

# NG-RXJS: OBSERVABLES

INF1013, Hiver 2022

DMI, UQTR

# Plan

- Observables
- Opérateurs
- ClientHttp (Get)
- Variante d'observables: Les Subjects

# Observables et Reactive-Programming

- Les observables font partie d'une philosophie de codage appelée réactive-programming (RP).
- L'idée consiste à voir le code sous forme de flux de données plutôt que de suite d'instructions. Le RP facilite le traitement parallèle, améliore la lisibilité et masque une partie de la complexité cyclomatique mais induit une nouvelle forme de complexité liée à la synchronisation des flux.
- Cette philosophie est particulièrement adaptée aux problématiques fortement asynchrones.
- Les observables sont une généralisation du pattern observateur.
- Plusieurs langages de programmation ont des bibliothèques de réactive-programming, nous utiliserons le RxJS pour le JavaScript/TS

# Observables

- Les Observables permettent le passage de message entre les ***publishers*** les ***subscribers*** dans une application.
- Les Observables sont particulièrement adaptés aux environnements asynchrones.
- Les Observables sont déclaratifs
  - on définit des fonctions ou **lamdas** à exécuter après le ***publishing***.
  - Elles ne sont pas exécutées s'il n'y a pas d'observateur souscrit.
- Les souscripteurs reçoivent les notifications jusqu'à la fin de la fonction ou leur des-inscription.

# Observable: Structure des Observateurs

- NG définit les observateurs des observables comme un triplet de fonctions (lambda) Handlers dont une est obligatoire.

TYPE DE NOTIFICATION (LAMBDA)	DESCRIPTIONS
<b>next</b>	Obligatoire: Handler pour chaque valeur émise par le publisher. Appelé zéro ou plusieurs fois après le début.
<b>error</b>	Optionnel: Handler pour la notification d'erreur. Une erreur arrête l'exécution de l'observable.
<b>complete</b>	Optionnel: Handler de complétion. Les valeur retardées peuvent quand même être émises après complétions.

# Construction d'un Observable

- Appel au constructeur d'Observable qui reçoit en paramètre une fonction
  - Observable(f : lamda)
- Cette fonction (f) ayant pour paramètre un observateur.
  - f( o: Observateur)
- Cette fonction doit retourner une fonction de des-inscription de l'observable.
  - f( o: Observateur): **lamda**
- Par convention dans les anciennes versions, les noms des observables avec Angular sont suffixés de \$

# Observable Construction: Exemple

```
const sequencerFunc = (observer) => {// fonction qui initialisera l'observable  
// synchronously deliver 1, 2, and 3, then complete  
  observer.next(1);  
  observer.next(2);  
  observer.next(3);  
  observer.complete();  
// retourner une fonction de désinscription de l'observable  
  return ()=>{};  
};  
const observableSequencer$ = new Observable(sequencerFunc);
```

Il est alors possible de souscrire à `observableSequencer$`

# Observables: Souscription

- Une instance d'Observable (sujet) ne commence à publier que si elle a des observateurs (souscripteurs).
- La souscription se fait par l'appel de la méthode `subscribe()` sur l'observable

```
const myObservable$ = of(1, 2, 3); // Équivalent à l'exemple précédent diapo 7
// Create observer object (sujet)
const myObserver = {
  next: x => console.log('Observer got a next value: ' + x),
  error: err => console.error('Observer got an error: ' + err),
  complete: () => console.log('Observer got a complete notification'),
};
// Execute with the observer object
myObservable$.subscribe(myObserver);
```



# Observable: Suivi de position du navigateur

```
const locations$ = new Observable((observer) => {
  const onSuccess: PositionCallback = (pos: Position) => { observer.next(pos); };
  const onError: PositionErrorCallback = (error) => { observer.error(error); };
  let watchId;
  if ('geolocation' in navigator) {
    watchId = navigator.geolocation.watchPosition(onSuccess, onError);
  } else {
    onError({code: 100, message: 'Pas de geoloc ici', PERMISSION_DENIED: 1,
      POSITION_UNAVAILABLE: 1, TIMEOUT: 1});
  }
  const unsubscribe = () => { navigator.geolocation.clearWatch(watchId); };
  return unsubscribe;
});

const locationsSubscription = locations$.subscribe({
  next(position) { console.log('Current Position: ', position); },
  error(msg) { console.log('Error Getting Location: ', msg); }
});

setTimeout(() => { locationsSubscription.unsubscribe(); }, 30000);
```

# Observable: Suivi des touches du clavier

- La fonction **suiivante** crée un observable pour un *event* (nom de l'événement) et une *target* (élément du DOM) données

```
function fromEventObservable(target: HTMLInputElement, eventName: string) {  
    return new Observable((observer) => {  
        const handler = (e) => observer.next(e);  
        target.addEventListener(eventName, handler); // Add the event handler to the target  
        return () => {target.removeEventListener(eventName, handler);};  
    });  
}  
  
const nameInput = document.getElementById('name') as HTMLInputElement;  
const subscription = fromEventObservable(nameInput, 'keydown')  
    .subscribe((e: KeyboardEvent) =>  
        {// do something });
```

# La bibliothèque RxJS

- Cette bibliothèque contient les composants de programmation réactive pour JS qui seront utilisés par Angular
- On y retrouve entre autre
  - L'implémentation des Observables.
  - Des Convertisseurs des opérations asynchrone en observables
  - Des Convertisseurs boucle en flux (stream).
  - Le Mapping des valeurs en flux avec transformer
  - Le Filtrage en flux avec prédicats
  - Les opérations de composition de multiples flux

# RxJS: Observable à partir de fonctions

- Plusieurs fonctions sont définies dans RxJS pour transformer des opérations asynchrones (ou avec rappel) en des observables rendant leur gestion plus simple.
- Exemples
  - **from**( promise), ex: `fetch('/api/endpoint')`
  - **interval**(interv\_milliSec), ex: `interval(1000)` -> n chaque 1000ms -> 1s
  - **fromEvent**(domElementDom, eventName),
  - **ajax**(url-to-res)

```
import { from, interval, fromEvent, ajax } from 'rxjs';
```

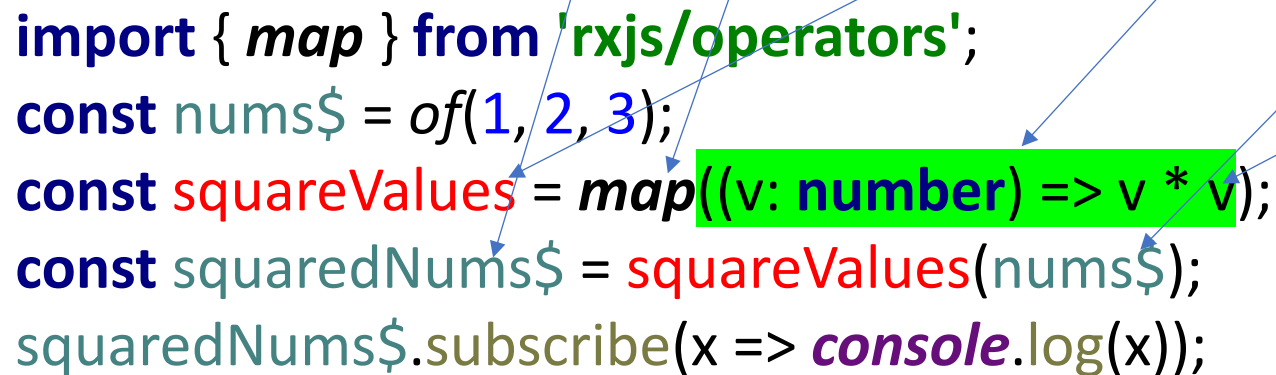
# Définitions (Domaines des définitions)

- Opérateur :
  - Transforme des fonctions en Opérateurs de Fonctions  
$$\text{Op: \{Fonctions\} \rightarrow \{Opérateurs de Fonctions\}}$$
- Opérateur de Fonction:
  - Transformation d'un observable en un autre observable.  
$$\text{OpFunc : \{Observables\} \rightarrow \{Observables\}}$$
- Observable d'ordre supérieur:
  - Un observable qui agit sur ou émet d'autres observables
- Observable de premier ordre:
  - Observables qui agit et émet des variables non observables.

# RxJS: Les Opérateurs (<https://rxjs.dev/api>)

- RxJS définit plusieurs opérateurs afin de réaliser des traitement sur les observables. Il s'agit de fonctions qui créent des fonctions de flux qui retournent des observables une fois appliquées aux observables.
- Ces fonctions reçoivent en paramètre des lambdas agissant sur les valeurs des observables.
  - Exemple: [map\(\)](#), [filter\(\)](#), [concat\(\)](#), and [flatMap\(\)](#).

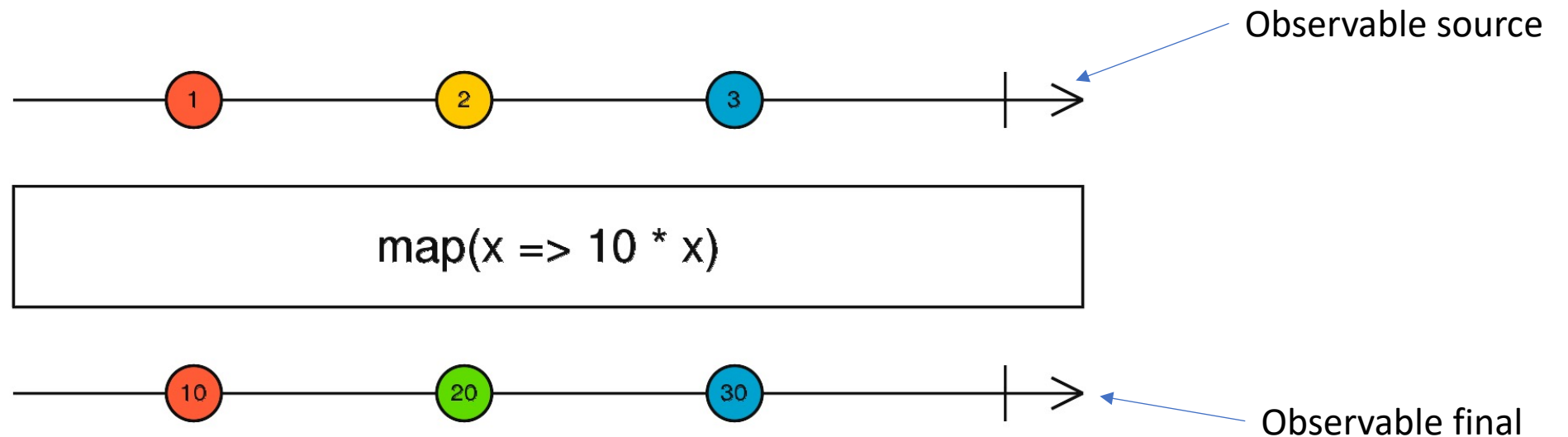
```
import { map } from 'rxjs/operators';  
const nums$ = of(1, 2, 3);  
const squareValues = map((v: number) => v * v);  
const squaredNums$ = squareValues(nums$);  
squaredNums$.subscribe(x => console.log(x));
```



# Opérateur map()

- Applique une fonction donnée à chaque valeur émise par l'observable source.
- map crée alors un opérateur de fonction qui, une fois appliqué à un observable, crée un autre observable qui réémet les valeurs transformées.

`map<T, R>(func: (value: T, index: number) => R, thisArg?: any): OperatorFunction<T, R>`



## Opérateur map(): Exemple

```
const nums$ = of(1, 2, 3); // observable source
```

```
// création de l'opérateur de fonction de mappage
```

```
const mapToSquare = map((v: number) => v * v);
```

```
// Transformation de l'observable source vers un autre selon mappage
```

```
const squaredNums$ = mapToSquare(nums$);
```

```
// squaredNums$ observable final
```

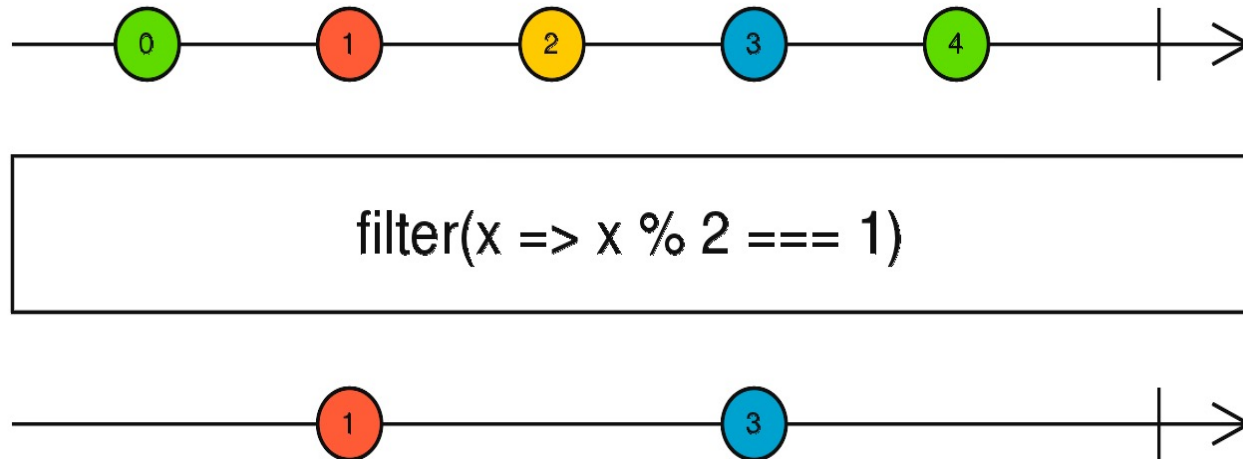
```
nums$.map((v: number) => v * v).subscribe(...)
```



# Opérateur filter()

- Applique un filtre sous format de prédicat aux valeurs émises par l'observable source.
- Filter crée alors un opérateur de fonction qui transforme l'observable source en observable qui ne réémet que les valeurs filtrées.

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



## Opérateur filter(): Exemple

*// observable source*

```
const nums$ = of(1, 2, 3, 4, 5);
```

*// création de l'opérateur de fonction du filtrer*

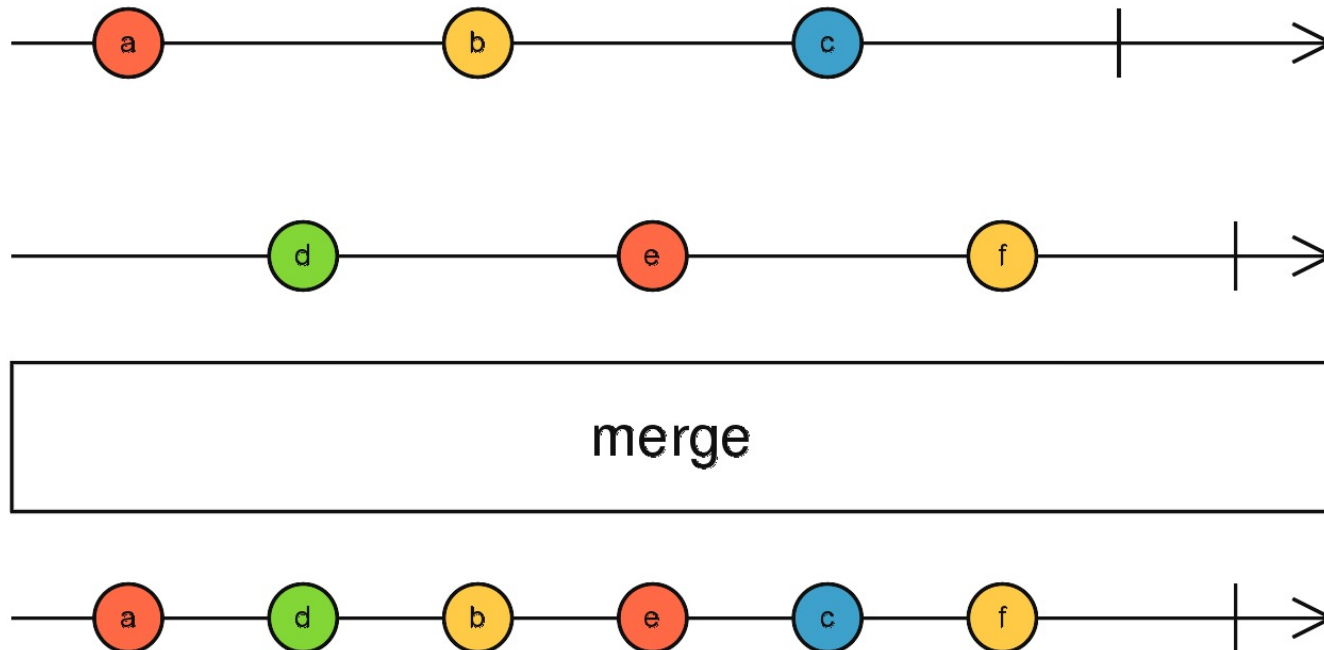
```
const filterOddVals = filter((n: number) => n % 2 !== 0);
```

*// Create an Observable that will run the filter and map functions*

```
const squareOdd$ = filterOddVals(nums$);
```

# Opérateur merge()

- Crée une sortie Observable qui émet simultanément toutes les valeurs de chaque entrée Observable donnée. On préfère utiliser la version *static*



# Opérateur merge(): Exemple

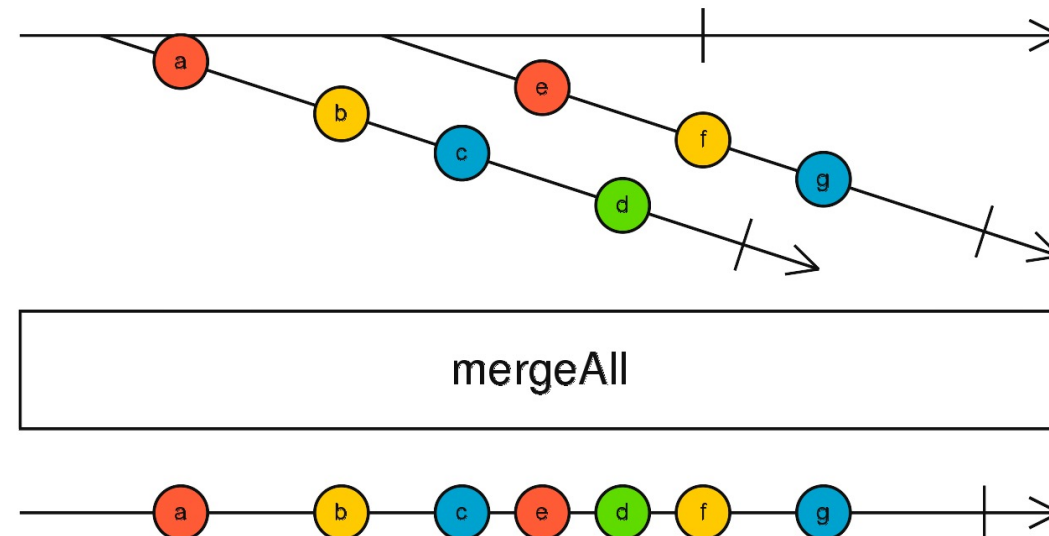
```
const timer1 = interval(1000).pipe(take(10));  
const timer2 = interval(2000).pipe(take(6));  
const timer3 = interval(500).pipe(take(10));  
const concurrent = 2; // the argument  
const merged = merge(timer1, timer2, timer3, concurrent);  
merged.subscribe(x => console.log(x));
```

- 1.// - First timer1 and timer2 will run concurrently
- 2.// - timer1 will emit a value every 1000ms for 10 iterations
- 3.// - timer2 will emit a value every 2000ms for 6 iterations
- 4.// - after timer1 hits its max iteration, timer2 will
- 5.// continue, and timer3 will start to run concurrently with timer2
- 6.// - when timer2 hits its max iteration it terminates, and
- 7.// timer3 will continue to emit a value every 500ms until it is complete

# Opérateur mergeAll()

- Convertit un observable d'ordre **supérieur** en un observable du **premier ordre** qui fournit simultanément toutes les valeurs émises sur les observables internes.

**function** mergeAll<T>(concurrent?: **number**): OperatorFunction<ObservableInput<T>, T>;



# Opérateur pipe

- L'opérateur pipe(op1, op2) permet de chaîner plusieurs observables/opérateurs de fonction:

```
const nums$ = of(1, 2, 3, 4, 5);  
const squareOddVals = pipe(  
  filter((n: number) => n % 2 !== 0), // premier  
  map(n => n * n) //deuxième  
);  
// Create an Observable that will run the filter and map functions  
const squareOdd$ = squareOddVals(nums$);
```


- Sa déclinaison fonctionnelle se trouve dans tous les observables

```
const squareOdd$ = of(1, 2, 3, 4, 5).pipe(filter(n => n % 2 !== 0)).pipe(map(n => n * n));
```

# Opérateur mergeAll(): Exemple

```
const clicks = fromEvent(document, 'click');  
const higherOrder = clicks.pipe(  
  map((event) => interval(1000).pipe(take(10)))  
);  
const merger = mergeAll(3);  
const firstOrder = merger (higherOrder); // ou higherOrder.pipe(merger);  
firstOrder.subscribe(x => console.log(x));
```

Crée un observable à  
chaque évènement

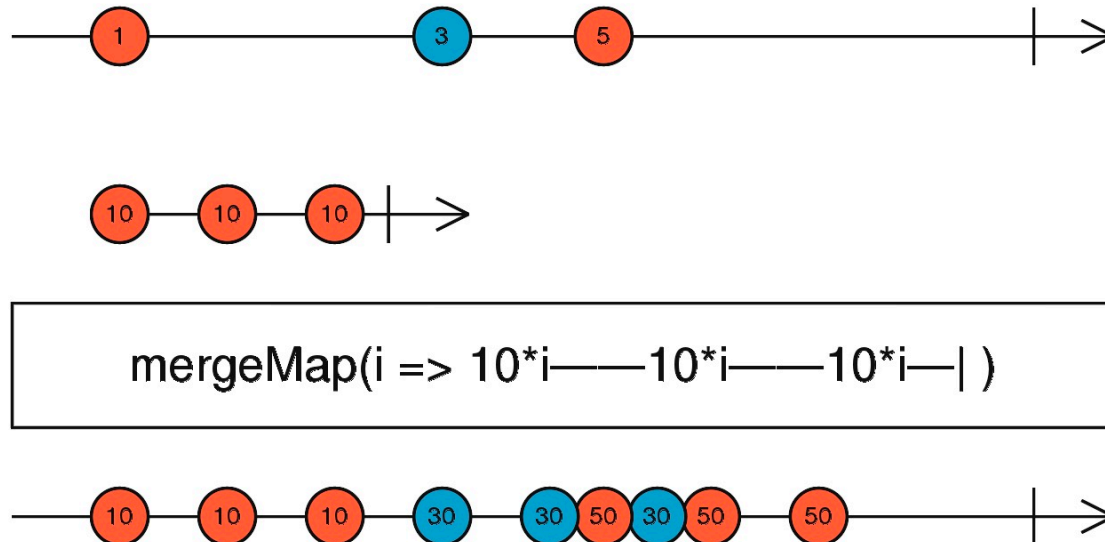


Compter de 0 à 9 secondes à chaque click,  
mais n'autoriser que 3 minuteries simultanées

# Opérateur mergeMap()

- Projette chaque valeur source dans un observable qui est fusionné à l'observable de sortie

```
function mergeMap<T, R>(this: Observable<T>, func: (value: T, index: number) => ObservableInput<R>,  
concurrent?: number): Observable<R>;
```





## Opérateur mergeMap(): exemple

```
const letters$ = of('a', 'b', 'c');  
const secondes$ = interval(1000);  
  
const merger$ = mergeMap(x => secondes$.pipe(map(i => x + i)));  
  
const result$ = merger$(letters$);  
  
// resultats de result$.subscribe(r=> console.log(r))  
//a1, b1, c1, a2, b2, c2, a3, b3, c3...
```

# Opérateur map vs mergeMap vs MergeAll

- Map permet des transformations élémentaires.
- MergeAll permet de combiner des observables d'ordre sup en un seul de 1<sup>e</sup> ordre
- MergeMap permet de combiner un observable à un autre et de mapper leur sortie à une func

*// using a regular map*

```
from([1,2,3,4]).pipe( map(param => getData(param)) )  
                    .subscribe(val => val.subscribe(data => console.log(data)));
```

*// using map and mergeAll*

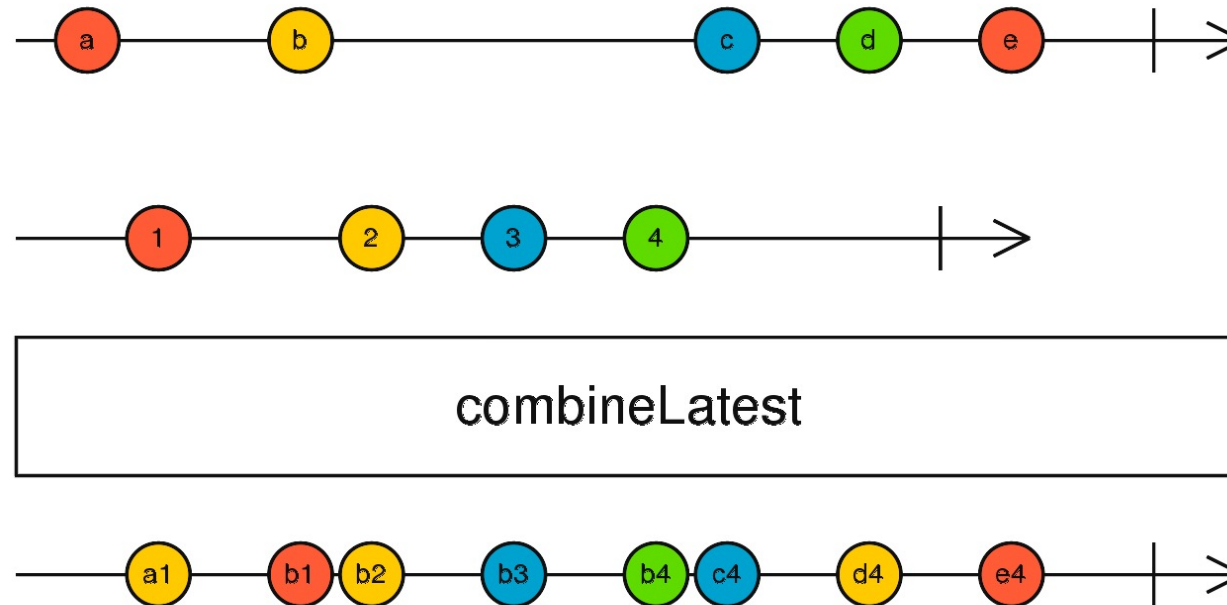
```
from([1,2,3,4]).pipe( map(param => getData(param)),  
                    mergeAll()  
                    ).subscribe(val => console.log(val));
```

*// using mergeMap*

```
from([1,2,3,4]).pipe( mergeMap(param => getData(param))).subscribe(val => console.log(val));
```

# Opérateur combineLatest

- Combine plusieurs observables pour créer une sortie observable dont les valeurs sont obtenues à partir des dernières valeurs de chacune des observables d'entrées.
- À chaque émission d'un observable d'entrée, l'observable de sortie émet une valeur contenant toutes les dernières valeurs de toutes les entrées observables, sous forme de tableau.



## Opérateur combineLatest: Exemple

*// emit 0, 1, 2... after every second, starting from now*

```
const firstTimer = timer(0, 1000);
```

*// emit 0, 1, 2... after every second, un delai de 0,5s*

```
const secondTimer = timer(500, 1000);
```

```
const combinedTimers = combineLatest(firstTimer, secondTimer);
```

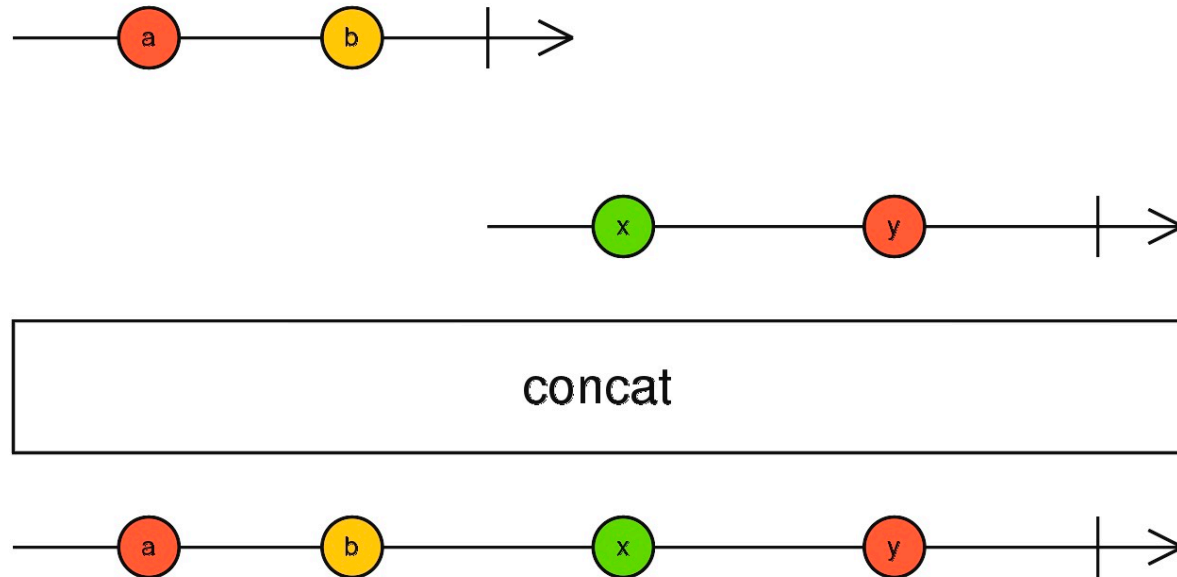
```
combinedTimers.subscribe(value => console.log(value));
```

*// Logs // [0, 0] after 0.5s // [1, 0] after 1s // [1, 1] after 1.5s // [2, 1] after 2s*

Pour garantir que le tableau de sortie a toujours la même dimension, combineLatest attendra que tous les observables en entrée émettent au moins une fois, avant de commencer à émettre des résultats.

# Opérateur concat

- Crée un observable en sortie qui émet séquentiellement toutes les valeurs d'un observable donné, puis passe au suivant.
  - Il concatène plusieurs Observables, en souscrivant à un à la fois et en fusionnant leurs résultats en sortie Observable.



## Opérateur concat: Exemple

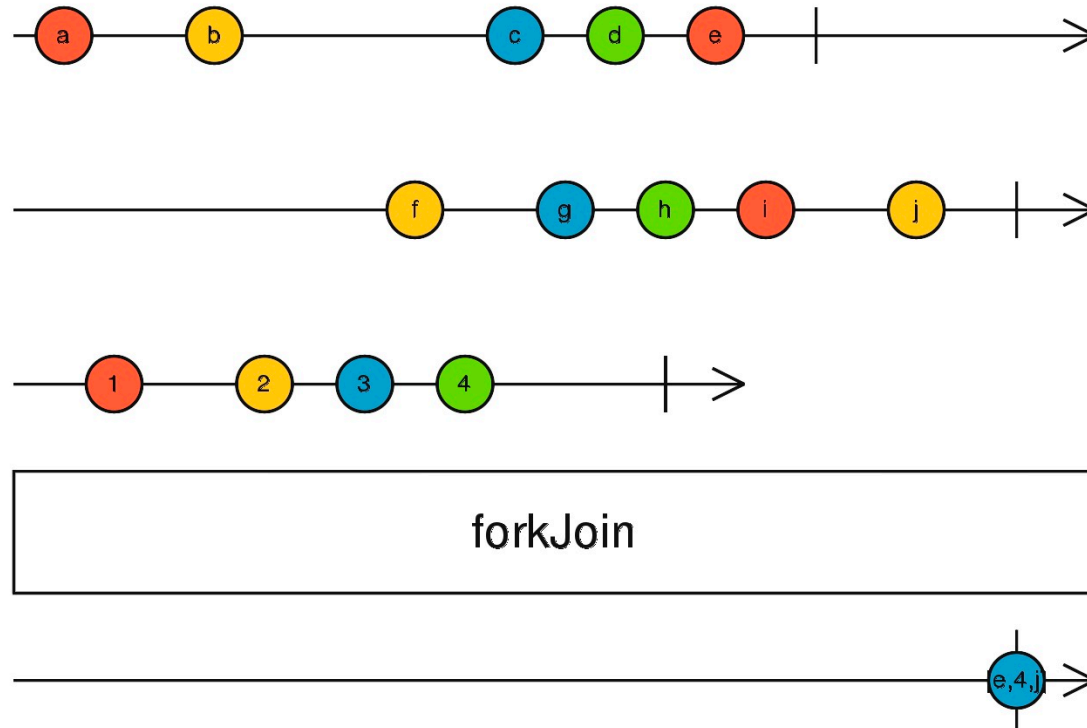
```
const myTimer = interval(1000).pipe(take(4));  
const mySequence = range(1, 10);  
const result = concat(myTimer, mySequence);  
result.subscribe(x => console.log(x));
```

- Il peut prendre en paramètre un tableau d'observables aussi

# Opérateur forkJoin

- Accepte un [] / {} d'Observables et retourne une sortie Observable émettant soit un [] / {} de valeurs dans le même ordre que [] / {} passé en paramètre.
- forkJoin attendra la complétion de tous les observables en paramètre.

forkJoin(...sources: any[]): Observable<any>



# Opérateur forkJoin: Exemple

```
const myNumbers = of(1, 2, 3, 4, 5);  
const myLetters = of('a', 'b', 'c' );  
const mySeconds = interval(1000).pipe(take(7));  
const result = forkJoin(myNumbers, myLetters, mySeconds);  
result.subscribe(x => console.log(x));
```

- Résultat ?



# RxJS: Les Opérateurs (<https://rxjs.dev/api>)

- L'opérateur **catchError** permet de traiter les erreurs sans interrompre le processus par une exception.
- L'opérateur **retry** permet de tenter plusieurs fois avant de catché et de traiter l'erreur.

```
const apiData = ajax('/api/data').pipe(retry(3), // Retry up to 3 times before failing
                                     map(res => {
                                       if (!res.response) {throw new Error('Value expected!'); }
                                       return res.response;
                                     }
                                     ),
                                     catchError(err => of([])) );
```

```
apiData.subscribe({
  next(x) { console.log('data: ', x); },
  error(err) { console.log('errors already caught... will not run'); }
});
```

# Observables et Angular

- Angular utilisent les observables à différents niveaux.
- Le décorateur **@Output**
  - @Output combiné avec EventEmitter génère un observable par réflexion pour transmettre les messages de l'enfant au parent
- Le client **HttpClient**
  - Le client retourne un observable sur lequel les opérateurs de transformation peuvent être chaînés grâce au pipe
- Le template Pipe **async**
  - Permet d'observer un Observable directement dans le Template grâce au pipe **async**.
  - l'enregistrement et le désenregistrement se font automatiquement.

# Observable et Angular: Exemple Template pipe

```
@Component({  
  selector: 'async-observable-pipe',  
  template: `<div>Time: {{ time$ | async }} </div>`  
})  
export class AsyncObservablePipeComponent {  
  time$ = new Observable(o =>  
    setInterval(() => o.next(new Date().toString()), 1000)  
  );  
}
```

# Observable et Angular :Filtre Recherche Live

```
const searchBox = document.getElementById('search-box');

const typeahead = fromEvent(searchBox, 'input').pipe(
  map((e: KeyboardEvent) => e.target.value), // transforme en observable de chaîne de caractères
  filter(text => text.length > 2), // plus de deux caractères
  debounceTime(10), // prévenir écriture effacement rapide
  distinctUntilChanged(), // prévenir écriture effacement du même caractère
  switchMap(() => ajax('/api/endpoint')) // Switch to new observable and cancel previous
);

typeahead.subscribe(data => {
  // Handle the data from the API
});
```

L'opérateur **switchMap** : <https://www.learnrxjs.io/learn-rxjs/operators/transformation/switchmap>

# Angular Service

- Les services en Angular sont vus comme des composants injectables.
  - Création de service *ng generate service <nom du service>*
- Le module d'injection de dépendance permet d'instancier les services pour un composant.
  - Il suffit de déclarer une instance du service dans les paramètres du constructeur du composant client.
- L'annotation (décorateur) de classe `@Injectable({providedIn: 'root'})` prend en paramètre la portée du service.
- En portée *root*, Le provider indique que le service est un singleton pour toute l'application.

# Angular Provider

- Les providers sont des composants utilisés par l'injecteur de dépendance pour déterminer la portée de chaque composant injectable.
  - Par défaut le Provider (provideIn dans l'annotation @Injectable) prend en paramètre le root.
  - Il peut être modifié pour réduire sa portée à un module particulier
    - Dans ce cas il reçoit en paramètre le module

```
@Injectable({  
  providedIn: UserModule,  
})
```

```
@NgModule({  
  providers: [UserService], // déprécié depuis la version 9  
})  
  
export class UserModule {  
}
```

# Angular HttpClient Introduction.

- Les navigateurs modernes supportent 2 APIs pour les appels selon le protocole.
  - XMLHttpRequest
  - fetch() : promise
- Le HttpClient de Angular a été construit sur l'api XMLHttpRequest.
- Il se trouve dans le module HttpClientModule.
- HttpClient est injectable, il s'agit donc d'un service.

# Angular HttpClient Méthode Get

- La méthode `get<T>(url , opts)` de `HttpClient` retourne
  - un `Observable<HttpResponse<T>>`
  - Quand une interface est définie pour le modèle, on peut l'utiliser à la place du générique `T`
  - Sinon, on utilise `any`
- Le paramètre `url` reçoit l'url de destination.
- Les paramètres `opts` sont optionnels et peuvent contenir
  - Le type d'objet que l'on souhaite observer au retour de la requête.
  - Le header que l'on souhaite envoyer à notre api.



# Angular HttpClient Méthode Get

```
retrievePrograms() {  
    this.http.get<Program[]>(this.url , { observe: 'response' })  
        .subscribe(result => this.programs = result.body,  
            error => { console.log('erreur de requete');  
                this.programs = [];}  
        );  
}
```

# Variante d'observable: Subject

- Les Subjects sont des observateurs observables.
- Leurs rôle est de jouer les intermédiaires entre la vraie source et les vrais observateurs.
- Dans ces conditions:
  - une seule souscription et faite à l'observateur
  - Le sujet peut partager ses résultats avec plusieurs observateurs finaux
  - La sources n'est exécutée qu'une fois.

# Variante d'observable: Subject, Exemple

```
const obs = Observable.interval(1000).take(10);  
obs.subscribe(x => console.log(x));  
setTimeout(() => {  
  obs.subscribe(x => console.log(x)); }, 4500);
```

- L'énumération de 0 à 9 se fera à partir de 0 pour le deuxième observateur.

```
const obs = Observable.interval(1000).take(10);  
const subject = new Subject();  
subject.subscribe(x => console.log(x));  
obs.subscribe(subject);  
setTimeout(() => {  
  subject.subscribe(x => console.log(x)); }, 4500);
```

- l'énumération de 0 à 9 ne se fera pas à partir de 0 pour le deuxième observateur.

# Variante d'observable: BehaviorSubject

- Un BehaviorSubject est un Subject qui se souvient de la dernière valeur émise.
- Étant donné que les sujets fournissent une exécution partagée, il est possible que les abonnés ultérieurs passent à côté des premiers éléments émis.
- BehaviorSubject a toujours une valeur initiales.

# Subject vs BehaviorSubject

```
const sub = new Subject();
console.log('First Subscription');
sub.subscribe(x => console.log(x));
sub.next(1);
sub.next(2);
sub.next(3);
setTimeout(() => { console.log('Second Subscription');
sub.subscribe(x => console.log(x)); }, 1000);
```

Le deuxième souscripteur ne verra aucun résultat

```
const sub = new BehaviorSubject(0);
console.log('First Subscription');
sub.subscribe(x => console.log(x));
sub.next(1);
sub.next(2);
sub.next(3);
setTimeout(() => { console.log('Second Subscription');
sub.subscribe(y => console.log(y)); }, 1000);
```

Le deuxième souscripteur verra toutes les résultats.  
Par ailleurs, l'observable initiale s'il y'en avait, ne serait pas appelée 2x

# Variante d'observable: ReplaySubject

- Ce sont des Subjects qui agissent comme des BehaviorSubject,
- Cependant, Ils ne gardent qu'un nombre limité des n derniers résultats.
- Ils ne prennent pas de valeur initiale.

```
const obs = Observable.of(1, 2, 3, 4, 5);
const sub = new ReplaySubject(3);
console.log('First Subscription');
sub.subscribe(x => console.log(x));
obs.subscribe(sub); setTimeout( () => {
    console.log('Second Subscription');
    sub.subscribe(x => console.log(x)); },
    1000);
```

# Variante d'observable: AsyncSubject

- Ce type de sujet n'émet que la dernière valeur émise, et seulement une fois le sujet terminé.

```
const sub = new AsyncSubject();
console.log('First Subscription');
sub.subscribe(x => console.log(x));
sub.next(1);
sub.next(2);
sub.next(3);
console.log('Second Subscription');
sub.subscribe(x => console.log(x));
setTimeout(() => sub.complete(), 1000);
```

La valeur trois est émise une fois à la fin et est loggée par les 2 observateurs

# Exercice d'application

- Visiter le site <https://openweathermap.org/api>
  - Ce site propose une API permettant d'avoir la météo gratuitement.
  - Créer un compte gratuitement, obligatoire pour avoir un APPID.
  - vous n'êtes pas obligé de mettre un bon email.
  - Le APPID vous permettra de pouvoir interroger l'api.
    - `http://api.openweathermap.org/data/2.5/XXXXXX&APPID={APIKEY}`
- Voici les requêtes permettant de trouver la météo
  - d'une ville à partir du nom
    - <https://openweathermap.org/current#name>
  - De coordonnées géographiques
    - <https://openweathermap.org/current#geo>
  - N villes à partir de leur coordonnées géographiques
    - <https://openweathermap.org/current#cycle>
- Faire une app avec live-search de météo de villes
  - Définir des interfaces en fonction des types de données
  - Utiliser le typeahead.
  - Localiser le navigateur de l'utilisateur et donner la météo de 10 villes autour de l'utilisateur.