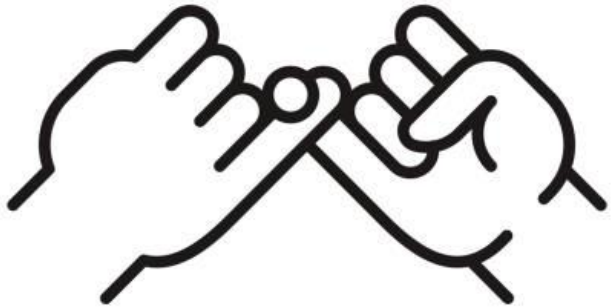
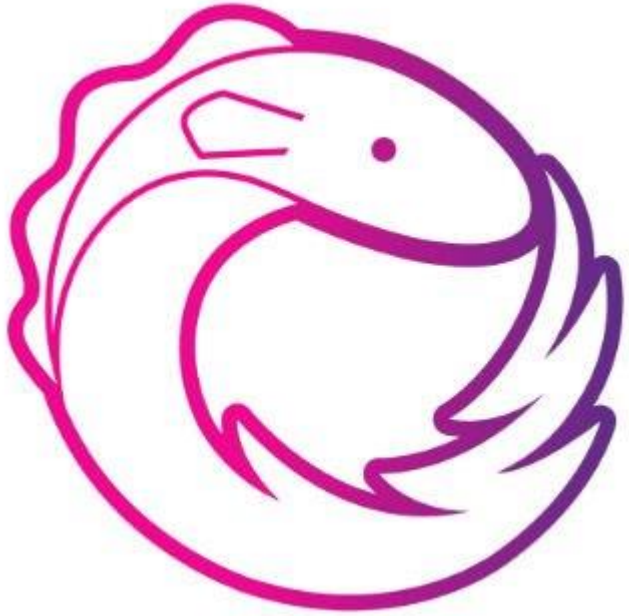
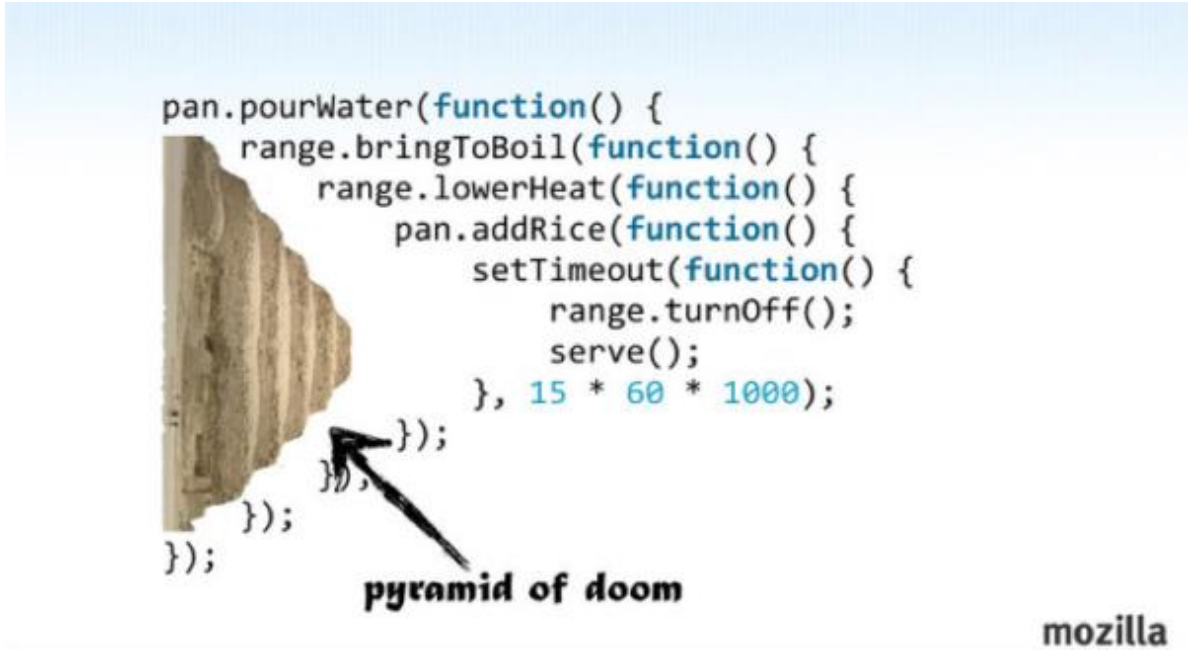




Promesses & Observables

Animé par Mazen Gharbi

Promesses




```
document.querySelector("#mon_bouton").addEventListener(() => {
    callBackend('/mes-cadeaux-de-noel', (cadeaux) => {
        setTimeout(() => {
            alert(`Avec un peu d'attente, voici enfin vos cadeaux de Noël -> ${cadeaux}`)
        }, 10000);
    });
});
```

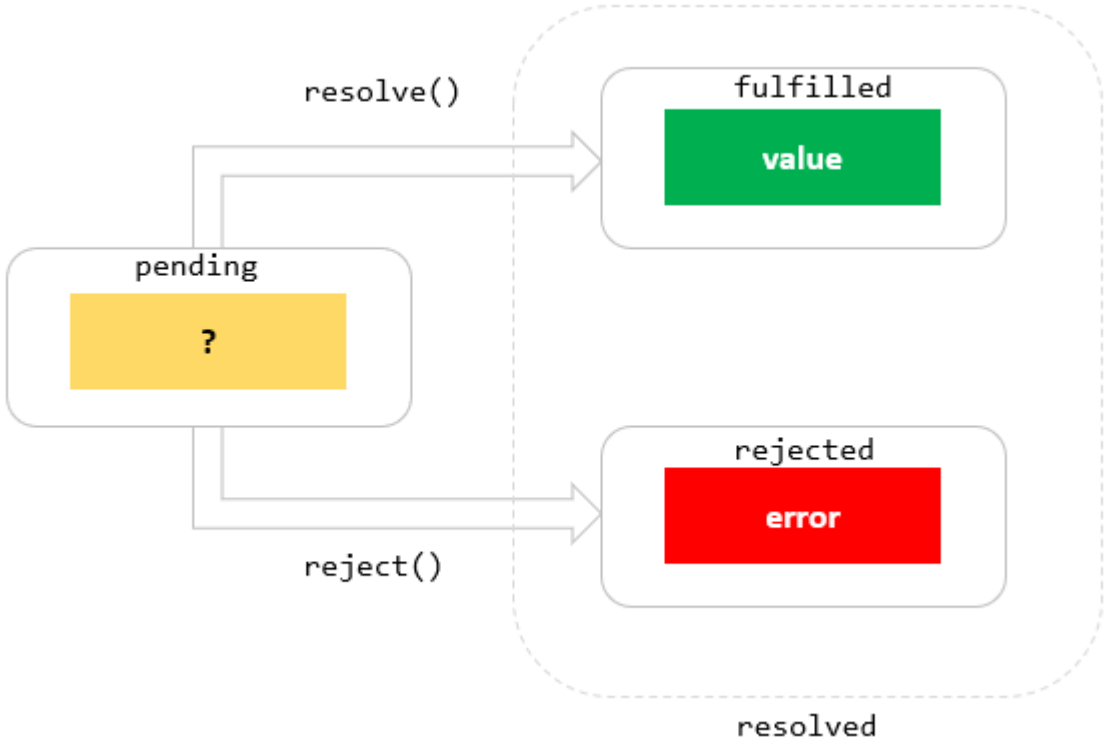
Promesses

- Les promesses font désormais parti des fonctionnalités ES6
  - Digne successeur de la librairie « q promise »

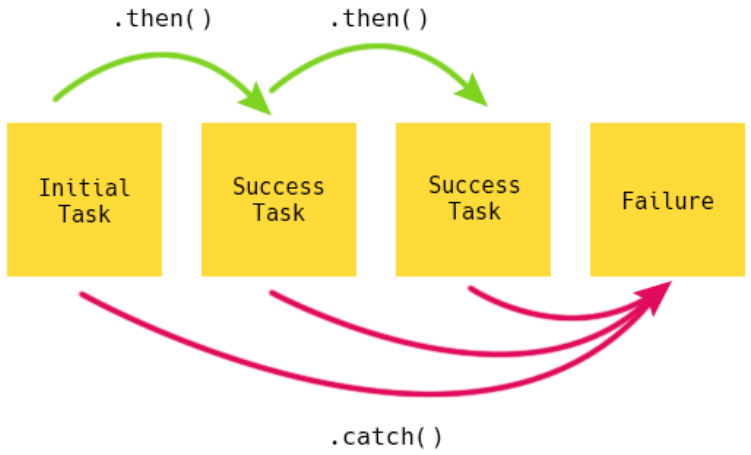
- Depuis peu, permet d'utiliser le mot-clé finally *!*

- Pas « lazy » ;
- Pas annulables de l'extérieur ;
- Et pour alourdir le panier, les promesses sont robustes !

Plusieurs états possibles



www.javascripttutorial.net



▷ Il y a en réalité un 4em état : « Achevé »

Deux entités A et B

```
// Entité A
const promise_alice = new Promise((resolve, reject) => {
  setTimeout(() => { // On simule le retour d'un serveur avec l'attente d'une seconde
    resolve({ // C'est l'objet qu'Alice va fournir à Bob
      fraises: 3,
      chantilly: 1,
      noix_macadamia: 4
    })
  }, 1000);
})
// Entité B
promise_alice.then((sacDeCourse) => {
  console.log("Merci !");
});
// Ce code va être exécuté avant la fin de la promesse :)
console.log("Une promesse n'est pas bloquante !");
```

Avantages

- Permet d’avoir un code bien plus flexible
- Et évidemment, plus agréable à lire :

Cascade de callbacks

Promesses

Promesses chaînées

```
faireLesCoursesPromise()
  .then((sacDeCourse) => { // Coureur 1
    console.log("Il n'y en a pas assez, il faut y retourner...");
    return faireLesCoursesPromise();
  })
  .then((sacDeCourseComplet) => { // Coureur 2
    console.log("Merci, je m'occupe de réaliser la glace");
    return faireLaGlacePromise();
  })
  .then((glace) => { // Coureur 3
    console.log("Je l'envoie au serveur");
    return envoyerAuServeur();
  })
  .then((retourDuServeur) => { // Coureur 4
    console.log("La glace a été correctement enregistrée côté serveur, fin du programme.")
  })
;
```

Robustes !

```
faireLesCoursesPromise()
  .then((sacDeCourse) => {
    console.log("Il n'y en a pas assez, il faut y retourner..."); // Pas affiché
    return faireLesCoursesPromise();
  })
  .then((sacDeCourseComplet) => {
    console.log("Merci, je m'occupe de réaliser la glace"); // Pas affiché
    return faireLaGlacePromise();
  })
  .then((glace) => {
    console.log("Je l'envoie au serveur"); // Pas affiché
    return envoyerAuServeur();
  })
  .then((retourDuServeur) => {
    console.log("La glace a été correctement enregistrée côté serveur, fin du programme."); // Pas affiché
  })
  .catch((err) => {
    console.log("Erreur lors de l'éxecution de la promesse.");
    console.error(err);
  })
;
```

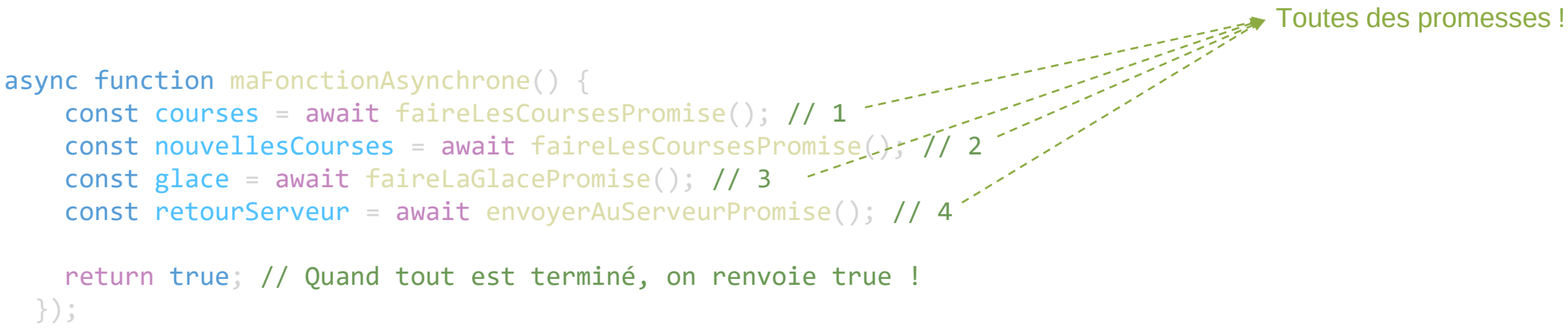
Ecouter plusieurs promesses

- Parfois, il est nécessaire d'écouter plusieurs promesses simultanément
- Pour ce faire, il existe la méthode Promise.all([tabPromises])
- Améliore les performances
- Attention, si une promesse échoue, vous entrez dans le .catch même si toutes les autres ont réussies !

Async / Await, une révolution

- Les promesses ont été simplifiées depuis ES 2017 !

- On rêverait d'une écriture Synchrone avec un comportement Asynchrone

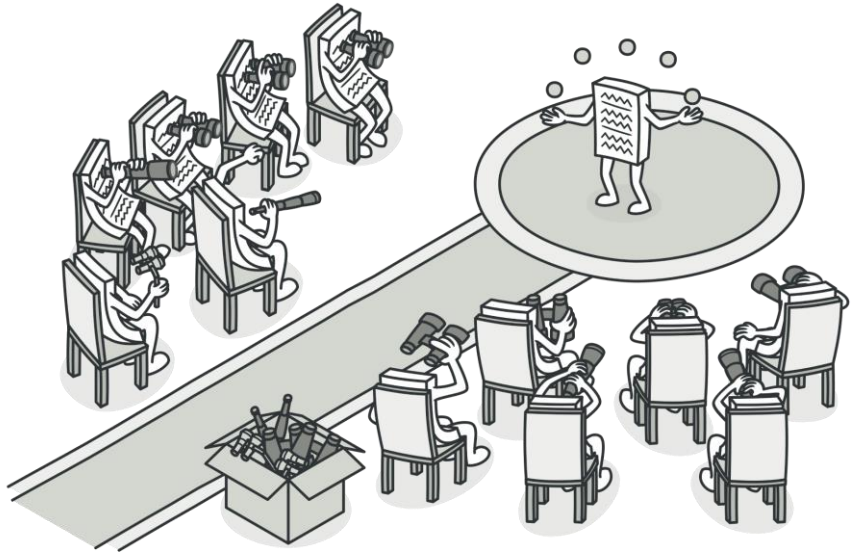



```
async function maFonctionAsynchrone() {
    const courses = await faireLesCoursesPromise(); // 1
    const nouvellesCourses = await faireLesCoursesPromise(); // 2
    const glace = await faireLaGlacePromise(); // 3
    const retourServeur = await envoyerAuServeurPromise(); // 4

    return true; // Quand tout est terminé, on renvoie true !
  });
```

Obserables avec RxJS

Observer / Observable

- Design pattern connu
- Un observable
  - Un ou plusieurs obseravateurs
- Nouveauté Angular !

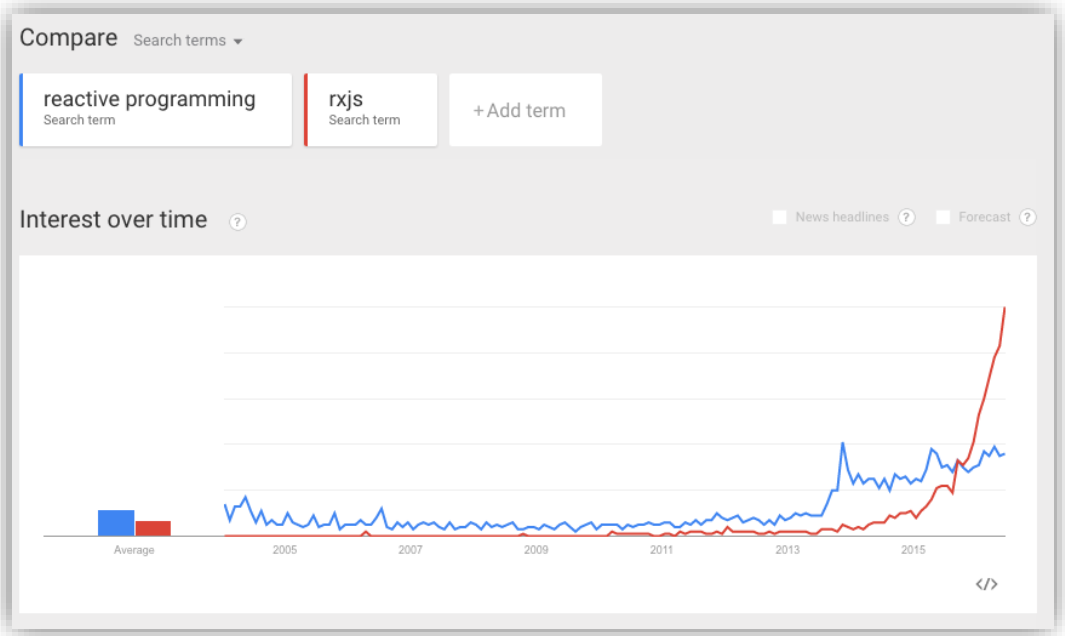


refactoring.guru

- Permet de limiter le couplage entre nos composants
- On parle aussi de « Reactive programming »

RxJS

- En JavaScript vanilla, il n'y a pas encore d'Observable
- La librairie RxJS vient simuler ce comportement
  - Mondialement reconnu



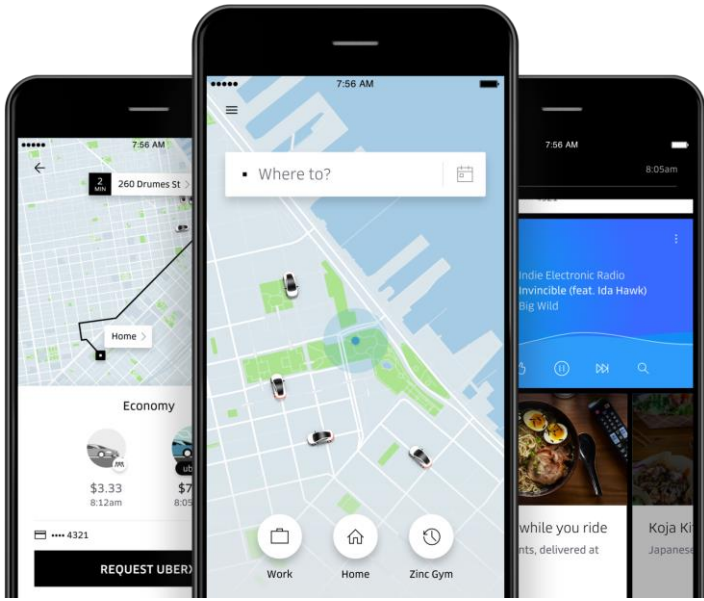
Une histoire de réactivité

- Imaginons une application où il est nécessaire de réagir à une information en temps réel

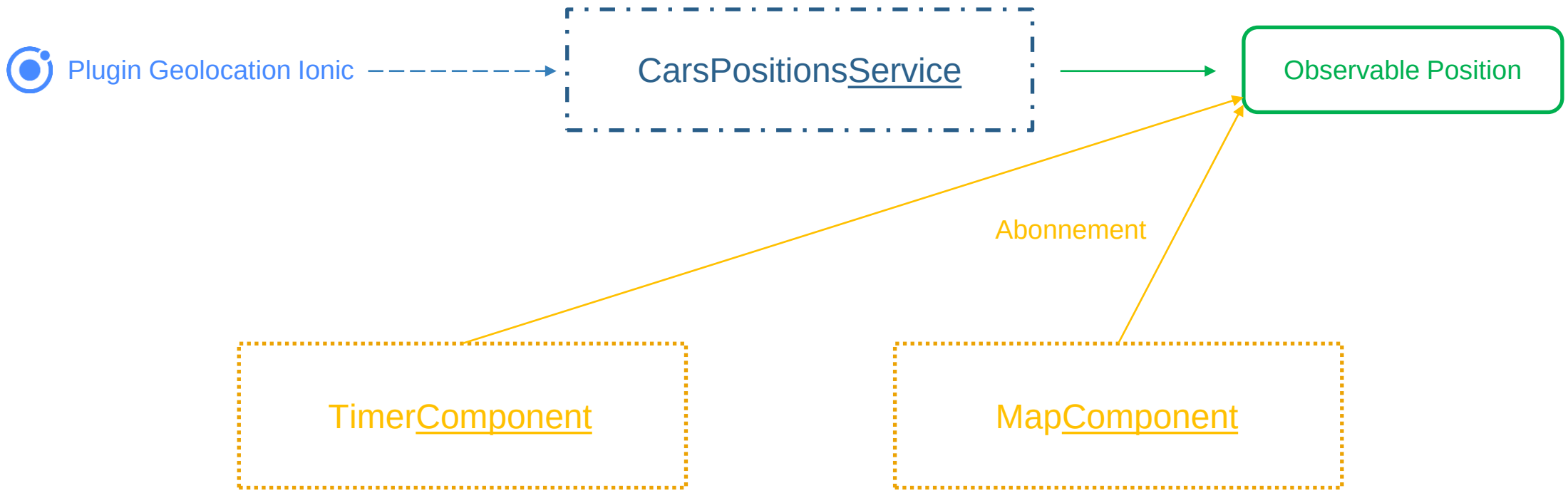
- L'observable, c'est le GPS
  - Renvoie la nouvelle position à chaque déplacement

- On aura 3 observateurs
  1. Mettre à jour la vue ;
  2. Modifier les minutes restantes avant l'arrivée ;
  3. Faire vibrer le téléphone à l'arrivée du chauffeur.

- Chaque observateur est indépendant !



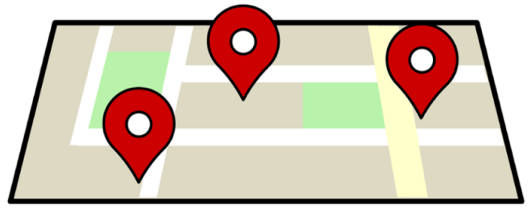
Une architecture classique



Concrètement

Injection du service
Geolocation fournit par Ionic

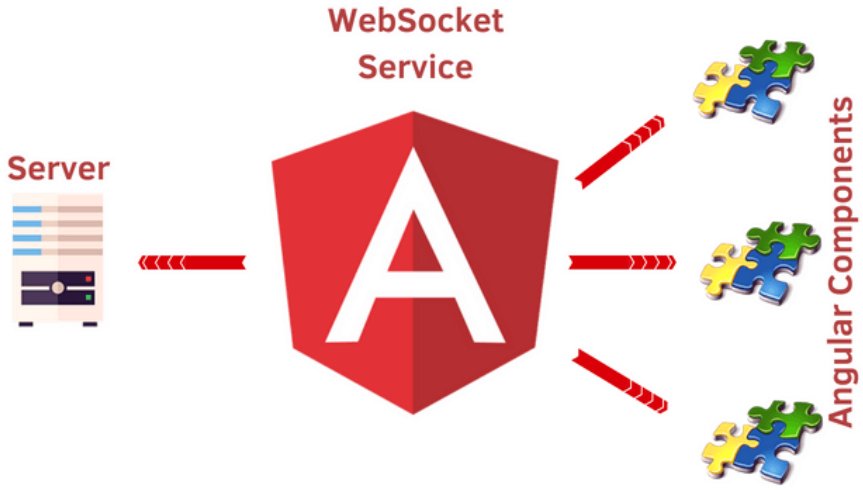
```
import { Geolocation } from '@ionic-native/geolocation/ngx';
...
constructor(private geolocation: Geolocation) {}
...
let watch = this.geolocation.watchPosition(); // OBSERVABLE RECUPERE !
watch.subscribe((data) => {
  // data.coords.latitude
  // data.coords.longitude
});
```



Nos composants

- La plupart des cas, vous récupérerez des observables fournis par les bibliothèques

- Parfois, il faudra créer nos Observables à nous



Création

- Créons un observable qui renverra une nouvelle couleur toutes les 5 secondes pendant 1 minute !

- Pour y arriver, nous profiterons de la classe Observable

https://stackblitz.com/edit/ionic-macademia-first-observable

Création

```
this._data = new Observable<string>(observer => {
    let valueIndex = 0;
    let valueInterval;
    let completeTimeout;

    valueInterval = setInterval(() => {

        observer.next(`Value ${valueIndex}`);
        valueIndex++;

        if (this.throwError) {
            observer.error(new Error('error'));
        }

    }, 1000);

    completeTimeout = setTimeout(() => {
        observer.complete();
    }, 4000);

    /* TEAR DOWN logic. */
    return () => {
        clearTimeout(completeTimeout);
        clearInterval(valueInterval);
    }
});
```

Représente l'observateur

La fonction « Tear down logic » est appelé au moment ou l'observateur reçoit l'instruction « complete » ou de désabonne via « unsubscribe ». Cette fonction vide la mémoire manuellement et permettra d'éviter les fuites mémoire

Création d'un observateur

```
public subscribe() {
    this.finished = false;
    this.success = false;
    this.error = null;
    this.valueList = [];

    this._data
        .finally(() => this.finished = true)
        .retry(3)
        .subscribe(
            value => this.valueList.push(value), // .next
            error => this.error = error, // .error
            () => this.success = true // .complete
        );
}
```

Les observables en Angular

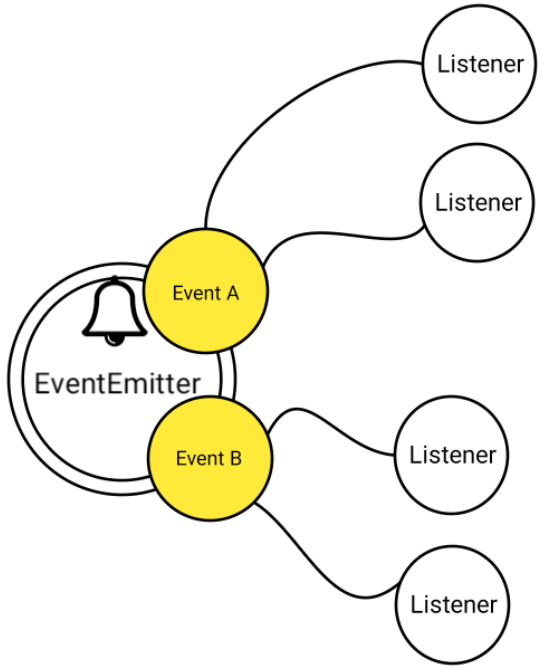
- Sans le savoir, vous avez déjà utilisé les observables…

- Avec les Event Emitter
  - new EventEmitter => Observable
  - (event) => Observateur

- Avec les évènements de formulaire
  - Controls

- HttpClient

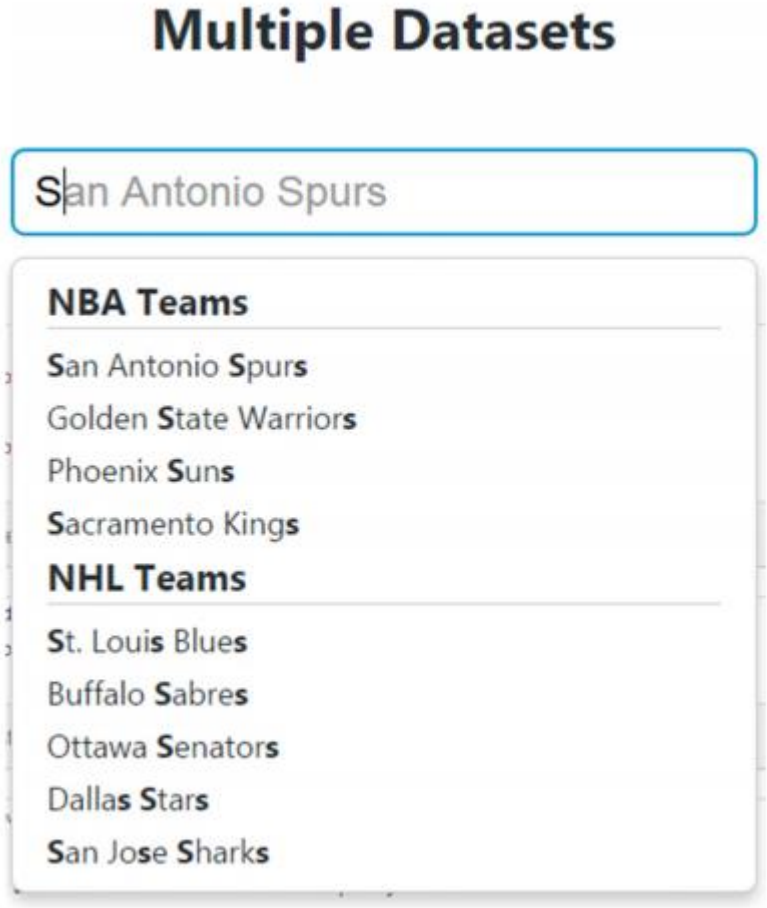
etc.



La puissance de RxJS

- L'outil est devenu un incontournable pour tout projet qui se respecte
  - Il y a une raison à cela
- Fournit pléthore de méthode pour nous faciliter la vie
  - https://github.com/Reactive-Extensions/RxJS/blob/master/doc/gettingstarted/categories.md
- Il faut imaginer un observable comme un tunnel par lequel transite des tableaux de données

Les opérateurs RxJS



1. Ne pas faire une recherche bêtement à chaque fois que l'utilisateur appuie sur une touche
2. Ne pas requêter le serveur si la requête a déjà été faite précédemment
3. Ne pas lancer plusieurs requêtes serveur en simultané

```
let term = new FormControl();

term.valueChanges
    .filter(query => query.length >= 3)
    .debounceTime(400)
    .distinctUntilChanged()
    .switchMap(value => autocompleteService.search(value).catch(error => Observable.of([])))
    .subscribe(results => theResults = results);
```

C'est tout.

Quelques fonctions utiles

of

- Permet de créer très rapidement un observable renvoyant la valeur (ou les valeurs) passé en paramètre

```
import { of, from } from "rxjs";

ngOnInit() {
  of("Ma valeur").subscribe(res => {
    console.log(res); // Ma valeur
  });
}
```

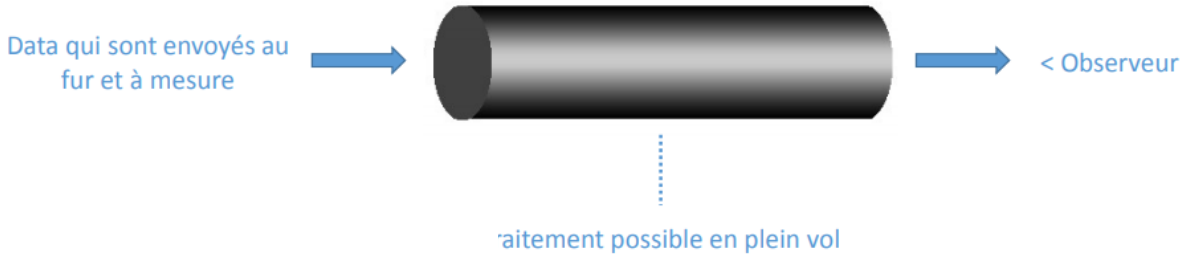
from

- Permet également de créer un Obervable, mais à partir d'un tableau !

```
from(["Ma valeur 1", "Ma valeur 2"]).subscribe(res => {
  console.log(res); // Ma valeur 1 PUIS Ma valeur 2
});
```

Chaîner les appels

▷ pipe va nous permettre de chainer différentes fonctions pour appliquer des traitements sur la donnée retournée




```
import { map, tap } from "rxjs/operators";


ngOnInit() {
  from(["123", "1234"])
    .pipe(
      map(value => value.length),
      map(value => value * value),
      // Le tap ne sert à rien, permet simplement de faire un appel intermédiaire
      tap(value => console.log(value)) // Affichera 9 PUIS 16
    )
    .subscribe();
}
```

Ecouter plusieurs observables

- forkJoin va vous permettre de réagir à plusieurs observables
  - Similaire à Promise.all

Tester ce code

```
const example = forkJoin({
  //emit 'Hello' immediately
  sourceOne: of('Hello'),
  //emit 'World' after 1 second
  sourceTwo: of('World').pipe(delay(1000)),
  //emit 0 after 1 second
  sourceThree: interval(1000).pipe(take(1)),
  //emit 0...1 in 1 second interval
  sourceFour: interval(1000).pipe(take(2)),
  //promise that resolves to 'Promise Resolved' after 5 seconds
  sourceFive: myPromise('RESULT')
});

// { sourceOne: "Hello", sourceTwo: "World", sourceThree: 0, sourceFour: 1,sourceFive: "Promise Resolved: RESULT"}
const subscribe = example.subscribe(val => console.log(val));
```

Ecouter plusieurs observables – 2 !

- forkJoin attend que tous les observables soient terminés d'abord
- combineLatest lui réagira à chaque tick de chacun des observables !

```
const arrayAllInputs = [
  this.globalForm.get('nutritionalProgram').get('program').valueChanges.pipe(startWith()),
  this.rationsForm.get('k1').valueChanges.pipe(startWith(this.rationsForm.get('k1').value)),
  this.rationsForm.get('k2').valueChanges.pipe(startWith(this.rationsForm.get('k2').value)),
  this.rationsForm.get('k3').valueChanges.pipe(startWith(this.rationsForm.get('k3').value)),
];
combineLatest(arrayAllInputs).subscribe(([program, k1, k2, k3]) => {
  const result = k1 * k2 * k3 * k4 * k5;
  this.rationsForm.get('resultCoeff').setValue(+result.toFixed(2));
  this.evaluateRations();
});
```

Cold et Hot Observables

https://stackblitz.com/edit/angular-observables-create-cancel-hot

HOT !

```
this._hotData = this._data.publish(); // La conversion en HOT Observable !
```

```
this.hotDataSubscription = this._hotData.connect(); // On lance l'observable
```

```
// Enfin, on crée un observateur
this.subscription = this._hotData.subscribe(value => this.valueList.push(value));
```

Comprendre le Marble Diagram

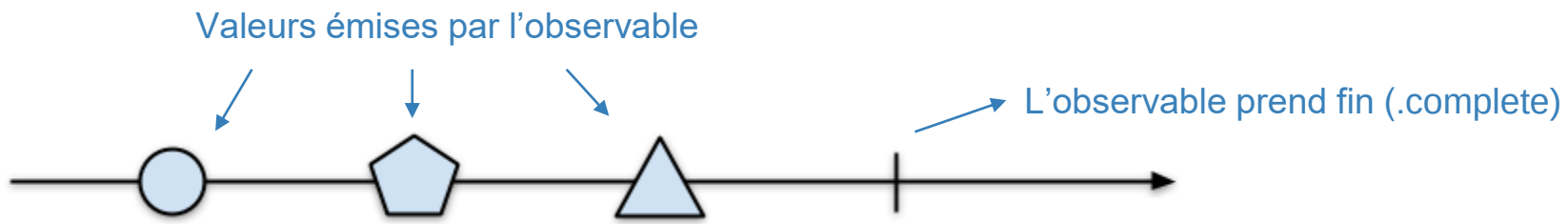
- Il existe énormément de fonctions fournies par RxJS
- Une documentation excellente !
  - … A condition de la comprendre

- Etudions les diagrammes les plus populaires
- Essayez de deviner l'utilité des Observables

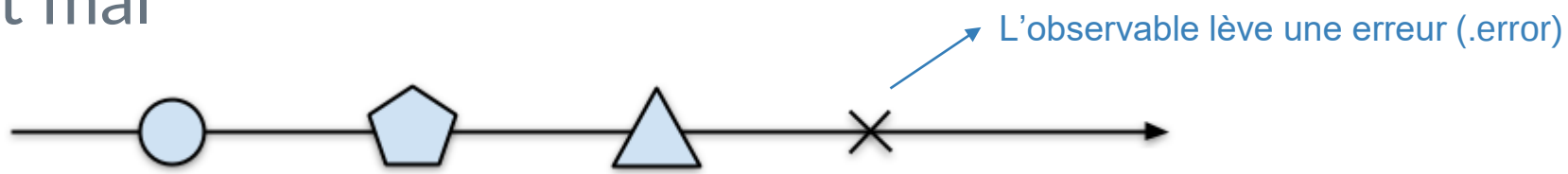
Commençons par la base

- Tous les observables ont un début ..

- Et une fin

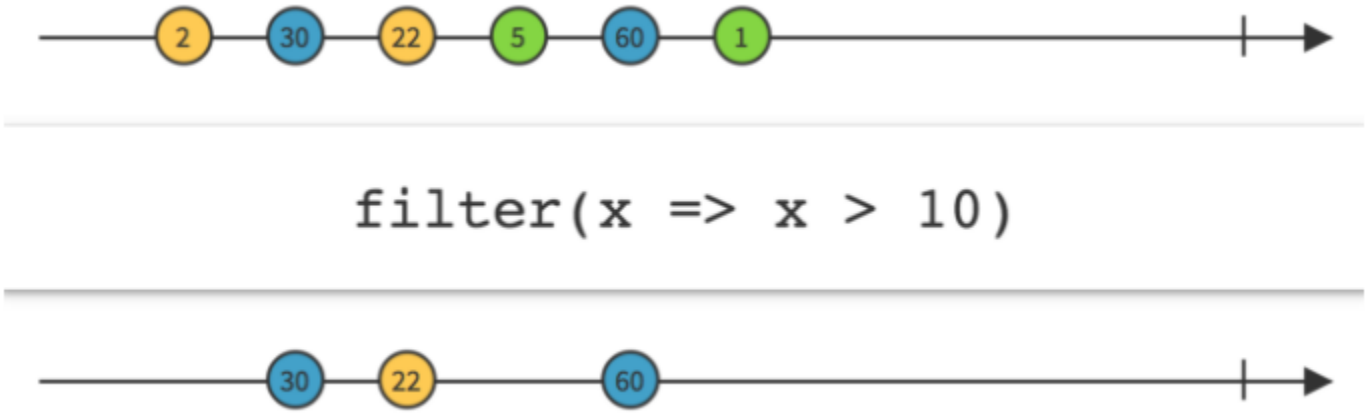


- Parfois, ça finit mal



- Et parfois, pas du tout :

Premier opérateur : filter

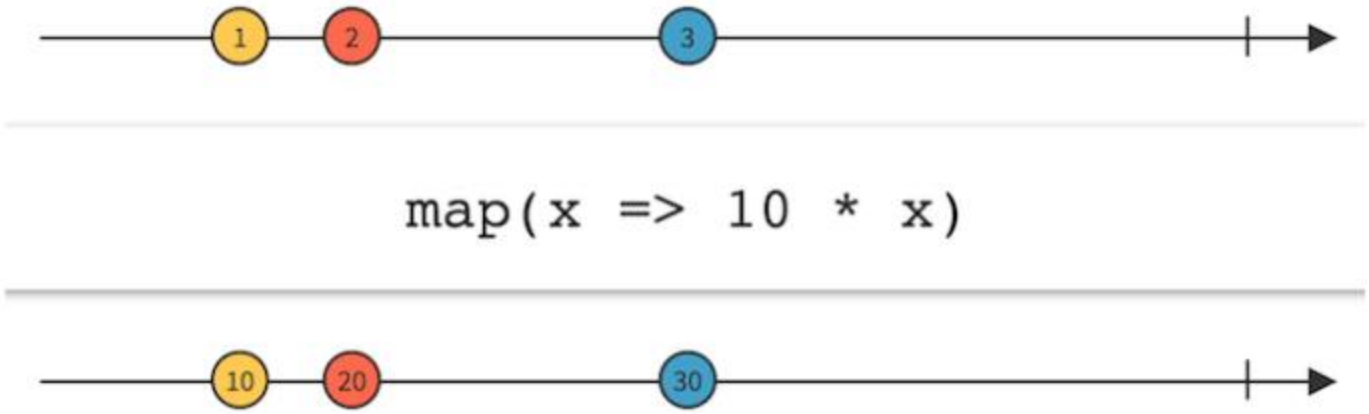


- filter prend une fonction en paramètre
- Renvoie uniquement les éléments dont la fonction renvoie true

Filter() en Angular / RxJS

```
import { from } from 'rxjs';
import { filter } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  template: `Hello tout le monde`
})
export class AppComponent implements OnInit{
  ngOnInit(): void {
    from([1, 2, 3, 4, 5, 6, 7, 8])
      .pipe(filter(x => x % 2 === 0))
      .subscribe(console.log); // 2 - 4 - 6 - 8
  }
}
```

Opérateur map

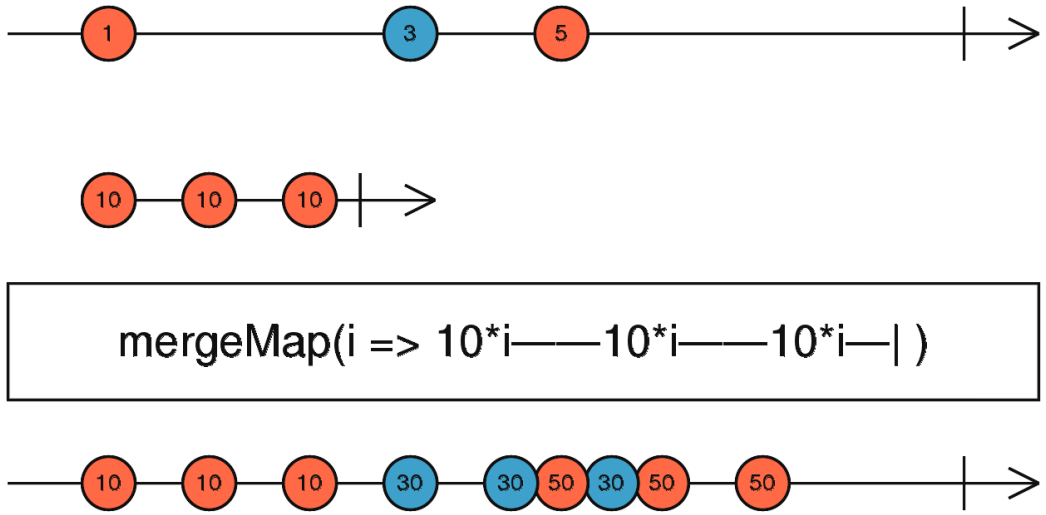


- Transforme chaque élément renvoyé par l'observable
- Renvoie la valeur transformée

Map() en Angular / RxJS

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

interface IUser {
  name: string;
  sexe: 'male' | 'female';
}

@Component({
  selector: 'app-root',
  template: `
    <p *ngFor="let name of names"> Bonjour {{ name }} </p>
  `
})
export class AppComponent implements OnInit {
  public names: string[];

  ngOnInit(): void {
    this.names = [];
    const users: IUser[] = [
        { name: 'Gertrude',  sexe: 'female' },
        { name: 'Jacques', sexe: 'male' },
        { name: 'Salim', sexe: 'male' }
    ];
    from(users)
      .pipe(map(user => user.sexe === 'male' ? `M. ${user.name}` : `Mme ${user.name}`))
      .subscribe((name) => this.names.push(name));
  }
}
```

Opérateur mergeMap



- Transforme chaque élément renvoyé un observable
- Puis renvoie une tableau de donnée complètement aplati !
- L'observable renvoyé peut à son tour renvoyer plusieurs valeurs

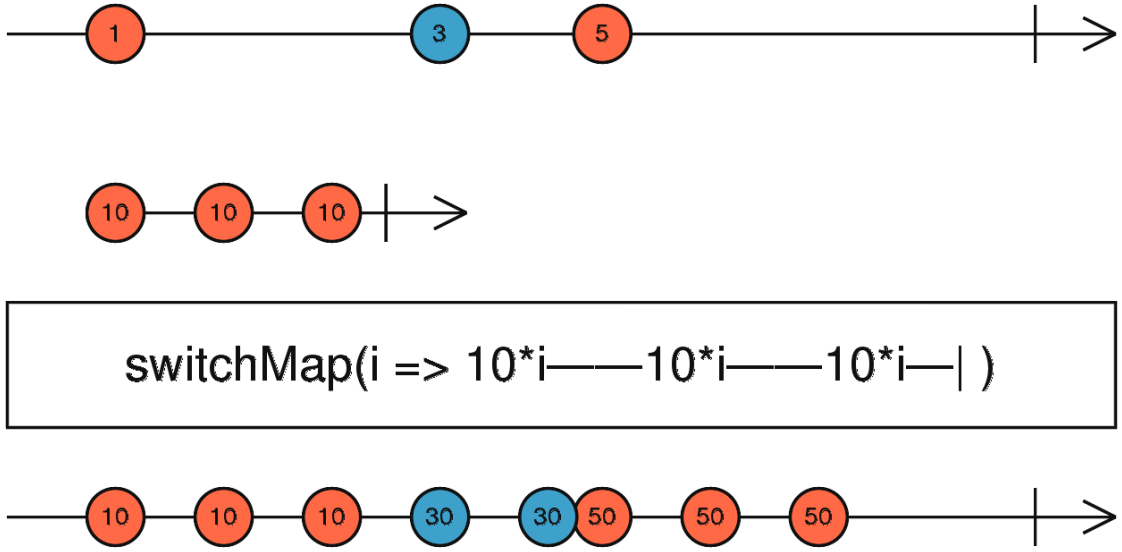
Marble diagram

MergeMap() en Angular / RxJS

```
@Component({
  selector: 'app-root',
  template: `
    <button #buttonConfirm>Je confirme ma réservation</button>
  `
})
export class AppComponent implements OnInit {
  @ViewChild('buttonConfirm', {static: true}) buttonConfirmation: ElementRef;

  public constructor(private tripRecommendor: TripRecommendorService) {
  }

  ngOnInit(): void {
    // Au click ->
    // Réservation de l'hôtel terminée
    // Réservation de la voiture OK
    // Réservation du vol effectuée
    fromEvent(this.buttonConfirmation.nativeElement, 'click')
      .pipe(
        mergeMap(this.tripRecommendor.bookHotel),
        mergeMap(this.tripRecommendor.bookCar),
        mergeMap(this.tripRecommendor.bookFly),
      )
      .subscribe()
    ;
  }
}
```

```
@Injectable({
  providedIn: 'root'
})
export class TripRecommendorService {

  constructor() {
  }

  public bookFly(value): Observable<string> {
    return of('Réservation du vol effectuée')
      .pipe(delay(1000), tap(console.log));
  }

  public bookHotel(value): Observable<string> {
    return of('Réservation de l\'hôtel terminée')
      .pipe(delay(600), tap(console.log));
  }

  public bookCar(value): Observable<string> {
    return of('Réservation de la voiture OK')
      .pipe(delay(400), tap(console.log));
  }
}
```

Opérateur switchMap



- Transforme chaque élément renvoyé par l'observable
- Renvoie un observable !
- L'observable renvoyé peut à son tour renvoyer plusieurs valeurs

Map, SwitchMap et MergeMap, quelle différence ?

- Map : Projection simple de données
- Vous obtiendrez le même nombre de sorties que d'entrées
- Utile pour transformer une information renvoyé par le serveur
  - Pour gagner du temps pour les prochains traitements par exemple

```
const names = callServerToGetAllUsers()
            .pipe(
                map(user => `${user.firstname} ${user.lastname.toUpperCase()}`)
            );
```

Map, SwitchMap et MergeMap, quelle différence ?

▷ MergeMap : Les éléments en entrée seront utilisés pour renvoyer des observables et non des valeurs primitives

▷ Attention, l'ordre de sortie dépend des messages envoyés par l'observable et non pas de l'ordre de définition

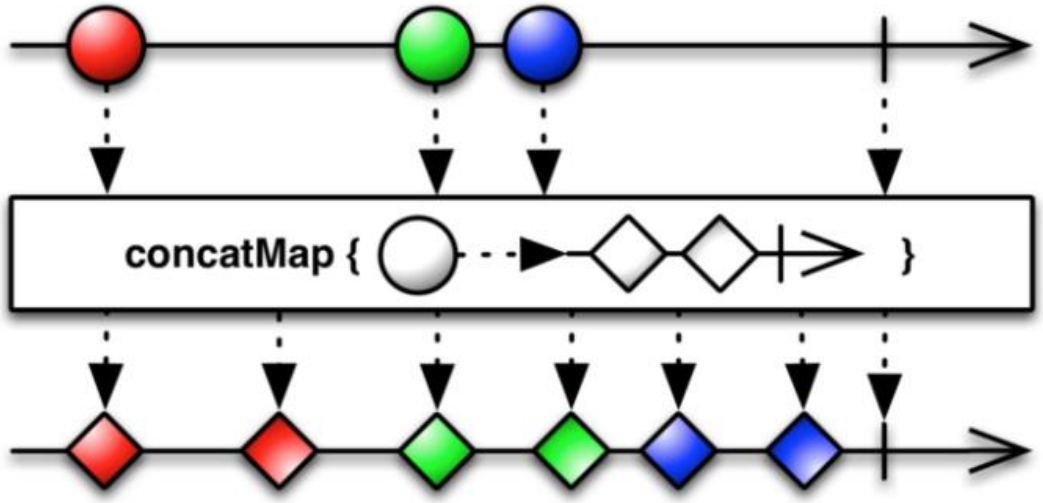
Map, SwitchMap et FlatMap, quelle différence ?

- **SwitchMap** : Comparable à FlatMap
  - Il est important de noter qu'un nouveau tick d'Observable annule tous les observables créés précédemment !
  - L'ordre de sortie dépend de l'ordre de définition

- Renvoie uniquement le résultat du dernier observable
  - Très pratique pour chaîner des Observables inter-dépendants

```
fromEvent(document, 'click')
  .pipe(
    // Recommence à chaque click !
    // First click: 0, 1, 2...
    // Au deuxième click, l'interval précédent est annulé et on repart : 0, 1, 2...
    switchMap(() => interval(1000))
  )
  .subscribe(console.log);
```

Opérateur concatMap

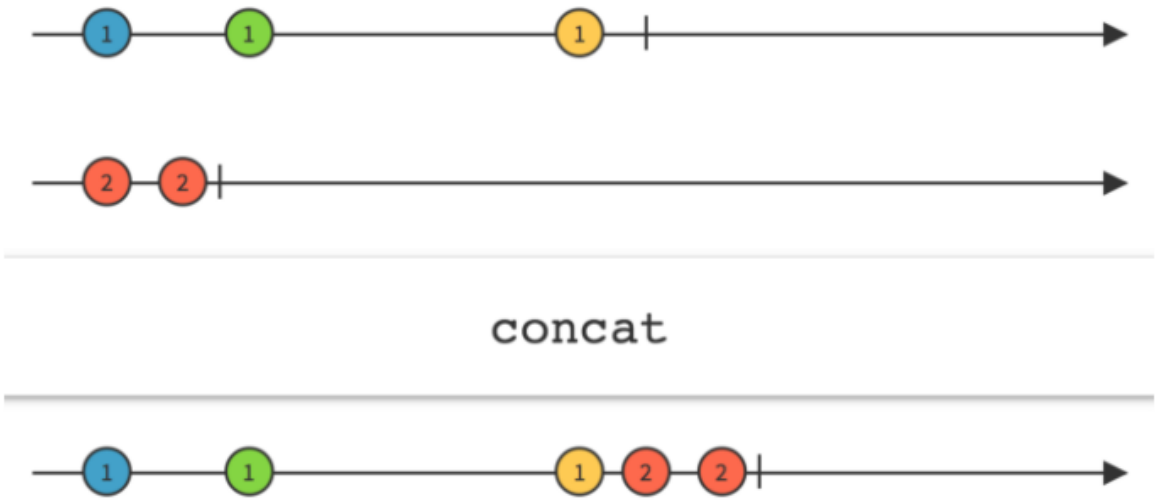


- Comme FlatMap, mais renvoie les données une par une
- Attend l'observable précédent avant de passer au suivant
- Garanti l'ordre

ConcatMap() en Angular / RxJS

```
interface ICarInfos {
  name: string;
  power: number;
}

@Component({
  selector: 'app-root',
  template: `Yo`
})
export class AppComponent implements OnInit {
  public constructor() {
  }

  ngOnInit(): void {
    of('Peugeot 208', 'Citroen')
      .pipe(
        concatMap(
          value => this.getCarsInfos(value)
        )
      )
      .subscribe(carInfos => console.log(carInfos));
    // Après 1 seconde : {name: "Peugeot 208", power: 442}
    // Puis 1 seconde plus tard : {name: "Citroen", power: 71}
  }

  private getCarsInfos(carName): Observable<ICarInfos> {
    // TODO Effectuer un appel au serveur
    return of({name: carName, power: Math.floor(Math.random() * 1000)})
      .pipe(delay(1000));
  }
}
```

Opérateur concat



▷ Comme ConcatMap mais renvoie des valeurs primitives !

Marble diagram

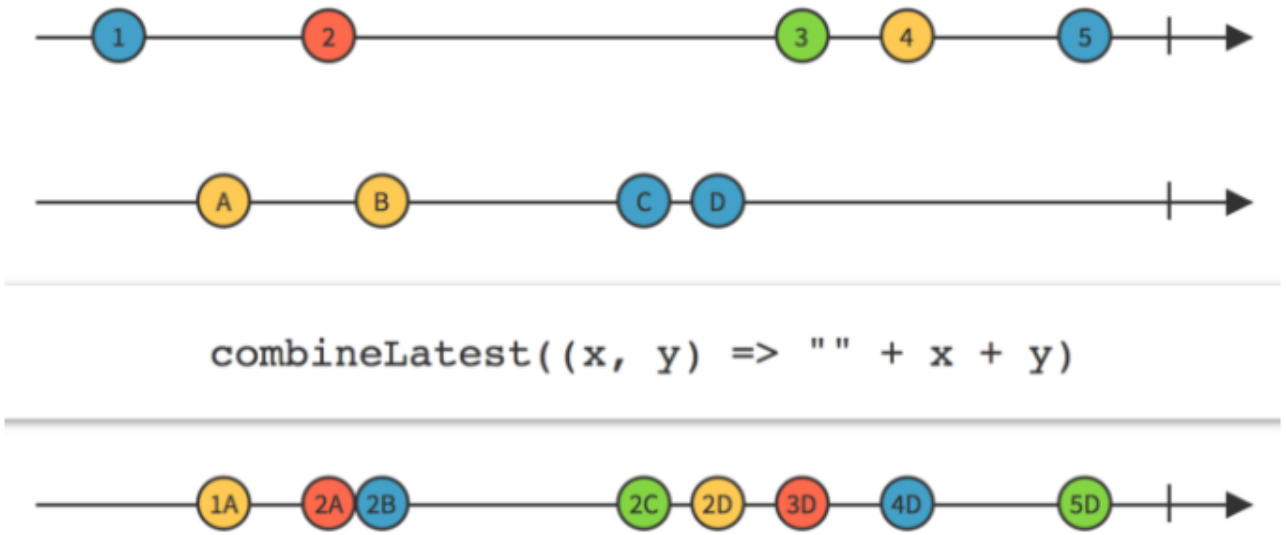
Concat() en Angular / RxJS

```
import { concat, of } from 'rxjs';

ngOnInit(): void {
  concat(
    of(1, 2, 3),
    // subscribed after first completes
    of(4, 5, 6),
    // subscribed after second completes
    of(7, 8, 9)
  )
  // log: 1, 2, 3, 4, 5, 6, 7, 8, 9
    .subscribe(console.log);
}
```

Lien vers la documentation

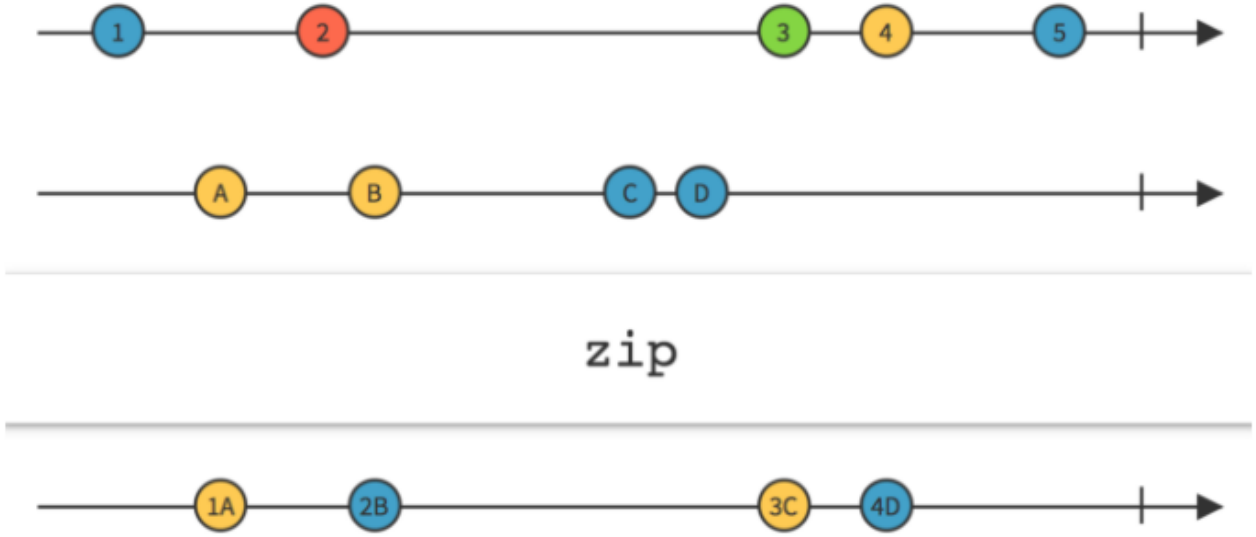
Marble diagram

Opérateur combineLatest



- Combine vos observables
- Renvoie un tableau avec les dernières valeurs émises
  - A chaque tick d'un observable !

Opérateur zip



- Combinaison d'observable comme combineLatest
- Ne réagit QUE quand les 2 observables ont émis une valeur

Zip() en Angular / RxJS

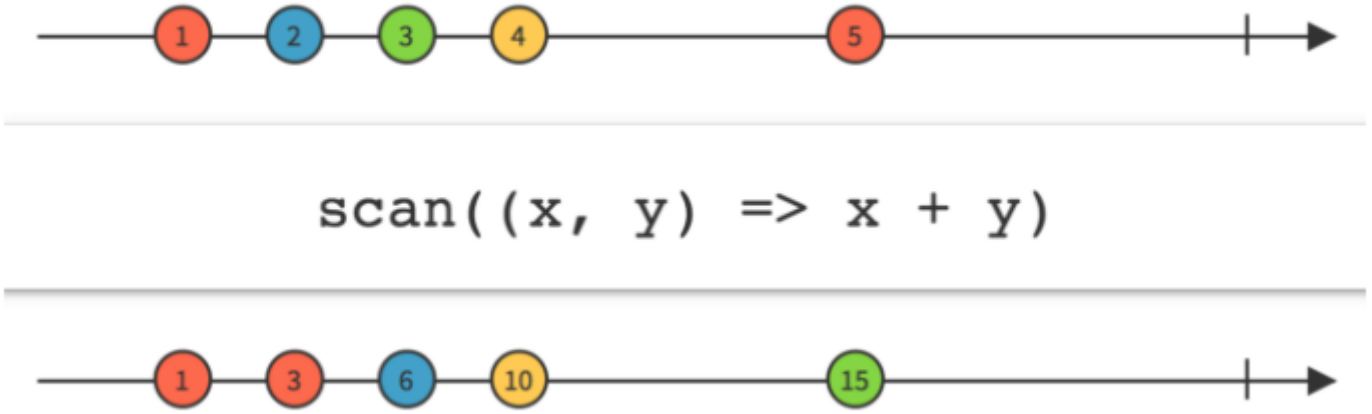
```
import { Component, OnInit } from '@angular/core';
import { RobotCurieuxService } from './shared/services/robot-curieux.service';
import { zip } from 'rxjs';

@Component({
  selector: 'app-root',
  template: `
    <h1>Apprenez quelque chose à notre robot !</h1>

    <ng-container *ngIf="learnProperty">
      <label for="property">Nouvelle propriété : </label>
      <input id="property" [(ngModel)]="property" type="text"/>

      <button (click)="validateProperty()">Valider</button>
    </ng-container>

    <ng-container *ngIf="!learnProperty">
      <label for="value">Nouvelle valeur : </label>
      <input id="value" [(ngModel)]="value" type="text"/>

      <button (click)="validateValue()">Valider</button>
    </ng-container>
  `
})
```

```
export class AppComponent implements OnInit {
  public learnProperty: boolean;
  public property: string;
  public value: string;

  public constructor(private robotCurieux: RobotCurieuxService) {
  }

  public validateProperty(): void {
    this.robotCurieux.$newValue.next(this.property);
    this.property = '';
    this.learnProperty = !this.learnProperty;
  }

  public validateValue(): void {
    this.robotCurieux.$newProperty.next(this.value);
    this.value = '';
    this.learnProperty = !this.learnProperty;
  }

  ngOnInit(): void {
    this.learnProperty = false;

    zip(this.robotCurieux.$newProperty, this.robotCurieux.$newValue)
      .subscribe(([prop, value]) => {
        console.log(`Petit robot a apprit que ${prop}, c'est ${value}`);
      });
  }
}
```

Zip() en Angular / RxJS

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class RobotCurieuxService {
  public $newProperty: Subject<string>;
  public $newValue: Subject<string>;

  constructor() {
    this.$newProperty = new Subject();
    this.$newValue = new Subject();
  }
}
```

Opérateur scan



- Applique une fonction à chacun des éléments
- Prend en paramètre la valeur renvoyée précédemment

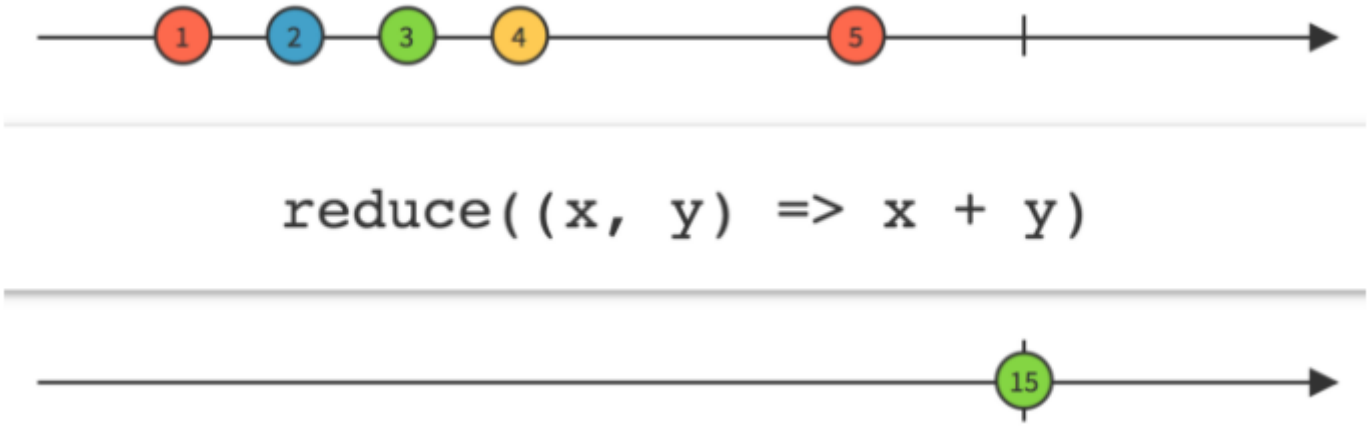
Marble diagram

Scan() en Angular / RxJS

```
import { Component, OnInit } from '@angular/core';
import { Subject } from 'rxjs';
import { scan } from 'rxjs/operators';

@Component({
  selector: 'app-root',
  template: `Le silence est d'or`
})
export class AppComponent implements OnInit {
  public constructor() {
  }

  ngOnInit(): void {
    // RxJS v6+
    const subject = new Subject();
    // scan example building an object over time
    const example = subject.pipe(
      scan((acc: any, curr: any) => ({acc, ...curr}))
    );
    // log accumulated values
    const subscribe = example.subscribe(val =>
      console.log('Accumulated object:', val)
    );
    // next values into subject, adding properties to object
    // {name: 'Joe'}
    subject.next({name: 'Joe'});
    // {name: 'Joe', age: 30}
    subject.next({age: 30});
    // {name: 'Joe', age: 30, favoriteLanguage: 'JavaScript'}
    subject.next({favoriteLanguage: 'JavaScript'});
  }
}
```

Exemple inspiré d'ici

Opérateur reduce



▷ Renvoie une et unique valeur à partir de toutes les valeurs précédentes

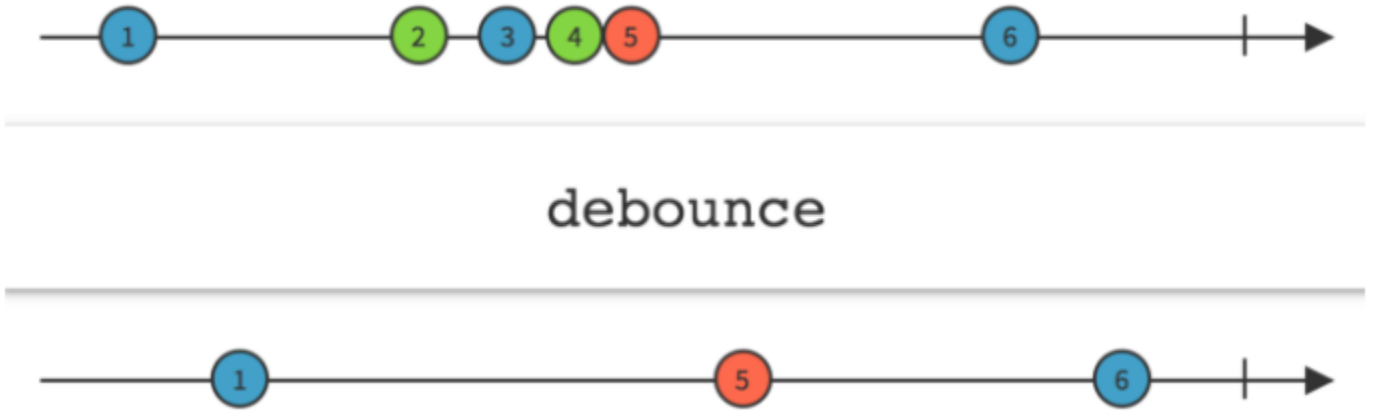
Reduce() en Angular / RxJS

```ts
import { Component, OnInit } from '@angular/core';
import { reduce } from 'rxjs/operators';
import { FoodsService, IFood } from './shared/services/foods.service';

@Component({
  selector: 'app-root',
  template: `Le silence est d'or`
})
export class AppComponent implements OnInit {
  public constructor(private foods: FoodsService) {
  }

  ngOnInit(): void {
    this.foods.getAllReceips()
      .pipe(
        reduce((price: number, recipe: IFood) => {
          return price + recipe.ingredients.reduce((acc, ing) => ing.price + acc, 0);
        }, 0)
      )
      .subscribe(totalPrice => console.log(`Au total, ça va nous coûter ... ${totalPrice} € !`));
    // Au total, ça va nous coûter ... 7.5 € !
  }
}
```

```ts
export interface IIngredient {
  name: string;
  price: number;
}

export interface IFood {
  title: string;
  ingredients: IIngredient[];
}

@Injectable({
  providedIn: 'root'
})
export class FoodsService {

  constructor() {
  }

  public getAllReceips(): Observable<any> {
    return from(MOCK_FOODS).pipe(delay(1000));
  }
}
```

Marble diagram

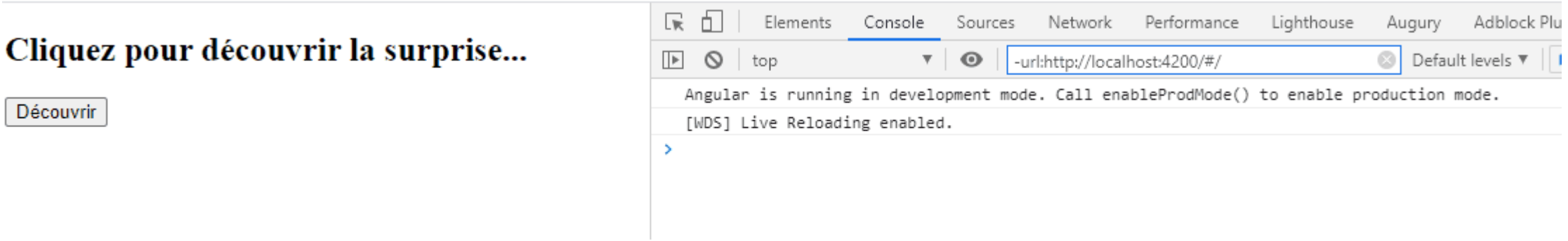
Opérateur debounce



▷ Emet un élément à partir d'un observable uniquement si un intervalle de temps particulier s'est écoulé sans qu'un autre élément ait été émit avant.

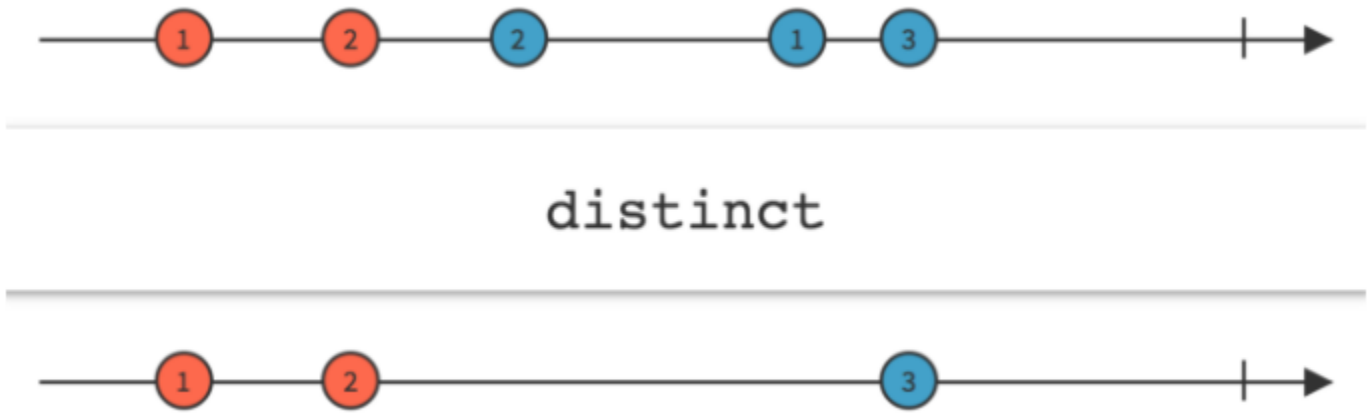
▷ Plus complexe que debounceTime si temps variable

Debounce() en Angular / RxJS




```
@Component({
  selector: 'app-root',
  template: `
    <h2>Cliquez pour découvrir la surprise...</h2>

    <button #buttonDiscover timeToWait="1000">Découvrir</button>
  `
})
export class AppComponent implements OnInit {
  @ViewChild('buttonDiscover', {static: true}) private button: ElementRef;

  public constructor() {
  }

  ngOnInit(): void {
    fromEvent(this.button.nativeElement, 'click')
      .pipe(debounce((el) => timer(+this.button.nativeElement.getAttribute('timeToWait'))))
      .subscribe(() => console.log('Bon anniversaire !'));
  }
}
```

Opérateur distinct



- Retire les éléments dupliqués

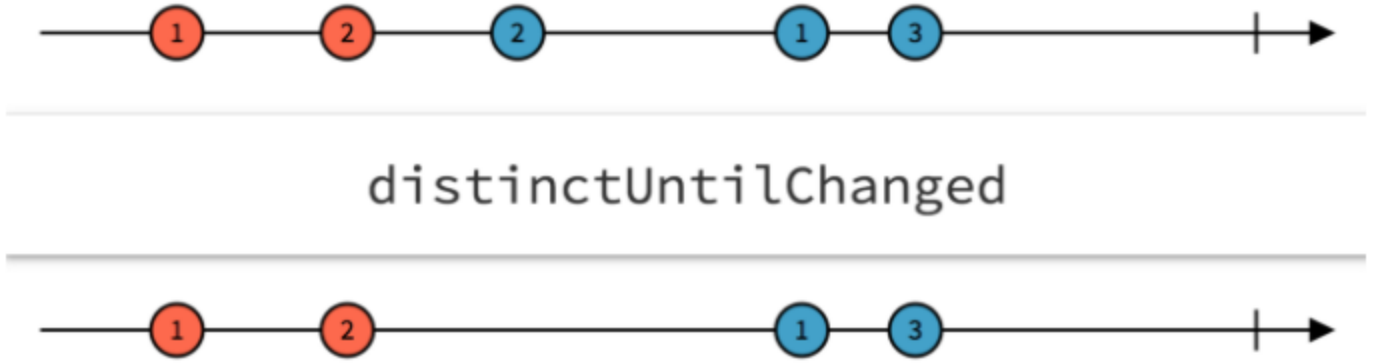
Distinct() en Angular / RxJS

```
import { from } from 'rxjs';
import { distinct } from 'rxjs/operators';

const obj1 = { id: 3, name: 'name 1' };
const obj2 = { id: 4, name: 'name 2' };
const obj3 = { id: 3, name: 'name 3' };
const vals = [obj1, obj2, obj3];

from(vals)
  .pipe(distinct(e => e.id))
  .subscribe(console.log);
```



```
ngOnInit(): void {
  interval(1000)
    .pipe(
      map(val => val % 2)
    )
    .pipe(distinct())
    .subscribe(console.log); // 0 .. 1
}
```

Opérateur distinctUntilChange

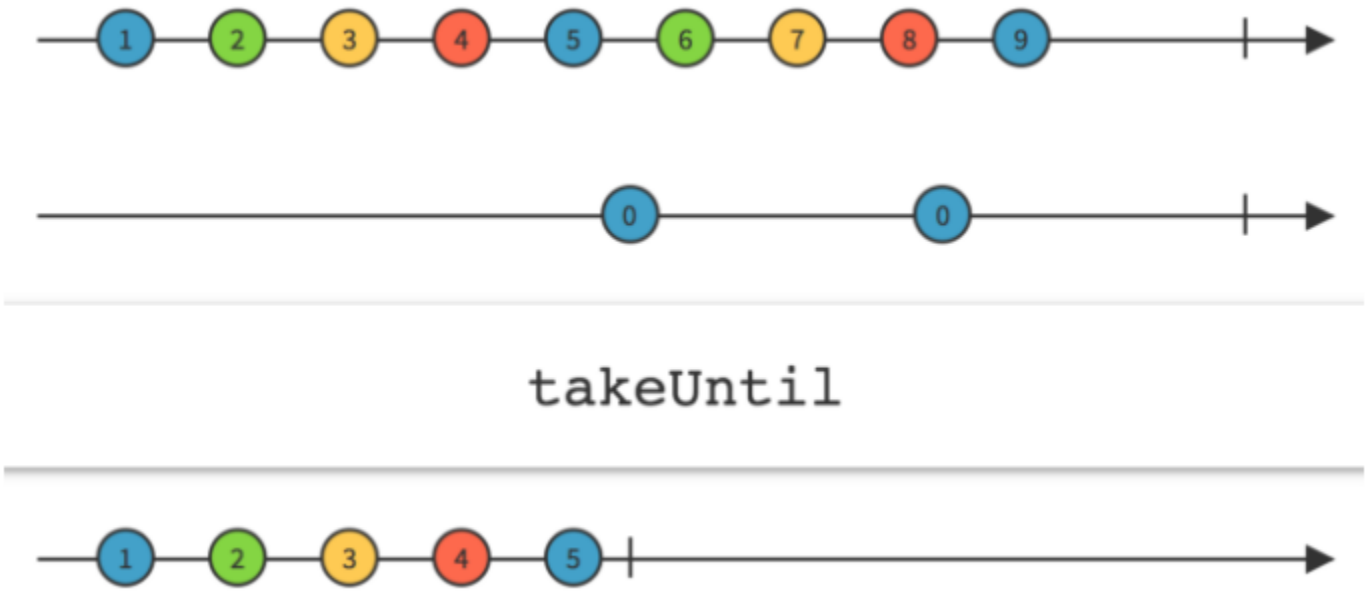


▷ L'élément doit obligatoirement être différent que le dernier renvoyé

DistinctUntilChange() en Angular / RxJS

```
this.tmpControl.valueChanges
  .pipe(
    filter((values: any) => values && values.length >= 3),
    distinctUntilChanged(),
    debounceTime(500)
  )
  .subscribe((values: any) => {
    this.loading = true;
    this.control.setValue({
      location: {
        description: values,
        placeId: null
      }
    });

    this.triggerSearch(values);
  });
```

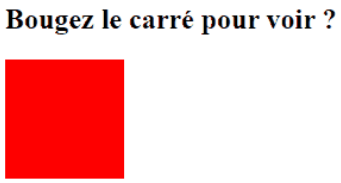
Opérateur takeUntil



▷ Applique un écouteur sur le premier observable et l'arrête dès que le deuxième commence à émettre des données

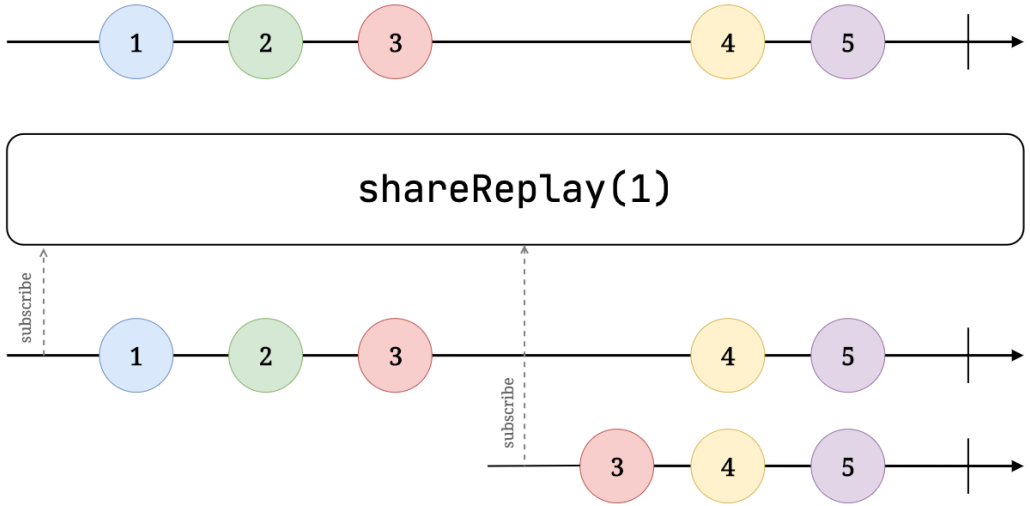
TakeUntil() en Angular / RxJS

```
...

  ngOnInit(): void {
    const $mousedown = fromEvent(this.square.nativeElement, 'mousedown');
    const $mouseup = fromEvent(this.square.nativeElement, 'mouseup');
    const $mousemove = fromEvent(this.square.nativeElement, 'mousemove');

    $mousedown
      .pipe(
        mergeMap(() => $mousemove.pipe(
          map((e: any) => ({
            x: e.clientX,
            y: e.clientY
          }))
        )),
        takeUntil($mouseup)
      )
      .subscribe(data => {
        this.square.nativeElement.style.top = `${data.y}px`;
        this.square.nativeElement.style.left = `${data.x}px`;
      })
    ;
  }
}
```



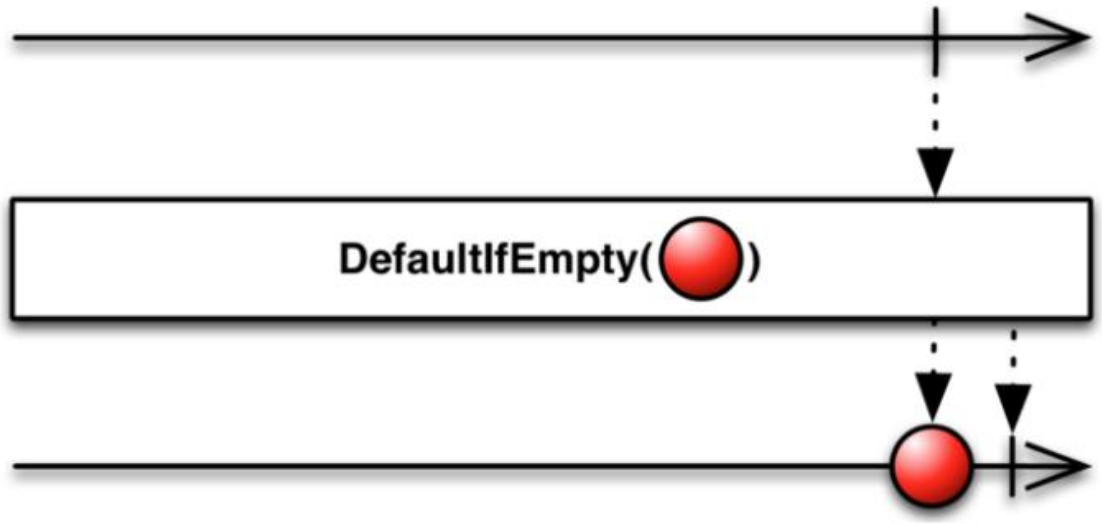
Marble diagram

Opérateur ShareReplay



- Permets de rejouer des éléments précédemment émis
- Puis s'abonne comme à un observable normal

Opérateur defaultEmpty



▷ Émet des éléments à partir de la source Observable, ou un élément par défaut si la source se termine sans émettre d'éléments

Marble diagram

DefaultEmpty() en Angular / RxJS

```
import { defaultIfEmpty } from 'rxjs/operators';
import { of } from 'rxjs';

//emit 'Observable.of() Empty!' when empty, else any values from source
const exampleOne = of().pipe(defaultIfEmpty('Observable.of() Empty!'));

//output: 'Observable.of() Empty!'
const subscribe = exampleOne.subscribe(val => console.log(val));
```

Lien vers la documentation

Questions