Angular : RxJS

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

Plan

Angular

Programmation réactive

- paradigme de programmation orienté flux de données et propagation des changements
- inspiré du patron de conception observable
- Deux concepts importants :
  - **Observable**
  - **Observer**

Angular

Programmation réactive

- paradigme de programmation orienté flux de données et propagation des changements
- inspiré du patron de conception observable
- Deux concepts importants :
  - **Observable**
  - **Observer**

RxJS

Reactive e**x**tensions for **J**ava**S**cript

Angular

Observable

- Fonction retournant un flux de valeurs (à un observateur)
- De manière synchrone ou asynchrone.
- S'exécutant uniquement s'il y a un observateur (**observer**) et un abonnement (avec la méthode `subscribe()`)

Angular

Observer

- Consommateur de valeurs émises par l'**observable**
- Objet **JavaScript** qui se souscrit (**subscribe()**) à un observable
- Ensemble de trois callbacks :
  - `next` : contient une fonction qui s'exécute à la réception de chaque valeur émise par l'observable (la valeur est reçue comme paramètre)
  - `error` : contient une fonction qui s'exécute en cas d'erreur, et reçoit cette erreur comme paramètre
  - `complete` : contient une fonction qui s'exécute lorsque l'observable se termine, et ne reçoit aucun paramètre.

Angular

Remarques

- La méthode `subscribe()` de l'**observable** prend comme paramètre un **observer**
- Il n'est pas nécessaire d'implémenter les trois fonctions de l'**observer**. Seule la première peut être implémentée si besoin.

Angular

Commençons par

- créer un composant `observable`
- ajouter `<app-observable>< /app-observable>` dans `app.html`
- renommer la classe `Observable` en `ObservableComponent`

Le fichier `observable.ts`

```
import { Component, OnInit, signal } from '@angular/core';

@Component({
  selector: 'app-observable',
  imports: [],
  templateUrl: './observable.html',
  styleUrl: './observable.css'
})
export class ObservableComponent implements OnInit {

  tab = signal<number[]>([]);
  status = signal('');
  constructor() { }
  ngOnInit() { }
}
```

Le fichier `observable.ts`

```
import { Component, OnInit, signal } from '@angular/core';

@Component({
  selector: 'app-observable',
  imports: [],
  templateUrl: './observable.html',
  styleUrl: './observable.css'
})
export class ObservableComponent implements OnInit {

  tab = signal<number[]>([]);
  status = signal('');
  constructor() { }
  ngOnInit() { }
}
```


Le fichier `observable.html`

```
<h2>éléments</h2>
<ul>
    @for (elt of tab(); track elt) {
    <li>
        {{ elt }}
    </li>
    }
</ul>
<div>{{ status() }}</div>
```

Angular

Pour créer un observable vide, on peut utiliser `EMPTY`

```
import { Component, OnInit, signal } from '@angular/core';
import { EMPTY, Observable } from 'rxjs';

@Component({
  selector: 'app-observable',
  templateUrl: './observable.html',
  styleUrls: ['./observable.css']
})
export class ObservableComponent implements OnInit {

  tab = signal<number[]>([]);
  status = signal('');
  constructor() { }
  ngOnInit(): void {
    const observable$: Observable<number> = EMPTY;
  }
}
```

Angular

Pour créer un observable vide, on peut utiliser `EMPTY`

```
import { Component, OnInit, signal } from '@angular/core';
import { EMPTY, Observable } from 'rxjs';

@Component({
  selector: 'app-observable',
  templateUrl: './observable.html',
  styleUrls: ['./observable.css']
})
export class ObservableComponent implements OnInit {

  tab = signal<number[]>([]);
  status = signal('');
  constructor() { }
  ngOnInit(): void {
    const observable$: Observable<number> = EMPTY;
  }
}
```

Il est recommandé de suffixer le nom d'un observable par `$`

Angular

Pour créer un observable, on peut utiliser la méthode `of()` qui convertit un ensemble de paramètres en `observable`

```
import { Component, OnInit, signal } from '@angular/core';
import { Observable, of } from 'rxjs';

@Component({
  selector: 'app-observable',
  templateUrl: './observable.html',
  styleUrls: ['./observable.css']
})
export class ObservableComponent implements OnInit {

  tab = signal<number[]>([]);
  status = signal('');
  constructor() { }
  ngOnInit(): void {
    const observable$: Observable<number> = of(1, 2, 3);
  }
}
```

Angular

On peut utiliser `from()` pour construire un observable à partir d'un tableau

```
import { Component, OnInit, signal } from '@angular/core';
import { Observable, from } from 'rxjs';

@Component({
  selector: 'app-observable',
  templateUrl: './observable.html',
  styleUrls: ['./observable.css']
})
export class ObservableComponent implements OnInit {

  tab = signal<number[]>([]);
  status = signal('');
  constructor() { }
  ngOnInit() {
    const tableau = [1, 2, 3];
    const observable$: Observable<number> = from(tableau);
  }
}
```

Angular

On peut utiliser `range()` pour définir un intervalle de nombres entiers

```
import { Component, OnInit, signal } from '@angular/core';
import { Observable, range } from 'rxjs';

@Component({
  selector: 'app-observable',
  templateUrl: './observable.html',
  styleUrls: ['./observable.css']
})
export class ObservableComponent implements OnInit {

  tab = signal<number[]>([]);
  status = signal('');
  constructor() { }
  ngOnInit() {
    const observable$: Observable<number> = range(1, 3);
  }
}
```

Angular

Lorsqu'un observateur s'abonne à notre observable, il peut implémenter trois méthodes pour spécifier ce qu'il faut faire

```
ngOnInit() {
    const tableau = [1, 2, 3];
    const observable$: Observable<number> = from(tableau);
    const observer: Observer<number> = {
        next: (value) => {
            this.tab.update(currentTab => [...currentTab, value]);
        },
        error: (error) => {
            this.status.set(error);
        },
        complete:   () => {
            this.status.set('fini');
        }
    };
}
```

Angular

Ensuite la souscription

```
ngOnInit() {
    const tableau = [1, 2, 3];
    const observable$: Observable<number> = from(tableau);
    const observer: Observer<number> = {
        next: (value) => {
            this.tab.update(currentTab => [...currentTab, value]);
        },
        error: (error) => {
            this.status.set(error);
        },
        complete:   () => {
            this.status.set('fini');
        }
    };
    observable$.subscribe(observer)
}
```

Angular

On peut aussi faire

```
ngOnInit() {
  const tableau = [1, 2, 3];
  const observable$: Observable<number> = from(tableau);
  observable$.subscribe({
    next: (value) => {
      this.tab.update(currentTab => [...currentTab, value]);
    },
    error: (error) => {
      this.status.set(error);
    },
    complete: () => {
      this.status.set('fini');
    }
  });
}
```

L'émission de valeurs s'effectue d'une manière synchrone et par conséquent on ne voit pas la réception des éléments dans le navigateur.

Angular

Pour avoir une exécution asynchrone, on utilise la fonction `interval(1000)` une infinité de valeurs incrémentielles commençant de 0 : une valeur par seconde

```
ngOnInit() {
  const observable$: Observable<number> = interval(1000);
  observable$.subscribe({
    next: (value) => {
      this.tab.update(currentTab => [...currentTab, value]);
    },
    error: (error) => {
      this.status.set(error);
    },
    complete: () => {
      this.status.set('fini');
    }
  });
}
```

Pour avoir une exécution asynchrone, on utilise la fonction `timer(3000, 1000)` qui envoie une infinité de valeurs incrémentielles commençant de 0 : une valeur par seconde la première valeur sera envoyée après 3 secondes

```
ngOnInit() {
  const observable$: Observable<number> = timer(3000, 1000);
  observable$.subscribe({
    next: (value) => {
      this.tab.update(currentTab => [...currentTab, value]);
    },
    error: (error) => {
      this.status.set(error);
    },
    complete: () => {
      this.status.set('fini');
    }
  });
}
```

Les deux fonctions `timer()` et `interval()` ne se terminent jamais.

Angular

Pour éviter les problèmes de mémoire, il faut penser à se désabonner à la destruction du composant

```
import { Component, OnInit, OnDestroy, signal } from '@angular/core';
import { Observable, Subscription, timer } from 'rxjs';

@Component({
  selector: 'app-observable',
  templateUrl: './observable.html',
  styleUrls: ['./observable.css']
})
export class ObservableComponent implements OnInit, OnDestroy {

  tab = signal<number[]>([]);
  status = signal('');
  subscription!: Subscription;

  constructor() { }

  ngOnInit() {
    const observable$: Observable<number> = timer(3000, 1000);
    this.subscription = observable$.subscribe({
      next: (value) => this.tab.update(currentTab => [...currentTab, value]),
      error: (error) => this.status.set(error),
      complete: () => this.status.set('fini')
    });
  }
  ngOnDestroy() { this.subscription.unsubscribe(); }
}
```

Angular

Pour indiquer le nombre d'élément à émettre, on utilise la méthode `pipe()` qui nous permet de faire appel à l'opérateur `take()`

```
ngOnInit() {
  const observable$: Observable<number> = interval(1000).pipe(take(10));

  observable$.subscribe({
    next: (value) => {
      this.tab.update(currentTab => [...currentTab, value]);
    },
    error: (error) => {
      this.status.set(error);
    },
    complete: () => {
      this.status.set('fini');
    }
  });
}
```

On peut importer les opérateurs ainsi : `import { take } from 'rxjs';`

Angular

On peut combiner les opérateurs en appliquant une modification sur les 10 éléments reçus

```
ngOnInit() {
  const observable$: Observable<number> = interval(1000).pipe(
          take(10),
          map(elt => elt + 3)
  );

  observable$.subscribe({
    next: (value) => {
      this.tab.update(currentTab => [...currentTab, value]);
    },
    error: (error) => {
      this.status.set(error);
    },
    complete: () => {
      this.status.set('fini');
    }
  });
}
```

Angular

On peut aussi filtrer les éléments pairs

```
ngOnInit() {
  const observable$: Observable<number> = interval(1000).pipe(
          take(10),
          map(elt => elt + 3),
          filter(elt => elt % 2 === 0)
  );

  observable$.subscribe({
    next: (value) => {
      this.tab.update(currentTab => [...currentTab, value]);
    },
    error: (error) => {
      this.status.set(error);
    },
    complete: () => {
      this.status.set('fini');
    }
  });
}
```

Pour suivre l'évolution des valeurs, on peut utiliser `tap`

```
ngOnInit() {
  const observable$: Observable<number> = interval(1000).pipe(
          take(10),
          tap(elt => console.log(elt)),
          map(elt => elt + 3),
          tap(elt => console.log(elt)),
          filter(elt => elt % 2 === 0),
          tap(elt => console.log(elt))
  );

  observable$.subscribe({
    next: (value) => {
      this.tab.update(currentTab => [...currentTab, value]);
    },
    error: (error) => {
      this.status.set(error);
    },
    complete: () => {
      this.status.set('fini');
    }
  });
}
```

Angular

Opérateurs d'agrégation

- `count`
- `max`
- `min`
- `reduce`

Angular

On peut aussi compter les éléments selon un critère

```
ngOnInit() {
  const observable$: Observable<number> = interval(1000).pipe(
          take(10),
          count(elt => elt % 2 === 0)
  );

  observable$.subscribe({
    next: (value) => {
      this.tab.update(currentTab => [...currentTab, value]);
    },
    error: (error) => {
      this.status.set(error);
    },
    complete: () => {
      this.status.set('fini');
    }
  });
}
```

Angular

On peut sélectionner le maximum

```
ngOnInit() {
  const observable$: Observable<number> = interval(1000).pipe(
          take(10),
          max()
  );

  observable$.subscribe({
    next: (value) => {
      this.tab.update(currentTab => [...currentTab, value]);
    },
    error: (error) => {
      this.status.set(error);
    },
    complete: () => {
      this.status.set('fini');
    }
  });
}
```

Angular

Exercice

Écrire un observable qui émet les 5 premiers nombres pairs compris entre 10 et 15.

Angular

On peut fusionner plusieurs observables avec `merge()`

```
ngOnInit() {
  const observable1$: Observable<number> = of(1, 2, 3);
  const observable2$: Observable<number> = of(4, 5, 6);

  const merged$ = merge(observable1$, observable2$);

  merged$.subscribe({
    next: (value) => {
      this.tab.update(currentTab => [...currentTab, value]);
    },
    error: (error) => {
      this.status.set(error);
    },
    complete: () => {
      this.status.set('fini');
    }
  });
}
```

Exercice : qu'affiche le code suivant ?

```
ngOnInit() {
  const strings$: Observable<string> = of('A', 'B', 'C');
  const numbers$: Observable<number> = of(1, 2, 3);
  strings$.subscribe(str => {
    numbers$.subscribe(nbr => {
      console.log(str + nbr);

    })
  })
}
```

Exercice : qu'affiche le code suivant ?

```
ngOnInit() {
  const strings$: Observable<string> = of('A', 'B', 'C');
  const numbers$: Observable<number> = of(1, 2, 3);
  strings$.subscribe(str => {
    numbers$.subscribe(nbr => {
      console.log(str + nbr);

    })
  })
}
```

Réponse

```
A1
A2
A3
B1
B2
B3
C1
C2
C3
```

Angular

Cependant, imbriquer les souscriptions est déconseillé pour les raisons suivantes

- Code plus complexe $\Rightarrow$ risque d'erreur plus élevé
- Haut risque de fuite de mémoire (si désabonnement oubliée)
- Perte de performance
- ...

Angular

Pour éviter les souscriptions imbriquées, on peut utiliser

- `switchMap` : attend la fin de l'observable externe pour créer l'interne (les valeurs de l'observable externe risque d'être perdues)
- `concatMap` : ordonne les valeurs selon l'observable externe
- `mergeMap` (`flatMap` déprécié) : ordonne les valeurs selon l'observable interne
- ...

Angular

Remplaçons les imbrications de souscription par `switchMap`

```
ngOnInit() {
  const strings$: Observable<string> = of('A', 'B', 'C');
  const numbers$: Observable<number> = of(1, 2, 3);
  strings$.pipe(
    switchMap((str) => numbers$.pipe(
      map((nbr) => str + nbr)
    ))
  ).subscribe((result) => {
    console.log(result);
  });
}
```

Angular

Remplaçons les imbrications de souscription par `switchMap`

```
ngOnInit() {
  const strings$: Observable<string> = of('A', 'B', 'C');
  const numbers$: Observable<number> = of(1, 2, 3);
  strings$.pipe(
    switchMap((str) => numbers$.pipe(
      map((nbr) => str + nbr)
    ))
  ).subscribe((result) => {
    console.log(result);
  });
}
```


Le résultat est le même

```
A1
A2
A3
B1
B2
B3
C1
C2
C3
```

Angular

Si une nouvelle valeur est émise avant que l'observable interne précédent n'ait terminé, l'observable interne précédent est annulé et remplacé par le nouvel observable interne.

```
ngOnInit() {
  const strings$: Observable<string> = of('A', 'B', 'C');
  const numbers$: Observable<number> = interval(1000).pipe(take(3))

  strings$.pipe(
    switchMap((str) => numbers$.pipe(
      map((nbr) => str + nbr)
    ))
  ).subscribe((result) => {
    console.log(result);
  });
}
```

Angular

Si une nouvelle valeur est émise avant que l'observable interne précédent n'ait terminé, l'observable interne précédent est annulé et remplacé par le nouvel observable interne.

```
ngOnInit() {
  const strings$: Observable<string> = of('A', 'B', 'C');
  const numbers$: Observable<number> = interval(1000).pipe(take(3))

  strings$.pipe(
    switchMap((str) => numbers$.pipe(
      map((nbr) => str + nbr)
    ))
  ).subscribe((result) => {
    console.log(result);
  });
}
```

Le résultat

```
C0
C1
C2
```

Angular

`concatMap` **ordonne les valeurs selon l'observable externe**

```
ngOnInit() {
  const strings$: Observable<string> = of('A', 'B', 'C');
  const numbers$: Observable<number> = interval(1000).pipe(take(3))

  strings$.pipe(
    concatMap((str) => numbers$.pipe(
      map((nbr) => str + nbr)
    ))
  ).subscribe((result) => {
    console.log(result);
  });
}
```

Angular

`concatMap` **ordonne les valeurs selon l'observable externe**

```
ngOnInit() {
  const strings$: Observable<string> = of('A', 'B', 'C');
  const numbers$: Observable<number> = interval(1000).pipe(take(3))

  strings$.pipe(
    concatMap((str) => numbers$.pipe(
      map((nbr) => str + nbr)
    ))
  ).subscribe((result) => {
    console.log(result);
  });
}
```


Le résultat

```
A0
A1
A2
B0
B1
B2
C0
C1
C2
```

Angular

`mergeMap` **ordonne les valeurs selon l'observable interne**

```
ngOnInit() {
  const strings$: Observable<string> = of('A', 'B', 'C');
  const numbers$: Observable<number> = interval(1000).pipe(take(3))

  strings$.pipe(
    mergeMap((str) => numbers$.pipe(
      map((nbr) => str + nbr)
    ))
  ).subscribe((result) => {
    console.log(result);
  });
}
```

Angular

`mergeMap` **ordonne les valeurs selon l'observable interne**

```
ngOnInit() {
  const strings$: Observable<string> = of('A', 'B', 'C');
  const numbers$: Observable<number> = interval(1000).pipe(take(3))

  strings$.pipe(
    mergeMap((str) => numbers$.pipe(
      map((nbr) => str + nbr)
    ))
  ).subscribe((result) => {
    console.log(result);
  });
}
```


Le résultat

```
A0
B0
C0
A1
B1
C1
A2
B2
C2
```

Angular

Observable froid (Cold Observable)

- Paresseux : ne produit des valeurs qu'après abonnement.
- Chaque abonné reçoit son propre flux depuis le début.
- Exemples : `of()`, `from()`, `ajax()`, `interval()`.

Angular

Exemple

```
const cold$ = interval(1000);

cold$.subscribe(v => console.log("A :", v));

setTimeout(() => {
    cold$.subscribe(v => console.log("B :", v));
}, 3000);
```

Angular

Exemple

```
const cold$ = interval(1000);

cold$.subscribe(v => console.log("A :", v));

setTimeout(() => {
    cold$.subscribe(v => console.log("B :", v));
}, 3000);
```


Constat

B recommence à zéro.

Angular

Observable chaud (Hot Observable)

- Producteur partagé, indépendant des abonnés.
- Les abonnés rejoignent en cours de route.
- Exemples : `fromEvent()`, WebSocket, Subject.

Angular

Exemple

```
const hot$ = fromEvent(document, "click");

hot$.subscribe(() => console.log("A clique"));

setTimeout(() => {
    hot$.subscribe(() => console.log("B clique"));
}, 5000);
```

Angular

Exemple

```
const hot$ = fromEvent(document, "click");

hot$.subscribe(() => console.log("A clique"));

setTimeout(() => {
    hot$.subscribe(() => console.log("B clique"));
}, 5000);
```


Constat

B ne voit pas les clics d'avant.

Angular

Cold vs Hot

	Cold	**Hot**
Flux	Rejoué pour chaque abonné	Partagé entre abonnés
Début	À l'abonnement	Tout de suite
Exemple	`interval(),timer()`	`fromEvent()`

Angular

Subject

- Il a à la fois le rôle d'un observateur et d'un observable
- Il diffuse à tous les abonnés présents.
- Il peut utiliser la méthode `next` pour émettre de données à ses observateurs

Angular

Subject

- Il a à la fois le rôle d'un observateur et d'un observable
- Il diffuse à tous les abonnés présents.
- Il peut utiliser la méthode `next` pour émettre de données à ses observateurs

Remarques

- Un Observable classique peut avoir plusieurs abonnés, mais chaque abonné reçoit son propre flux indépendant.
- Avec un **Subject**, tous les abonnés partagent le même flux.
- L'opérateur `multicast` permet de transformer un Observable en `ConnectableObservable` pour diffuser un flux commun.

Angular

Dans `ngOnInit()`, commençons par déclarer un `Subject`

```
const subject = new Subject<number>();
```

Angular

Dans `ngOnInit()`, commençons par déclarer un `Subject`

```
const subject = new Subject<number>();
```

Plusieurs observateurs peuvent s'abonner à notre `subject`

```
subject.subscribe({
    next: (value) => console.log(`A : ${value}`)
});
subject.subscribe({
    next: (value) => console.log(`B : ${value}`)
});
```

Pour émettre une valeur, le `Subject` utilise la méthode `next`

```
const subject = new Subject<number>();

subject.subscribe({
    next: (value) => console.log(`A : ${value}`)
});
subject.subscribe({
    next: (value) => console.log(`B : ${value}`)
});

subject.next(1);
subject.next(2);
```

Pour émettre une valeur, le `Subject` utilise la méthode `next`

```
const subject = new Subject<number>();

subject.subscribe({
    next: (value) => console.log(`A : ${value}`)
});
subject.subscribe({
    next: (value) => console.log(`B : ${value}`)
});

subject.next(1);
subject.next(2);
```


Le résultat est

```
// le résultat est
// A : 1
// B : 1
// A : 2
// B : 2
```

Angular

Exercice : qu'affiche le programme suivant

```
const subject = new Subject<number>();

subject.next(0);

subject.subscribe({
    next: (value) => console.log(`A : ${value}`)
});

subject.next(1);

subject.subscribe({
    next: (value) => console.log(`B : ${value}`)
});

subject.next(2);
```

Un `Subject` est aussi un observateur, il peut donc s'abonner à un observable

```
ngOnInit() {
    const subject = new Subject<number>();

    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });

    observable$.subscribe(subject);
}
```

Un `Subject` est aussi un observateur, il peut donc s'abonner à un observable

```
ngOnInit() {
    const subject = new Subject<number>();

    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });

    observable$.subscribe(subject);
}
```


Le résultat est

```
// A : 1
// B : 1
// A : 2
// B : 2
// A : 3
// B : 3
```

Angular

Behavior Subject

- Une des variantes de `Subject` fonctionnant avec la notion de valeur actuelle
- Il stocke la dernière valeur émise par ses observateurs
- Chaque fois qu'un nouvel observateur s'abonne, il reçoit immédiatement la valeur actuelle
- Le constructeur de la classe `BehaviorSubject` prend comme paramètre la valeur initiale

Exemple

```
ngOnInit() {
    const subject = new BehaviorSubject(0);
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.next(1);
    subject.next(2);
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });
    subject.next(3);
}
```

Exemple

```
ngOnInit() {
    const subject = new BehaviorSubject(0);
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.next(1);
    subject.next(2);
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });
    subject.next(3);
}
```


Le résultat est

```
// A : 0
// A : 1
// A : 2
// B : 2
// A : 3
// B : 3
```

Angular

ReplaySubject

- Une des variantes de `Subject` fonctionnant avec la notion de nombre de valeurs à conserver dans l'historique
- Il conserve un nombre de valeurs dans l'historique afin qu'elles puissent être envoyées aux nouveaux abonnés.
- Le constructeur de la classe `ReplaySubject` prend comme paramètre le nombre de valeurs à conserver dans l'historique

Exemple

```
ngOnInit() {
    const subject = new ReplaySubject<number>(2);
    subject.next(0);
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.next(1);
    subject.next(2);
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });
    subject.next(3);
}
```

Exemple

```
ngOnInit() {
    const subject = new ReplaySubject<number>(2);
    subject.next(0);
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.next(1);
    subject.next(2);
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });
    subject.next(3);
}
```


Le résultat est

```
// A : 0
// A : 1
// A : 2
// B : 1
// B : 2
// A : 3
// B : 3
```

Angular

Async Subject

- Une variante particulière de `Subject`.
- Il ne transmet qu'une seule valeur à ses abonnés : **la dernière émise**, et uniquement au moment de la complétion.
- L'appel à la méthode `complete()` est nécessaire pour déclencher l'envoi de cette valeur finale.
- Tous les abonnés reçoivent donc la même valeur, peu importe le moment de leur souscription. l'envoi de cet

Exemple

```
ngOnInit() {
    const subject = new AsyncSubject();

    subject.next(0);
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.next(1);
    subject.next(2);
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });
    subject.next(3);

    subject.complete();
}
```

Exemple

```
ngOnInit() {
    const subject = new AsyncSubject();

    subject.next(0);
    subject.subscribe({
      next: (value) => console.log(`A : ${value}`)
    });
    subject.next(1);
    subject.next(2);
    subject.subscribe({
      next: (value) => console.log(`B : ${value}`)
    });
    subject.next(3);

    subject.complete();
}
```

Le résultat est

```
// A : 3
// B : 3
```