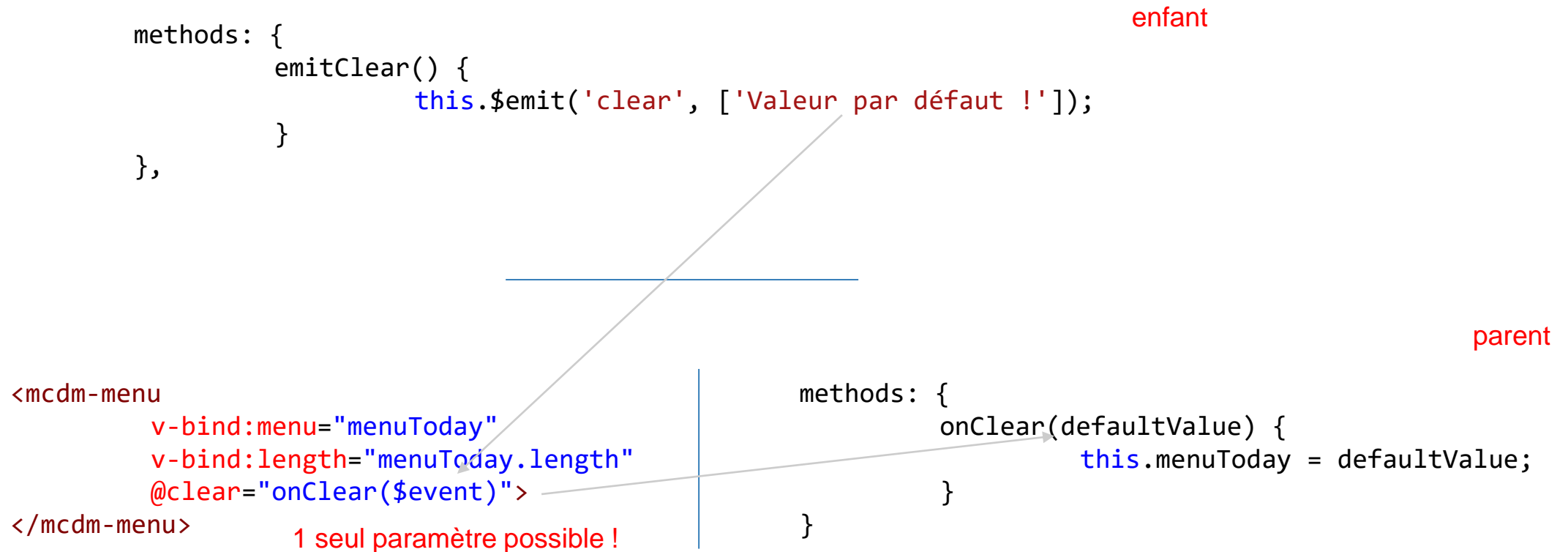
1

2

3

Passage de paramètres

- ▷ Il est évidemment possible de passer des paramètres



Simuler le two way data-binding

▷ Souvent, vous voudrez envoyer une propriété à un enfant puis la mettre à jour suite à un événement. Prenons ce cas de figure :

```
this.$emit('update:title', newTitle)
```

```
<text-document  
  v-bind:title="doc.title"  
  v-on:update:title="doc.title = $event"  
></text-document>
```

Ce qui revient au même que :

```
<text-document v-bind:title.sync="doc.title"></text-document>
```

Smart et Dumb components

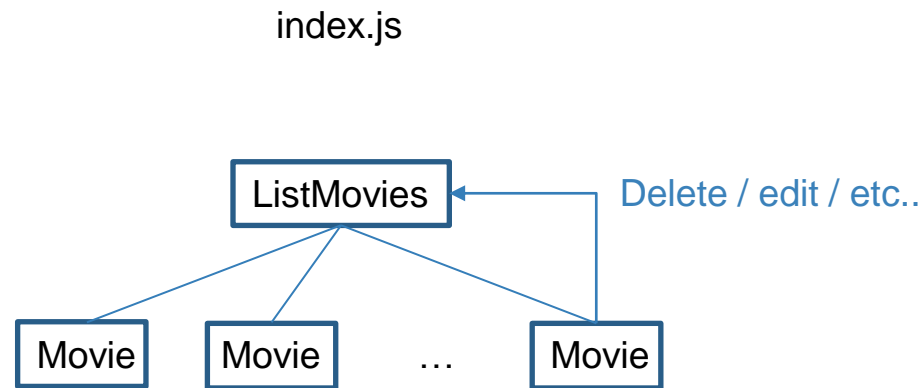
- ▷ Vous allez créer un nombre **conséquents** de composants ;
- ▷ Il va être nécessaire de catégoriser nos composants en 2 types : les **Smart** et les **Dumb** ;
- ▷ Les composants « Smart » sont des composants "High-Level" qui contrôlent la logique « **business** » ;
- ▷ Les composants « Dumb » ne contiennent pas de logique « business ». Ils se chargent principalement du design et doivent échanger les données avec les composants parents via les **événements**.
- ▷ **Préfixez ces composant avec un mot-clé (Base, V ou autre) pour les différencier**

TP orienté composant

- ▷ Nous allons factoriser notre TP n°1 pour l'adapter et intégrer la logique de composant
- ▷ Voici l'architecture que l'on souhaite avoir :

Attention à l'ordre !

```
<script src="./js/Movie.js"></script>  
<script src="./js/ListMovies.js"></script>  
<script src="./js/index.js"></script>
```



Les slots

- ▷ Pour appeler un composant, il suffit d'appeler son sélecteur entre chevrons « `<mon-composant></mon-composant>` »
- ▷ Que se passerait-il si l'on souhaitait ajouter du contenu entre les balises ?

```
<mon-composant>  
  <p>Mon contenu</p>  
</mon-composant>
```

- ▷ Rien.

Les slots

[Testez ce code !](#)

- ▷ En réalité, il est possible de récupérer le code passer entre les balises
- ▷ Pour ce faire, il faut appeler le composant « `<slot></slot>` » dans l'enfant :

```
<mcdm-panel>  
  <p>Ce texte doit être affiché chez l'enfant !</p>  
</mcdm-panel>
```

parent

```
<div>  
  <slot></slot>  
</div>
```

enfant

Résultat

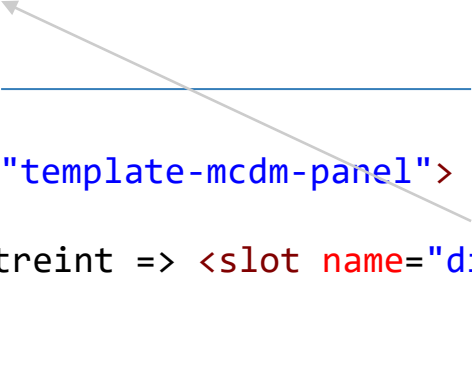
Ce texte doit être affiché chez l'enfant !

Sélecteurs

- Depuis la v2.6, nous avons la possibilité d'appliquer des sélecteurs aux slots

```
<mcdm-panel>                                     parent
  <template v-slot:display-2>Bonjour, ce texte ne s'affichera pas </template>
  <template v-slot:display>Mais celui-là, si !</template>
</mcdm-panel>
```

```
<script type="text/template" id="template-mcdm-panel">
  <div>
    Affichage restreint => <slot name="display"></slot>
  </div>
</script>                                     enfant
```



Résultat Affichage restreint => Mais celui-là, si !

Propriétés calculées

- ▷ Les « computed properties » sont des variables dont la valeur dépend d'autres facteurs ;
- ▷ « *Computed properties work like functions that you can use as properties* »
- ▷ Chaque fois qu'une dépendance d'une « *computed property* » change, la valeur de la propriété calculée est **réévaluée** ;
- ▷ Ces propriétés mettent en **cache** leurs résultats ;

Propriétés précalculées

```
<div id="app">  
  <input v-model.number="a" type="number" /><br />  
  a: {{ a }}, b : {{ b }}  
</div>
```

index.html

```
data() {  
  return {  
    a: 1  
  };  
,  
computed: {  
  b() {  
    return this.a + 1;  
  }  
}  
}
```

index.js

Objectif : Performances

- ▷ Continuons notre TP précédent et intégrons une **solution performante** pour calculer le nombre de films actuellement présents
- ▷ Reprenez le compteur de films et intégrez le avec les « **computed properties** »

3 film(s) au total

Ajouter un nouveau film



Les watchers

- ▷ Les propriétés calculées sont parfois limitées pour nos besoins ;
- ▷ Vue fournit une façon plus générique de réagir aux changements de données ;
- ▷ Un watcher observe une variable et réagit au moindre changement de celle-ci

Les watchers

[Testez ce code !](#)

index.js

```
data() {  
  return {  
    a: 1  
  };  
},  
watch: {  
  a(newValue, oldValue) {  
    console.log(`Changement de a : passage de ${oldValue} à ${newValue}`);  
  }  
}
```

Résultat

Changement de a : passage de 1 à 2

Changement de a : passage de 2 à 3

Réutilisabilité avec les mixins

- ▷ Vue **optimise la réutilisabilité** de notre code ;
- ▷ On l'a vu avec les composants ;
- ▷ Et si on pouvait appliquer ce principe aux méthodes / propriétés calculées / watchers / bref **à tout** !?
- ▷ Simple à mettre en place !
 - › Ressemble fortement à la mise en place des composants ;
- ▷ « *Un objet **mixin** peut contenir toute option valide pour un composant* »

Les mixins

► On va créer un fichier « mixins/watcher-a.js » :

```
export default {
  watch: {
    a(newValue, oldValue) {
      console.log(`a passe de ${oldValue} à ${newValue}`);
    }
  },
}
```

mixins/watcher-a.js

```
import watchersMixin from './mixins/watcher-a';
APP.component('mcdm-test', {
  mixins: [watchersMixin],
  data() {
    return {
      a: 0
    }
  }
})
...
```

index.js

Résultat

a passe de 0 à 1

a passe de 1 à 2

Surcharge de comportement

[Testez ce code !](#)

- ▷ Si un composant décide de redéfinir une méthode déjà définie par le mixins, notre entité surchargera le comportement de la fonction
- › **Ce n'est pas le cas pour les watchers.. Ceux-ci vont s'accumuler !**

```
export default {  
  methods: {  
    sayHello() {  
      console.log("Comportement initial");  
    }  
  },  
}
```

mixins/say-hello.js

```
APP.component('mcdm-test', {  
  mixins: [sayHelloMixin],  
  methods: {  
    sayHello() {  
      console.log("Surchargé ?");  
    }  
  },  
  created() {  
    this.sayHello();  
  },  
  ...  
});
```

index.js

Résultat

Surchargé ?

Cycle de vie d'un composant

- ▷ Les « Lifecycle Hooks » sont des fonction qui s'exécutent après des événements précis ;
 - › « Hooks de cycle de vie » en français
- ▷ Ces fonctions sont présentes par défaut dans les instances Vue ;
- ▷ Toutes peuvent être surchargées ;
- ▷ **Ne pas utiliser d'arrow functions avec les hooks !**
 - › Le this doit garder le binding avec l'appelant (Vue) et non le déclarant (composant)
- ▷ [En savoir plus](#)

Liste des hooks

Nom	Quand est-il appelé ?
beforeCreate	Après l'initialisation de l'instance, avant l'observation des données et la configuration des événements / observateurs
created	Une fois l'instance créée
beforeMount	Juste avant que le « montage » ne commence
mounted	Après que l'instance vient d'être montée sur le DOM
beforeUpdate	Lorsque les données changent, avant que le DOM virtuel soit restitué et corrigé
updated	Après une modification des données, lorsque le DOM virtuel est restitué et corrigé
activated	Lorsqu'un composant gardé en vie est activé
deactivated	Lorsqu'un composant gardé en vie est désactivé
beforeDestroy	Juste avant qu'une instance de Vue ne soit détruite
destroyed	Après qu'une instance de Vue ait été détruite

setup

▷ Permet d'alléger l'écriture

- › Améliore la réutilisabilité

▷ Prend 2 paramètres :

- › Les propriétés envoyées par le père (props) ;
- › Le contexte pouvant contenir les attributs / slots et les événements à émettre

```
export default {  
  setup(props, { attrs, slots, emit }) {  
    // Traitement ici  
  }  
}
```

Mis à jour à chaque changement



```
<script setup>  
  ...  
</script>
```

Avec Vite, ce raccourci d'écriture est possible !

Questions