



II. Communication entre composants

- Comprendre la communication parent \longleftrightarrow enfant
- Utiliser `input()` pour transmettre des données
- Utiliser `output()` pour émettre des événements
- Appliquer ces mécanismes dans notre projet Festival

1. Pourquoi communiquer entre composants ?

Un composant n'est jamais seul : il fait partie d'un système hiérarchique.

Il faut échanger des données :

- Parent → Enfant : le parent transmet des données (**input** ())
- Enfant → Parent : l'enfant émet des événements (**output** ())

Exemple :

- **StudentList** affiche plusieurs **StudentCard**.
- Chaque **StudentCard** (enfant) reçoit les infos d'un étudiant
- On pourrait avoir un bouton dans **StudentCard** pour demander au parent **StudentList** de supprimer cet étudiant

2. Communication parent \longrightarrow enfant (**input()**)

Avec Angular 20 : on utilise **input()** (fonction, pas décorateur) pour permettre à un parent de passer une donnée à un enfant.

Exemple

 composant enfant

student-card.ts

```
import { Component, input } from '@angular/core'
@Component({
  selector: 'student-card',
  templateUrl: './student-card.html'
})
export class StudentCardComponent {
  firstname = input<string>()
  name       = input<string>()
  program    = input<string>()
  programyear = input<string>()
  price      = input<number>()
  subscribed = input<Date>()
}
```

👉 composant parent

```
student-list.component.html  
  
<student-card  
  firstname = "Alice"    [name] = "'Demers'"  
  [program] = "'DaMS'"   [programyear] = "4"  
  [price]    = "628.0"    [subscribed] = "new Date()"   
</>
```

👉 Alice est transmis comme une chaîne brute (HTML) alors que pour `name` on évalue l'expression entre `''` qui est une chaîne de caractère aussi mais pourrait être le nom d'une variable (sans les `'`) et c'est alors la valeur de la variable qui serait transmise à `name`

3. Passer des variables à un composant

Avec des crochets [], Angular évalue une expression TypeScript et l'applique.

 **Exemple : passer une variable à un composant**

 composant parent

.ts

```
name = 'Alice'
```


.html

```
<student-card [firstname]="name"></student-card>
```

 la valeur de `name` est injectée dans la propriété `firstname` du composant enfant

.html

```
<student-card [firstname]=" 'name' "></festival-card>
```

 ici `name` n'est plus la propriété mais la chaîne de caractère `'name'`. C'est elle qui est injectée dans la propriété `firstname` du composant enfant

Exemple : Lier une propriété HTML native

 composant parent

```
.ts
isDisabled = true
.html
<button [disabled]="isDisabled"></button>
```

disabled est une propriété native HTML :

- si `isDisabled = true` → le bouton est grisé
- si `isDisabled = false` → le bouton est actif

 Les crochets `[]` servent à lier n'importe quelle propriété (Angular ou HTML) à une expression TypeScript.

!! Attention à bien faire la différence entre attribut et propriété :

```
<td [colspan]="{{1+1}}"> // ERREUR: colspan est un attribut
<td [colSpan]="{{1+1}}"> // OK: colSpan est la propriété
```

4. Exercice 3: transmettre des données

1. Créez un composant `StudentList`
 2. Dans `StudentList`, déclarez 2 ou 3 étudiants en propriétés TS (pas dans un tableau pour l'instant)
 3. Dans le template de `StudentList`, insérez plusieurs fois manuellement le composant enfant `StudentCard` pour chaque étudiant déclaré
- 🖥️ Résultat attendu : vous devez voir s'afficher les 2 étudiants
- 👉 Modifier une valeur de nom, et l'affichage doit changer

5. Communication enfant → parent (**output()**)

Avec Angular 20 : on utilise **output()** (fonction, pas décorateur) qui permet à un enfant d'émettre un évènement que pourra écouter le parent

 **Exemple avec un bouton "Supprimer"**

 composant enfant

```
.ts
import { Component, output } from '@angular/core'
@Component({
  selector: 'student-card',
  templateUrl: './student-card.html'
})
export class StudentCardComponent {
  ...
  remove = output<void>()
}
```


👉 composant parent

```
.html
```

```
<student-card (remove)="onDelete()"></student-card>
```

👉 L'enfant notifie le parent d'une action utilisateur qui `onDelete()` du parent

Remarque sur le typage :

`output<void>()` \Rightarrow l'événement ne transmet rien ;

`output<T>()` \Rightarrow l'événement transmet une valeur de type T ;
`emit()` transmet une valeur en paramètre qui est récupérée par un handler qui prend un paramètre du même type que le output.

Exemple avec un bouton "Supprimer" et un id :

composant enfant

```
.ts
@Component({
  selector: 'student-card',
  template: '<button (click)="remove.emit("id")">Supprimer</button>'
})
export class StudentCardComponent {
  remove = output<string>()
}
```

composant parent

```
.html
<student-card (remove)="onDelete($event)"></student-card>
```

 L'enfant notifie le parent d'une action utilisateur qui sera traitée par la fonction `onDelete()` du parent

Remarque : si le `output` est typé, le paramètre `$event` dans l'html du parent est obligatoire.

6. Exercice 4: suppression d'un étudiant

👉 **Objectif** : connecter `StudentCard` et `StudentList`.

1. Dans `StudentCard`, ajoutez un bouton “Supprimer” qui émet `remove`.
2. Dans `StudentList`, n'affichez qu'un seul étudiant et écoutez (`remove`) pour masquer la carte.

🔍 Exemple pour aider :

```
parent (html)
<student-card
  firstname    = "Alice"      [name]          = "'Demers'"
  [program]    = "'DaMS'"     [programyear] = "4"
  [price]      = "628"        [subscribed]  = "new Date()"
  (remove)     = "... "
  [hidden]     = "hideStudent">
</student-card>
```

7. Récapitulatif



Récapitulatif - Communication entre composants

- Données du parent vers l'enfant via `input()`
- Événements de l'enfant vers le parent via `output()`
- Import explicite des composants enfants (`standalone`)
- Différence de binding :
 - sans crochet `[]` : valeur brute (chaîne HTML).
 - avec crochet `[]` : expression TypeScript évaluée

8. Compétences acquises

Compétences acquises

- *Importer* un composant enfant dans un composant parent.
- *Utiliser* `input<T>()` pour transmettre des données typées du parent vers l'enfant.
- *Différencier* une valeur brute HTML d'une expression TS dans un binding.
- *Utiliser* `output<T>()` pour émettre un événement.
- *Implémenter* une interaction enfant \longrightarrow parent avec `.emit()`, avec ou sans valeur (`output<void>()` ou `output<string>()`).

9 Projet fil rouge : communication enfant \longleftrightarrow parent

Objectif : illustrer comment un composant enfant peut envoyer un événement au parent grâce à `output()` tout en mettant en œuvre une communication parent \longleftrightarrow enfant

1. Créez un composant `FestivalCard` qui affiche des 3 propriétés `name`, `location`, `year` et un bouton supprimer
 2. Créez un composant `FestivalList` qui :
 - a. crée 3 festivals en dur (`festival1`, `festival2`, `festival3`)
 - b. affiche 3 cartes `FestivalCard`, un pour chacun des festivals
 3. Faites en sorte que lorsqu'on clique sur "Supprimer" d'une carte, le parent masque la carte correspondante
- 💡 Astuce : utilisez des propriétés booléennes (`hide1`, `hide2`, `hide3`)

Résultat attendu :

1. au départ : trois cartes affichées ;
2. après clic sur “Supprimer” dans une carte, elle disparaît

Bonus pour les rapides 🚀

- Améliorez `FestivalList` pour gérer les suppressions avec un seul handler.
- Idée : l’enfant peut émettre une donnée utile (par ex. l’année du festival (un seul festival par an).
- Le parent utilise cette donnée pour déterminer quelle carte masquer.

Résultat attendu :

3. avec le bonus → même comportement mais un seul handler dans le parent qui enlève la bonne Card quelle que soit celle qui a été demandée à être supprimer.

Bonus pour les plus rapides 🚄

- Refactorisez `FestivalList` pour gérer les festivals dans un tableau

🖥️ Résultat attendu

Même résultat que précédemment mais un tableau qui permettra à l'avenir d'ajouter plus facilement des festivals mais aussi d'utiliser les flux de contrôle