

Opérateurs pipeables

- Essayons maintenant de combiner nos deux observables :

```
clients?: Client[]

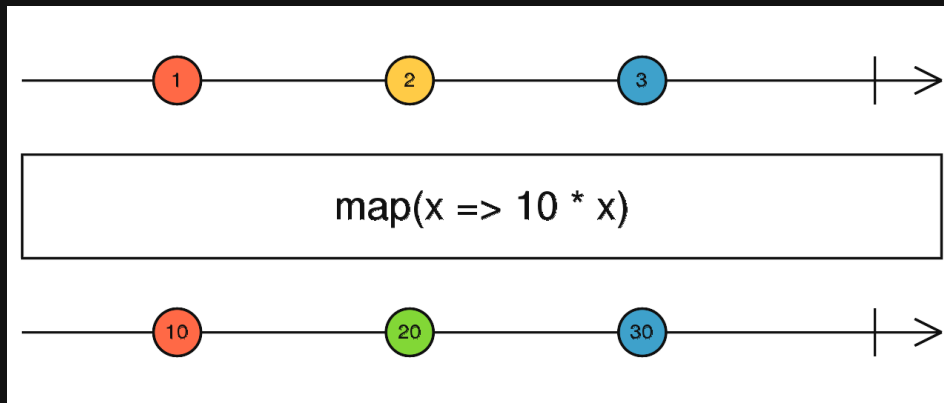
this.subscription = fromEvent(this.input.nativeElement, 'input').subscribe(
  ev => {
    let input = (ev.target as HTMLInputElement).value
    this.clientService.getFilteredSortedClients(input).subscribe(
      c => this.clients = c
    )
  }
)
```

- Notre exemple pourtant simple n'est pas si facile à écrire
- Le code se complexifie vite, avec des subscribe dans des subscribe, et des abonnements intermédiaires à gérer
- Ce phénomène est connu sous le nom de *Subscription Hell*

Opérateurs pipeables

- L'une des forces de RxJS est de pouvoir faire des opérations sur les observables, c'est à dire transformer un opérateur en un autre
- Un opérateur pipeables sont des fonctions, qui prennent des observables en entrée, et renvoient des observables en sortie
- Ces opérateurs ne modifient pas les observables d'entrée, mais ils créent de nouveaux observables
- L'opérateur gère l'abonnement et le désabonnement de tous les observables intermédiaires.
- Pour créer des opérateur pipeables, on va en général utiliser des factories, fournies par RxJS
 - Une factory est une fonction, qui renvoie un opérateur
 - Un opérateur est une fonction, qui renvoie un observable

Exemple et Marble Diagram



- Un observable en entrée : il génère les nombres 1, 2, 3, puis complète
- La factory `map(x => 10 * x)` génère un opérateur
- Cet opérateur transforme mon observable d'entrée en un nouvel observable qui génère 10, 20, 30, puis complète
- Vous pouvez utiliser les Marble Diagram pour vous aider à visualiser les opérateurs

map()

```
map<T, R>(project: (value: T, index: number) => R): OperatorFunction<T, R>
```

- La fonction map() est une factory, qui renvoie des OperatorFunction<T, R>
- Un OperatorFunction<T, R> est une fonction qui prend en entrée un Observable<T> et renvoie un Observable<R>
- Le callback project, paramètre de la fonction transforme les valeurs de type T émises par l'observable d'entrée en valeurs de type R émises par l'observable de sortie
- Le type T peut être le même que le type R
- L'utilisation est similaire à celle de map() avec des tableaux

Pipes (RxJS)

- `map()` renvoie une fonction
- L'écriture suivante est correcte

```
obs$ = map((i: number) => 10 * i)(this.interval$)
```

- En pratique, cette écriture est peu lisible, et encore plus si on veut enchaîner plusieurs opérateurs
- Pour résoudre le problème, la classe `Observable` a une méthode `pipe()`
- ⚠ Pipe RxJS ≠ Pipe Angular

Pipes (RxJS)

- La méthode `pipe()` de la classe `Observable` prends un `OperatorFunction<T, R>`, et renvoie un nouvel observable
- L'exemple précédent peut être écrit de la manière suivante

```
obsPipe$ = this.interval$.pipe(  
  map((i: number) => 10 * i)  
)
```

- Et pour les types de l'`OperatorFunction` ?
 - `T` doit être du même type que l'observable qui appelle `pipe()`
 - `R` est le type de retour de l'observable

Pipes (RxJS)

- On peut passer plusieurs OperatorFunction à la suite comme paramètres de la méthode pipe, ainsi :

```
obsPipe$ = this.interval$.pipe(  
  map((i: number) => 10 * i)  
).pipe(  
  map((i: number) => 'To String : ' + i)  
)
```

- Peut être écrit :

```
obsPipe$ = this.interval$.pipe(  
  map((i: number) => 10 * i),  
  map((i: number) => 'To String : ' + i)  
)
```



- Il faut s'assurer que le type de chaque opérateur de la chaîne est compatible avec le type suivant
- L'ordre des opérateurs dans la chaîne est important

Opérateurs RxJS

- Nous allons voir dans la suite quelques exemple d'opérateurs mis à disposition par RxJS
- <https://rxjs.dev/api/operators>
- <https://rxjs.dev/operator-decision-tree>
- Le but n'est pas de connaître par coeur tous les observables !

filter()

```
filter<T>(predicate: (value: T, index: number) => boolean): MonoTypeOperatorFunction<T>
```

- MonoTypeOperatorFunction<T> représente un opérateur qui renvoie un observable de même type que son entrée (OperatorFunction<T, T>)
- A chaque fois que l'observable source émet, la valeur émise est évaluée par le callback predicate, si il retourne true, l'observable renvoyé émet
- Que fait l'observable suivant ?

```
obs$ = interval(1000).pipe(  
  filter( i => i % 2 !== 0 )  
)
```

take()

```
take<T>(count: number): MonoTypeOperatorFunction<T>
```

- Crée un observable qui émet uniquement les n premières valeurs de l'observable source, puis complète
- Si l'observable source complète avant d'avoir émit n valeurs, l'observable généré crée complète également

first()

```
first<T, D>(  
    predicate?: (value: T, index: number, source: Observable<T>) => boolean,  
    defaultValue?: D  
): OperatorFunction<T, T | D>
```

- Crée un observable qui émet uniquement la première valeur de l'observable source, puis complète
- Cependant quelques différences avec take(1) :
 - Si l'observable source complète avant d'avoir émit une valeur, first émet une erreur, sauf si une valeur par défaut est définie
 - Il est possible de définir un prédicat, dans ce cas uniquement la première valeur qui satisfait ce prédicat sera renvoyée

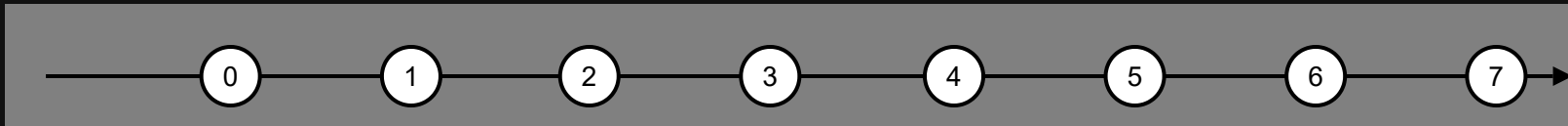
last()

```
last<T, D>(
  predicate?: (value: T, index: number, source: Observable<T>) => boolean,
  defaultValue?: D
): OperatorFunction<T, T | D>
```

- Similaire à first, mais renvoie la dernière valeur émise par l'observable source
- Pour s'assurer de la dernière valeur, l'observable généré par last émet uniquement lorsque l'observable source complète

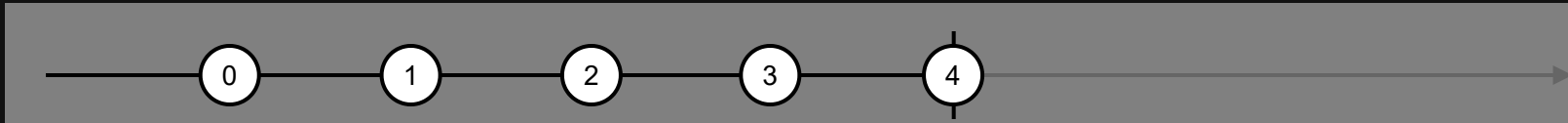
Exemple

- interval(1000)



- Toutes les secondes, une valeur est émise, l'observable ne complète jamais

- take(5)



- Les 5 premières valeurs sont émises, puis l'observable complète

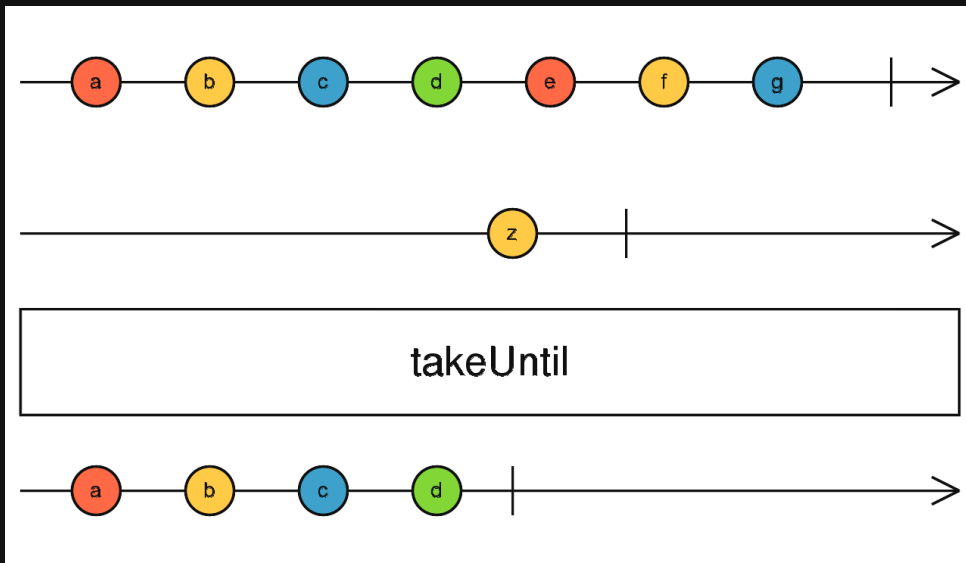
- last(x => x < 2)



- Lorsque l'observable, complète, la première valeur qui respecte la condition est émise

takeUntil()

```
takeUntil<T>(notifier: ObservableInput<any>): MonoTypeOperatorFunction<T>
```



- Émet les valeurs de l'observable source jusqu'à ce qu'une valeur de l'observable notifier soit émise, puis complète