Interfaces Funcionales y el Tipo Stream

Fundamentos de Programación
Departamento de Lenguajes y Sistemas Informáticos

2023/2024

2023/24

Interfases Funcionales y el tipo Stream

Índice

- Expresiones *Lamba*
- Expresiones *Referencia a Métodos*
- *Interfaces funcionales*
  - Comparator<T>
  - Function<T,R>
  - Predicate<T>
  - Consumer<T>
  - Suplier<T>
- Tipo *Stream*<T>

2023/24

Tipo Stream (*resumen hasta la fecha*)

Constructores:

Stream<T> *stream*()
Stream<T> *of* (obj1, obj2,…)
Stream<T> *concat* (Stream s1, Stream s2)

Operaciones terminales:

boolean allMatch (Predicate<T>)
boolean anyMatch (Predicate<T>)
T findAny()
T findFirst()
long count()
long sum()
OptionalDouble average()
T max(Comparator<T>)
T min(Comparator<T>)
void forEach(Consumer<T>)

Operaciones intermedias:

Stream<T> limit (long n)
Stream<T> skip (long n)
Stream<T> distinct ()
Stream<T> filter (Predicate<T>)
Stream<R> *map (Function<T, R>):*
    Stream<Integer> mapToInt(Function<T, Integer>)
    Stream<Long> mapToLong (Function<T, Long>)
    Stream<Double> mapToDouble (Function<T, Double>)
Stream<T> sorted()
Stream<T> sorted (Comparator<T>)

Tipo Stream (*resumen con lo nuevo*)

2023/24

Constructores:

Stream<T> *stream*()
Stream<T> *of* (obj1, obj2,…)
Stream<T> *concat* (Stream s1,
Stream s2)

Operaciones terminales:

boolean allMatch (Predicate<T>)
boolean anyMatch (Predicate<T>)
T findAny()
T findFirst()
long count()
long sum()
OptionalDouble average()
T max(Comparator<T>)
T min(Comparator<T>)
void forEach(Consumer<T>)

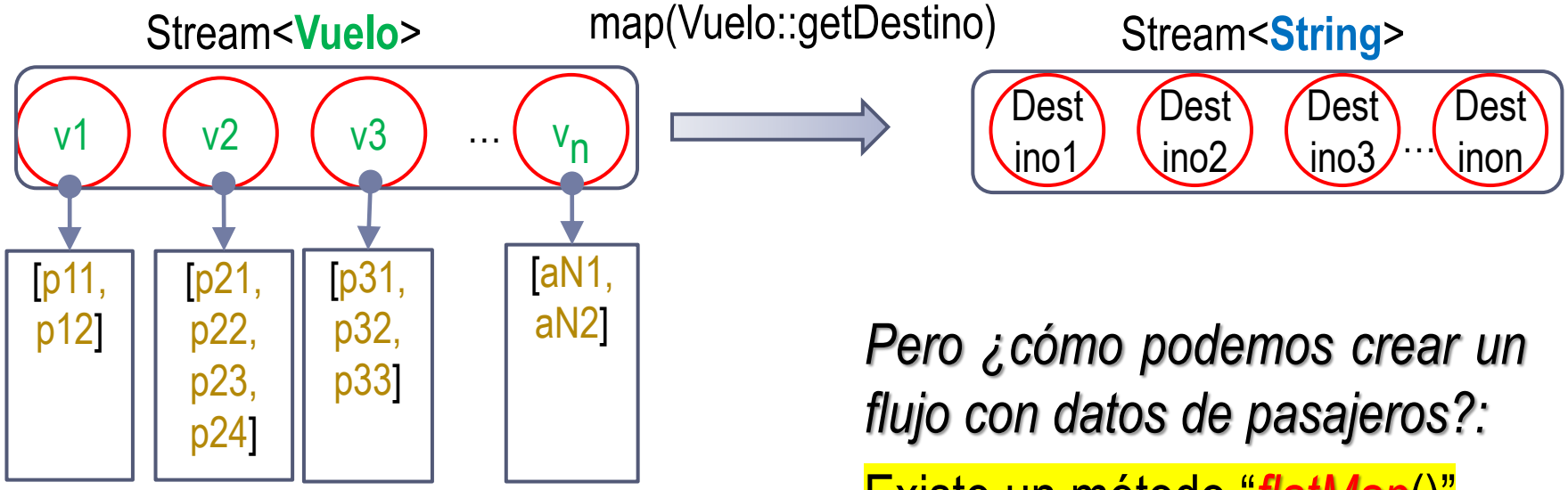
Operaciones intermedias:

Stream<T> limit (long n)
Stream<T> skip (long n)
Stream<T> distinct ()
Stream<T> filter (Predicate<T>)
Stream<R> map (Function<T, R>):
Stream<Integer> mapToInt(Function<T, Integer>)
Stream<Long> mapToLong (Function<T, Long>)
Stream<Double> mapToDouble (Function<T, Double>)
Stream<T> sorted()
Stream<T> sorted (Comparator<T>)
Stream<R> flatMap(Function<T, R>)
Stream<Integer> flatMapToInt(Function<T, Integer>)
Stream<Long> flatMapToLong (Function<T, Long>)
Stream<Double> flatMapToDouble (Function<T, Double>)
T *collect* (Collector c)

Tipo Stream

Otros Métodos para operaciones intermedias

- Habíamos estudiado el método *map* (Function<T, R>), que a partir de un tipo de objetos, obtiene otro que, normalmente, deriva de alguna/s de sus propiedad/es.



Listas con los **pasajeros** de cada vuelo

Pero ¿cómo podemos crear un flujo con datos de pasajeros?:

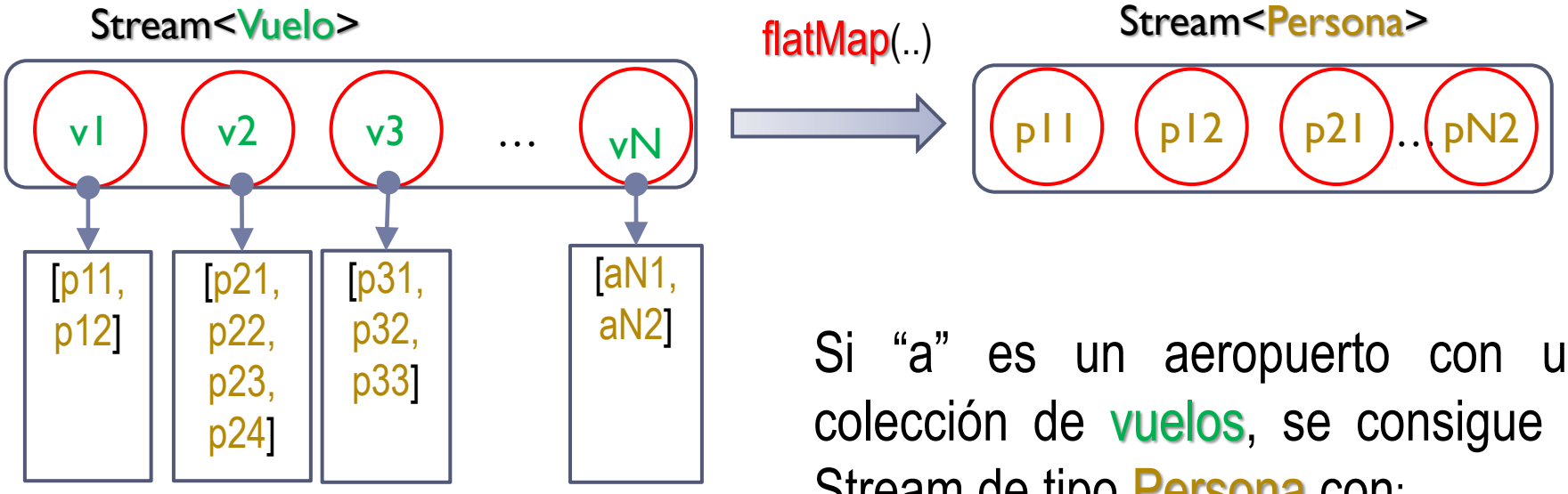
Existe un método "*flatMap*()"

Tipo Stream

Otros Métodos para operaciones intermedias

- Stream<*R*> *flatMap*(Function<*T*,*R*>):

  Veamos como "*aplanar*" los *pasajeros* que es una lista dentro de cada vuelo



Listas con los pasajeros de cada vuelo

Si "a" es un aeropuerto con una colección de vuelos, se consigue un Stream de tipo Persona con:

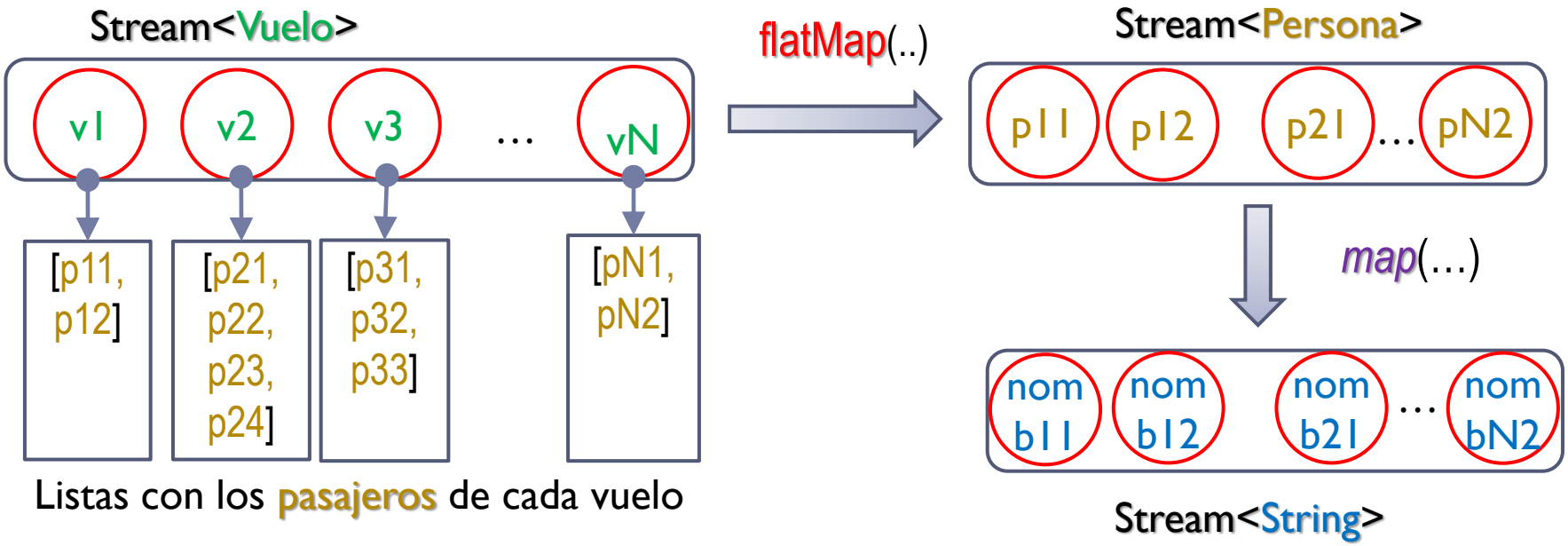
```
a.vuelos().stream().flatMap(v
  -> v.pasajeros().stream())
```

Tipo Stream

Otros Métodos para operaciones intermedias

Ejemplo: Supongamos que se quiere obtener una lista con los nombres de todos los pasajeros de los vuelos de un aeropuerto "a":

```
a.vuelos().stream().flatMap(v->v.pasajeros()
.stream()).map(Persona::nombre);
```



Tipo Stream

Otros Métodos para operaciones intermedias

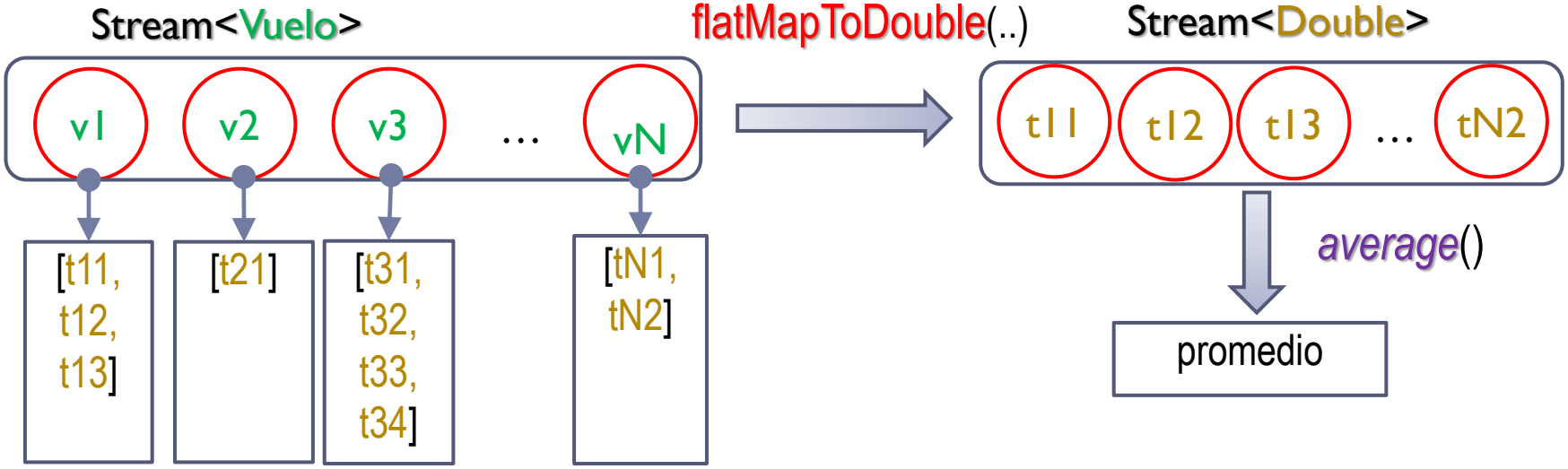
- Si la lista "más interna" es una lista numérica y sobre ella se desea realizar una operación matemática (sum, average,…), hay que usar:
  - Stream<Integer> *flatMapToInt*(Function<T, Integer>)
  - Stream<Long> *flatMapToLong* (Function<T, Long>)
  - Stream<Double> *flatMapToDouble* (Function<T, Double>)

Tipo Stream

2023/24

Otros Métodos para operaciones intermedias

Ejemplo: Si los *vuelos* de un aeropuerto "a", además de tener una lista de pasajeros, tuviesen una propiedad que fuese una lista de *temperaturas* (de tipo Double).



El promedio de las temperaturas de todos los vuelos sería:

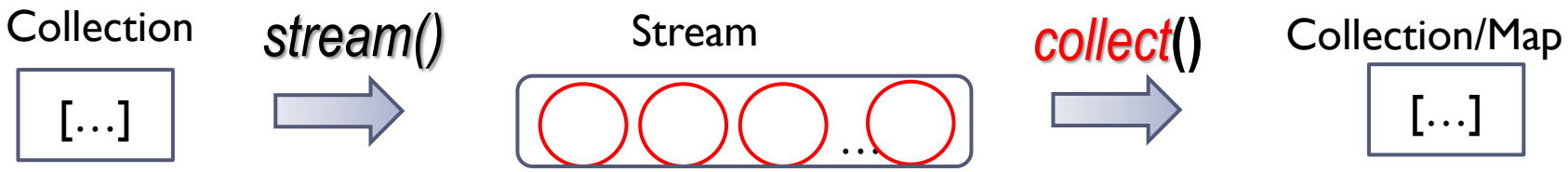
```
a.vuelos().stream().flatMapToDouble(v->v.temperaturas()
                                .stream()).average().orElse(0);
```

Tipo Stream

Otros Métodos para operaciones finales

- T *collect* (*Collector* c): El método *collect* permite transformar un *Stream* en colecciones ya estudiadas: *List, Set, SortedSet,* o bien: *Map y SortedMap*.

  Por decirlo de alguna forma, es el "inverso" al método stream()



Como método en sí mismo, no tiene nada interesante más que escribir correctamente la sintaxis: Si "s" es un *Stream* la sintaxis es: s.*collect*(…).

Lo importante de este método es son los parámetros, ya que realmente determina el comportamiento del mismo.

Así que vamos a ver los *principales métodos de la clase Collectors*.

Tipo Stream

2023/24

Métodos de Collectors:

Collectors.toList()
Collectors.toSet():
Collectors.toCollection(Supplier<C> constructor)
Collectors *groupingBy*(*con 1, 2 o 3 parámetros*)
Collectors.counting()
*Collectors.collectingAndThen(*método 1, método2*)*
Collectors.mapping(*Function*<T,R>, Collection)
Collectors.flatMapping(*Function*<T,R>, Collection)
Collectors.summingInt(propiedad numérica)
Collectors.summingLong(propiedad numérica)
Collectors.summingDouble(propiedad numérica)
Collectors.averagingInt(propiedad numérica)
Collectors.averagingLong(propiedad numérica)
Collectors.averagingDouble(propiedad numérica)
Collectors.maxBy(*Comparator*):
Collectors.minBy(*Comparator*):
Collectors.toMap(*Function*<T,K>, *Function*<T,V>)

Tipo Stream

2023/24

El tipo Collectors

- *Collectors.toList*() y *toSet*(): Permiten convertir un *Stream* (un flujo) en una lista o un conjunto respectivamente.

Ejemplo: A partir de un *Stream* de objetos tipo Vuelo "sv" se obtiene una lista de vuelos ordenada por el orden natural:

→
```
sv.sorted(Comparator.naturalOrder())
          .collect(Collectors.toList())
```

En el caso particular de obtener una lista, se puede sustituir *collect*(…) por *toList*:

→
```
sv.sorted(Comparator.naturalOrder()).toList()
```

Ejemplo en el que a partir de un *Stream* de objetos tipo Vuelo "sv" se obtiene un conjunto de vuelos:

→
```
sv.collect(Collectors.toSet())
```

Tipo Stream

El tipo Collectors

Collectors.*toCollection*(Supplier<C> constructor): Devuelve una colección según el constructor que se pasa como parámetro:

Ejemplos: A partir de un *Stream* de objetos tipo Vuelo "sv" obtener una lista de Vuelos (una forma más larga pero equivalente a *toList*())

- sv.collect(Collectors.*toCollection*(ArrayList::new));
- sv.collect(Collectors.*toCollection*(LinkedList::new)
- sv.collect(Collectors.*toCollection*(()->new ArrayList<Vuelo>())
- sv.collect(Collectors.*toCollection*(()->new LinkedList<Vuelo>());

Tipo Stream

El tipo Collectors

- De la misma manera, podemos construir un conjunto utilizando cualquiera de los constructores de conjuntos (*HashSet*; *TreeSet*; o *LinkedHashSet*), o un *SortedSet* con TreeSet.
  Ejemplo en el que a partir de un *Stream* de objetos tipo Vuelo "sv" se obtiene un conjunto de vuelos ordenada por el orden natural:
  → `sv.sorted(Comparator.naturalOrder())`
  `.collect(Collectors.toCollection(LinkedHashSet::new))`

- No obstante, cuando se use *TreeSet* para crear un conjunto ordenado, será necesario utilizar *expresión lambda* (no vale referencia a método) ya que hay que introducir como parámetro en el constructor un comparador.
  Ejemplo: Obtener un *SortedSet* ordenado por duración del vuelo

```
sv.collect(Collectors.toCollection(()->new TreeSet<Vuelo>
        (Comparator.comparing(Vuelo::duración)));
```

Tipo Stream

El tipo Collectors

- *Collectors.groupingBy*(paramétro/s): Devuelve un *Map* (mapa o diccionario).

  sv.collect(*Collectors.groupingBy*(*parámetro/s*))

  Los valores de los mapas pueden ser distintos tipos: Listas, conjuntos, conjuntos ordenados, *recuentos*, *porcentajes*, *máximos*, *mínimos*, etc, según los *parámetros* que reciba.

  Veremos que pueden ser *1, 2 o 3 parámetros*

Tipo Stream

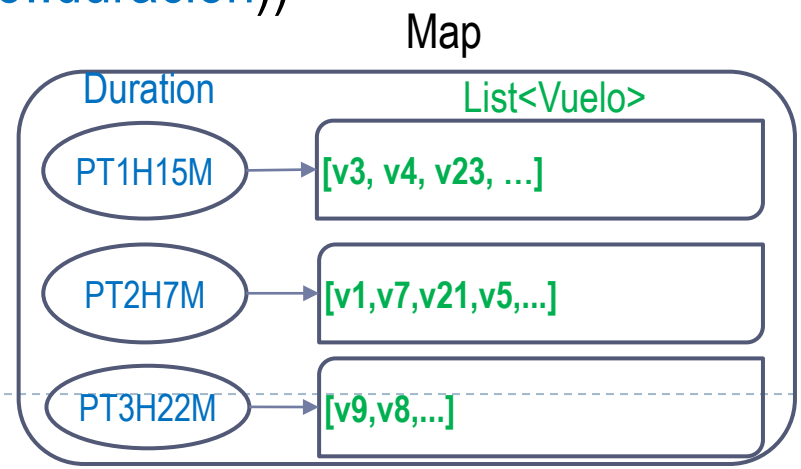
El tipo Collectors *groupingBy* (*con 1 parámetro*):

- El parámetro es una función y devuelve un Map en el que:
  - Las claves <R> son el resultado de dicha función: Funtion<T,R>
  - Los valores SIEMBRE son listas con los objetos <T> que se corresponden con el citado resultado.

Ejemplo: Obtener un diccionario de vuelos por duración:

Map<Duration,List<Vuelo>> ➔

sv.collect(*Collectors*.*groupingBy*(Vuelo::duración))



Tipo Stream

El tipo Collectors *groupingBy* (*con 2 parámetros*):

- Se usa cuando se desea que los valores del mapa que genera el groupingBy no sean listas, sino otro tipo de datos (*otras colecciones, contadores, sumas, promedios, máximos, mínimos, …*).
  - El primer parámetro, es una función Function<T,R>
  - El segundo será un método apropiado de *Collectors* en función de los valores que se pidan asociados a cada clave.

Tipo Stream

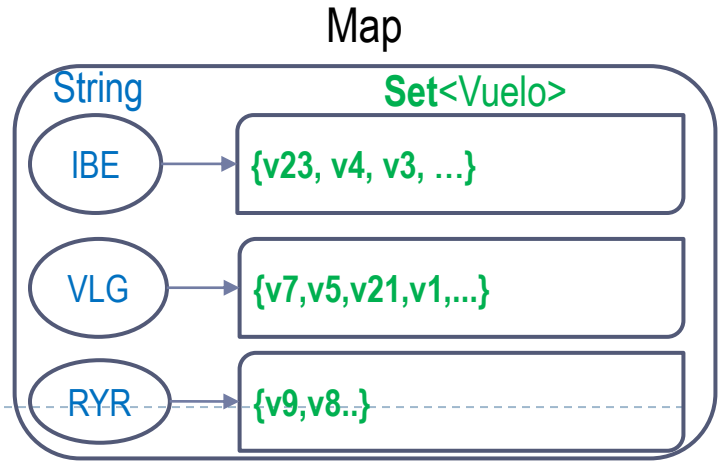
El tipo Collectors *groupingBy* (*con 2 parámetros*):

- *Collectors.toSet()*: para que los valores en lugar de ser listas (*como ocurre cuando el groupingBy que tiene un parámetro*), sean conjuntos.

  Ejemplo: Obtener conjuntos de vuelos por compañía (*se recuerda que la compañía son los tres primeros caracteres del código del vuelo*).

  Map<String, Set<Vuelo>> ➔

```
sv.collect(Collectors.groupingBy(v->v.compañía(),
                                 Collectors.toSet()))
```



Tipo Stream

2023/24

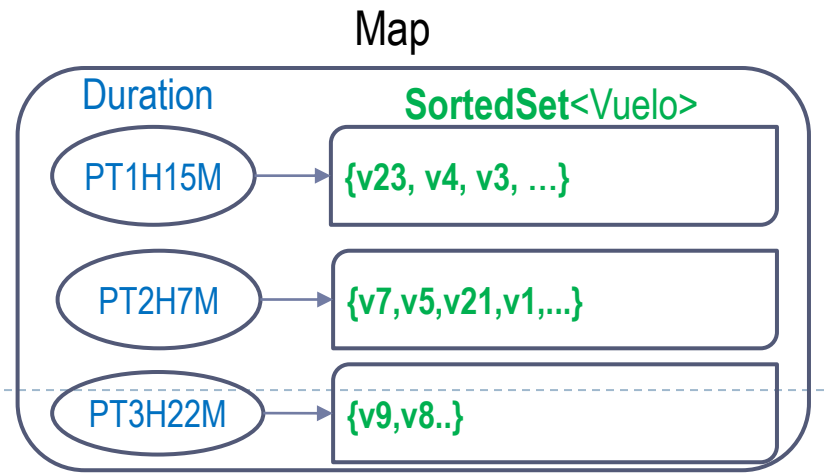
El tipo Collectors *groupingBy* (*con 2 parámetros*):

- *Collectors.toCollection()*: para que los valores sean otro tipo de colección (*normalmente un SortedSet cuyo constructor es un TreeSet*)

  Ejemplo: obtener un diccionario con los vuelos por duración y los valores son conjuntos ordenados por el número de pasajeros:

  Map<LocalDate,SortedSet<Vuelo>> ➔

```
sv.collect(Collectors.groupingBy(Vuelo::duracion,
    Collectors.toCollection(()->new TreeSet<Vuelo>
        (Comparator.comparing(Vuelo::númeroPasajeros))));
```



Tipo Stream

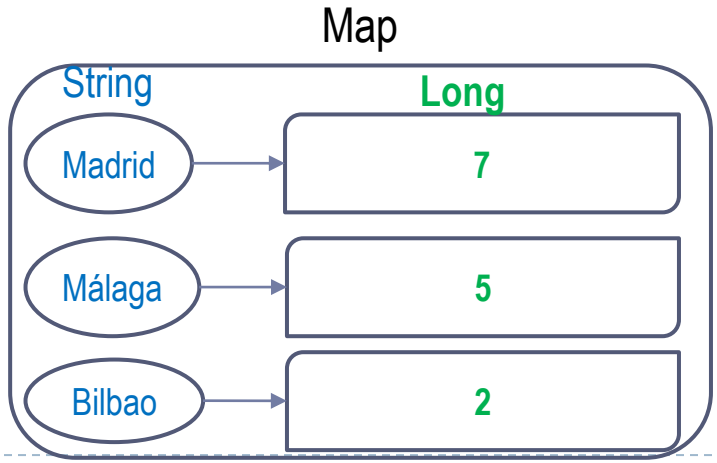
El tipo Collectors *groupingBy* (*con 2 parámetros*):

- *Collectors.counting():* Cuando los valores sean el número de objetos asociados a la clave (dichos valores serán de tipo *Long*).

  Ejemplo: Un diccionario que devuelva el número de vuelos a cada destino

  Map<String,Long> ➔

```
sv.collect(Collectors.groupingBy(Vuelo::destino,
                                 Collectors.counting())));
```



Tipo Stream

El tipo Collectors *groupingBy*(*con 2 parámetros*):

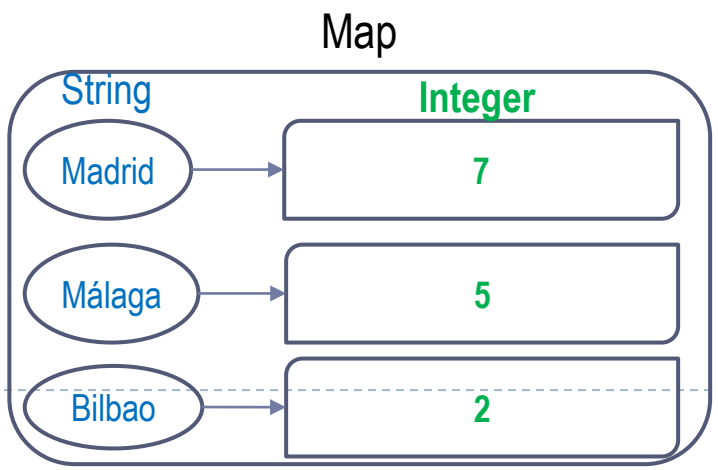
- *Collectors.collectingAndThen*(método 1, método2)
  Permite aplicar un segundo método al resultado de un primer método

  Ejemplo: Un diccionario que devuelva el número de vuelos a cada destino, siendo los valores de tipo *Integer*.

  Map<String,Integer>➔

```
sv.collect(Collectors.groupingBy(Vuelo::destino,
        Collectors.collectingAndThen(Collectors.counting(),
                c->c.intValue())));
```

(En la última expresión lambda, "c" representa el resultado de counting(). Permite convertir un tipo long en int)



Tipo Stream

El tipo Collectors *groupingBy* (*con 2 parámetros*):

- *Collectors.mapping*(Function<T,R>, Collection):

  Cuando los valores son colecciones de tipos distintos a la inicial del Stream. (*Es necesario convertir el tipo de entrada al groupingBy al tipo que se desea para la colección*).

  Importante: Observar que *mapping* tiene dos parámetros

  Ejemplo: Un diccionario que devuelva el conjunto con las duraciones de los vuelos que se producen en cada fecha de salida

  Map<LocalDate, Set<Duration>>➔

```
sv.collect(
 Collectors.groupingBy(v->v.fechaHoraSalida().toLocalDate(),
 Collectors.mapping(Vuelo::duration,Collectors.toSet())))
```

Es decir, el groupingBy tiene dos parámetros y, a su vez, mapping también tiene dos.

Tipo Stream

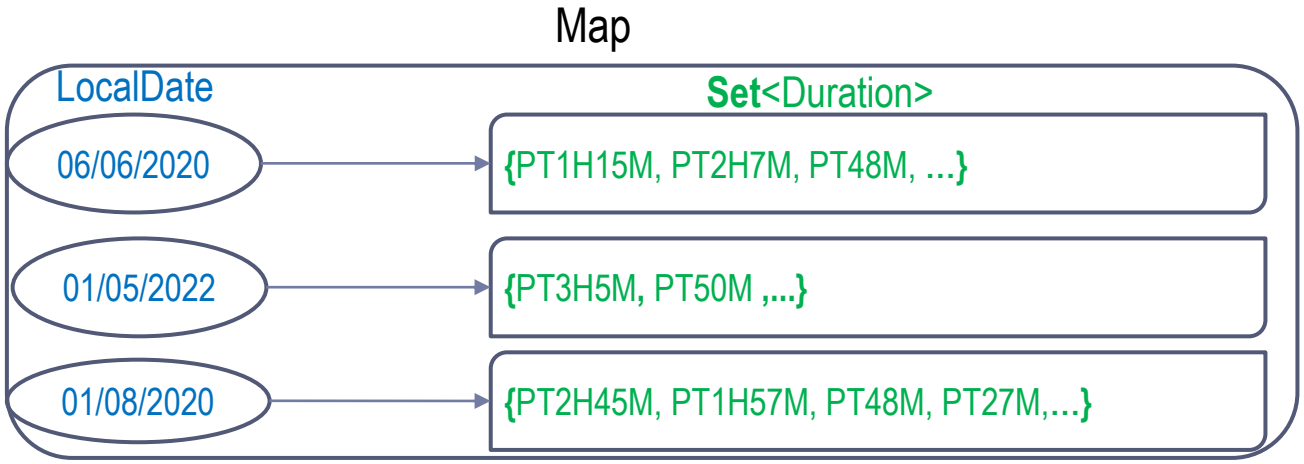
El tipo Collectors *groupingBy* (*con 2 parámetros*):

- *Collectors.mapping*(Function<T,R>, Collection):

  Ejemplo: Un diccionario que devuelva el conjunto con las duraciones de los vuelos que se producen en cada fecha de salida

  Map<LocalDate, Set<Duration>>➔

```
sv.collect(
  Collectors.groupingBy(v->v.fechaHoraSalida().toLocalDate(),
  Collectors.mapping(Vuelo::duration,Collectors.toSet())))
```



Tipo Stream

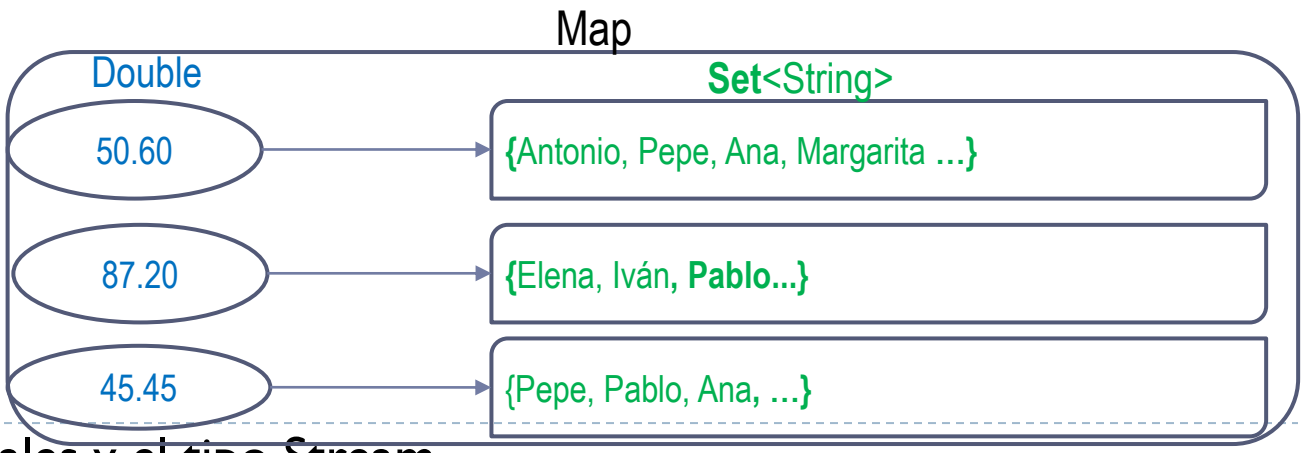
El tipo Collectors *groupingBy* (*con 2 parámetros*):

- *Collectors.flatMapping*(Function<T,R>, Collection):

Ejemplo: Un diccionario que devuelva el conjunto con los nombres de los pasajeros de los vuelos por cada precio. En este caso la propiedad es a su vez una colección ya hay que "aplanarla". (*ojo en la sintaxis*: lambda+stream)

Map<Double, Set<String>>➔

```
sv.collect(Collectors.groupingBy(Vuelo::precio,
                Collectors.flatMapping(v->v.pasajeros().stream(),
                                       Collectors.toSet())))
```



Ejercicio Aeropuerto

- Realice los ejercicios del ***EnunciadoAeropuerto08***

  Apartados 19 y 20