PARTIE 2: COMMUNICATION ENTRE COMPOSANTS ET PIPES

COMMUNICATION VERS LE DESCENDANT DIRECT (1/3)

Le composant descendant doit prendre une valeur (qui peut être un objet complexe) en entrée dans une propriété de la classe qui est un **signal `input`**. C'est l'équivalent d'une *props* dans Vue.js.

```
export class myComponent {
  // Déclare un input 'score' avec 0 comme valeur par défaut.
  readonly score = input(0);
}
```

```
  // Déclare un input 'surnom' requis.
  readonly surnom = input.required<string>();
```

COMMUNICATION VERS LE DESCENDANT DIRECT (2/3)

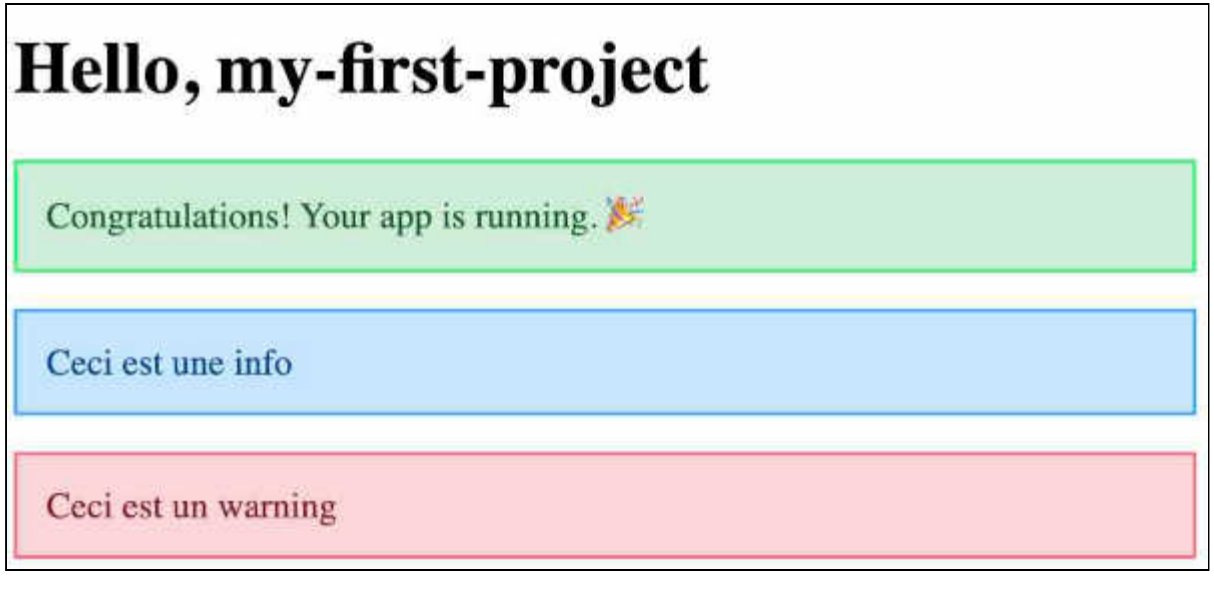
Dans le template du parent, le passage d'une valeur pour la propriété du descendant est possible en utilisant la syntaxe des attributs.

```
<app-descendant surnom="toto"/>
```

```
<app-descendant [score]="expression-a-evaluer"/>
```

COMMUNICATION VERS LE DESCENDANT DIRECT (3/3)

Dans votre projet, faire en sorte que le `message` affiché par le composant Alert soit forcément transmis par son composant parent. La catégorie (`category`) du message pourra l'être également (sinon ce sera par défaut 'info'). Votre application pourra ainsi afficher les trois types de messages.



CYCLE DE VIE D'UN COMPOSANT

- un `input` est transmis après que le composant soit construit
- On peut utiliser sa valeur dans le *hook* **ngOnInit** mais pas dans le constructeur
- **ngOnChanges** permet de lancer des instructions quand un ou plusieurs `input`(s) change(nt)
- **ngOnDestroy** permet de nettoyer le composant (supprimer des intervalles par exemple) avant sa destruction pour éviter les fuites mémoire

RÉAGIR AUX CHANGEMENTS DE SIGNAUX

- Une façon plus simple d'obtenir la transformation de la valeur d'un signal est d'utiliser un signal **computed**, dont la valeur est calculée et mémorisée lorsqu'un des signaux dont il dépend est modifié.

```
protected readonly login = computed(() => `${this.surnom().toLowerCase()}
```

- Lorsque ce sont des instructions à exécuter dans un composant, en réponse à la modification d'un signal, un effet est adapté. Il est toujours exécuté une fois et est détruit avec le composant.

```
protected readonly score = signal(0);
constructor() {
  effect(() => console.log(`Votre nouveau score: ${this.score()}`));
}
```

COMMUNICATION VERS LE PARENT

- Le descendant émet un événement pour transmettre une information à son parent. C'est l'équivalent d'un *emits* de Vue.js.
- Pour cela, il doit posséder une propriété définie comme `output`

```
readonly changeScore = output<number>();
```

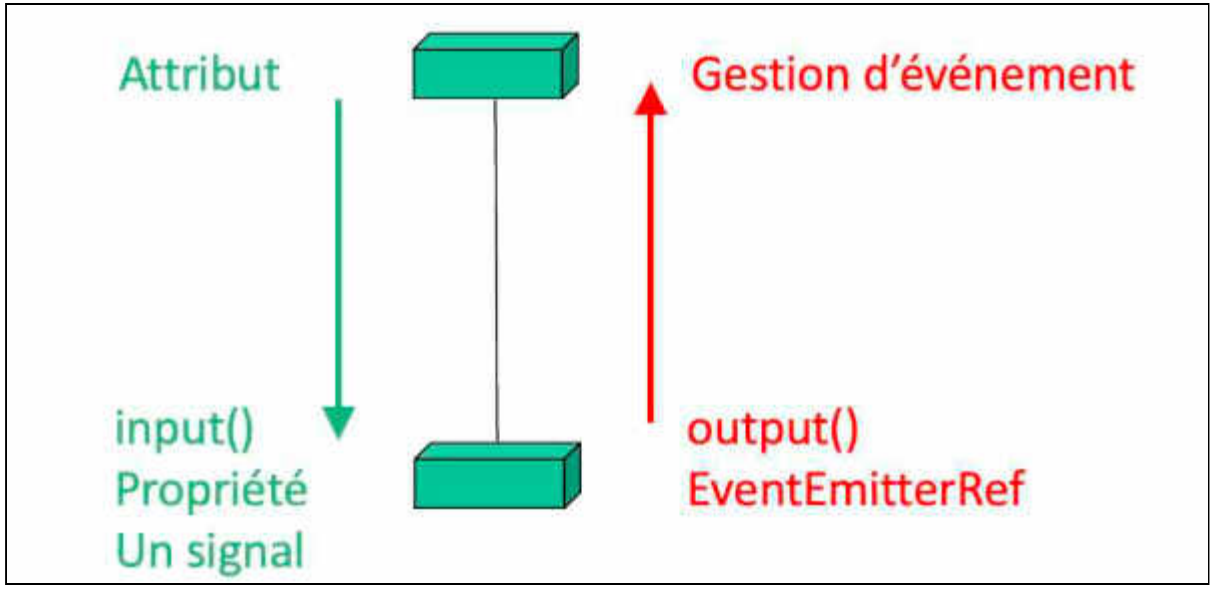
- La méthode emit de l'`output` permettra ensuite de déclencher l'événement, éventuellement porteur d'une information (un nombre dans l'exemple)

```
this.changeScore.emit(20);
```

ECOUTE DU PARENT

```
<app-descendant (changeScore)="onChangeScore($event)" />
```

```
// méthode du composant parent
onChangeScore(newScore: number): void {
  //fait quelque chose
}
```



$event est la valeur portée par l'événement (booléen, nombre, objet littéral…).

REFACTORER

Modifier le code du projet votre projet afin :

- d'ajouter un modèle de message, MessageModel pour les messages d'alerte. En plus du texte et de la catégorie d'alerte, un identifiant (number) doit être renseigné
- de stocker les messages d'alerte à afficher dans un signal qui contient un tableau tabMessages dans le composant App
- de modifier le composant Alert pour qu'une seule propriété `input` message lui soit transmise

AJOUTER UNE FONCTIONNALITÉ (1/2)

Nous allons ajouter la possibilité de fermer une alerte, ce qui devra avoir pour effet de supprimer le messages correspondant dans `tabMessages`.

- Ajoutons un bouton dans le composant Alert.

- Lorsqu'on clique sur la croix, le message sera supprimé de tabMessages et donc disparaîtra de la page.

AJOUTER UNE FONCTIONNALITÉ (2/2)

- Un bouton accessible (à compléter) à intégrer dans le parapgraphe

```
<button aria-label="Close">
  &times;
</button>
```

- du style pour l'intégrer dans le composant Alert

```
button {
  border: 0;
  background-color: transparent;
  font-size: 1.5rem;
  position: absolute;
  top: 0.1rem;
  right:0.1rem;
}

p {
  ...
  position: relative;
}
```

A vous de jouer!

LES PIPES

- Le pipe | est un opérateur spécial dans les templates qui permet de formater une donnée pour l'affichage sans la modifier.

```
import { UpperCasePipe } from '@angular/common';
@Component({
  selector: 'app-composant',
  standalone: true,
  imports: [UpperCasePipe],
  ...
})
```

```
<p>Ceci est la page d'accueil de {{user().prenom | uppercase}}
   {{user().nom | uppercase}}</p>
```

Dans votre projet, utiliser un pipe pour afficher les messages d'alerte avec une majuscule au début de chaque mot.

LES PIPES AVEC PARAMÈTRES

Les pipes d'Angular peuvent avoir zéro, un ou plusieurs paramètres (séparés par ":").

```
{{ expression | nomPipe : param1 : param2 }}
```

Le pipe DecimalPipe permet de formatter les nombres.

```
{{ 42.127 | number : '1.2-2' }}  // 42.13
{{ 42.1 | number : '1.2-2' }}  // 42.10
```

Le pipe DatePipe permet de formatter les dates.

```
<p>{{ '06/15/2019' | date : 'd MMMM y'}}</p> // 15 june 2019
```

LES PIPES ET L'INTERNATIONALISATION

- Le formattage des données par certains pipes intégrés au langage Angular dépend de la langue utilisée. C'est le cas pour les nombres et les dates.
- Par défaut, une application Angular est configurée avec 'en-US' (anglais américain). Pour un site en français, il faudra choisir 'fr-FR' (en France) ou 'fr-CA' pour un site canadien en Français.

```
{language_id}-{locale_extension}
```

LES PIPES ET L'INTERNATIONALISATION

- si la langue à utiliser est différente de l'anglais américain, il faut importer les données correspondant à cette langue

```
// dans main.ts
import '@angular/common/locales/global/fr'
```

- Les pipes peuvent prendre comme dernier paramètre la langue à utiliser

```
<p>{{ '06/15/2019' | date : 'd MMMM y' : '' : 'fr-FR'}}</p>
<!-- 15 juin 2019 -->
<p>{{ 42.127 | number : '1.2-2' : 'fr-FR' }}</p>
<!-- 42,13 -->
```

OU CHANGER LA LANGUE UTILISÉE PAR L'APPLICATION

Etape 1 => dans `main.ts`, l'import est nécessaire

```
import '@angular/common/locales/global/fr'
```

Etape 2 => dans `app.config.ts` (paramètre de `bootstrapApplication` de `main.ts`),

```
import { ApplicationConfig, LOCALE_ID, ... } from '@angular/core';

export const appConfig: ApplicationConfig = {
  providers: [
    { provide: LOCALE_ID, useValue: 'fr-FR' },
    ...
  ]
};
```

Etape 3 => dans `index.html`, modifiez l'attribut `lang` (`"fr"`) pour plus de cohérence.

MODIFIEZ LE CODE DE VOTRE PROJET

- votre projet est un site entièrement en français.
- changez le contenu de la balise `title` dans `index.html` et du titre de niveau 1 dans `app.html` => Mon premier projet
- améliorez l'affichage des exercices en écrivant clairement quelle est leur date limite de rendu

Vos exercices

- exercice 1: intro Typescript | à rendre le lundi 27 octobre 2025 | rendu
- exercice 2: démarrage du joli gestionnaire de devoirs | à rendre le mercredi 3 décembre 2025 | non rendu
- exercice 3: ajout d'un routeur | à rendre le jeudi 22 janvier 2026 | non rendu

INTÉRÊTS DES PIPE

- les pipes intégrés sont pratiques pour gérer les affichages de nombres, de dates et de sommes d'argent.
- les pipes purs utilisent un cache. Ils ne sont pas ré-exécutés si la valeur de la variable transformée (ou sa référence) n'a pas changé (comparable à *computed*). C'est la mémoïsation qui offre de meilleurs performances.
- on peut écrire un pipe personnalisé (voir aller plus loin) pour éviter une duplication de code dans plusieurs composants par exemple.

PIPES PURS OU IMPURS

Dans certaines circonstances très spécifiques, puisque ce sera forcément un coût de performances, il est souhaité une ré-évaluation à chaque détection de changement. On parle de pipe impur (sans mémoïsation).

Par exemple, le pipe JsonPipe, utilisé seulement pour du debug, met en forme l'affichage d'un objet littéral et sera ré-exécuté à chaque fois.

```
<p>{{ message }}</p>
```

[object Object]

```
<p>{{ message | json }}</p>
```

{ "id": 2, "text": "Ceci est une info", "category": "info" }

ALLER PLUS LOIN: CRÉATION DE PIPES PERSONNALISÉS

`ng generate pipe nom`

Code généré dans src/app/nom.pipe.ts:

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'nom',
  standalone: true
})
export class Nom implements PipeTransform {
  transform(value: unknown, ...args: unknown[]): unknown {
    return null;
  }
}
```

Il faut implémenter sa méthode `transform` en précisant ses paramètres et en retournant une valeur transformée.

On peut ensuite l'importer dans tous les composants de l'application qui en auront besoin.

ALLER PLUS LOIN: CRÉER UN PIPE PERSONNALISÉ

Pour le fun, créer un pipe qui affiche une chaîne de caractères à l'envers. Utilisez ce pipe pour afficher le titre de niveau 1.