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

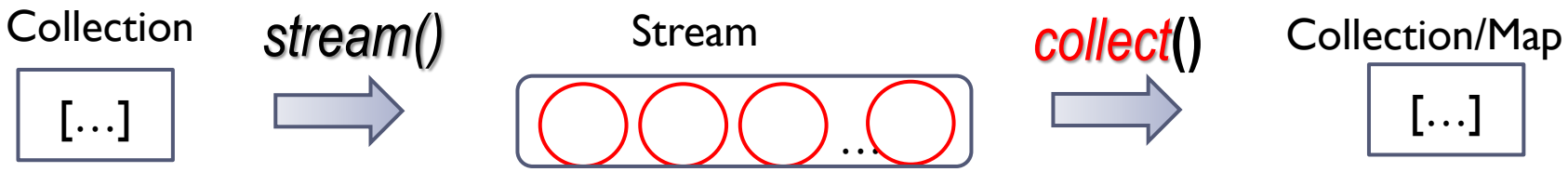
- Expresiones *Lamba*
- Expresiones *Referencia a Métodos*
- *Interfaces funcionales*
  - Comparator<T>
  - Function<T,R>
  - Predicate<T>
  - Consumer<T>
  - Suplier<T>
- Tipo *Stream*<T>

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

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 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

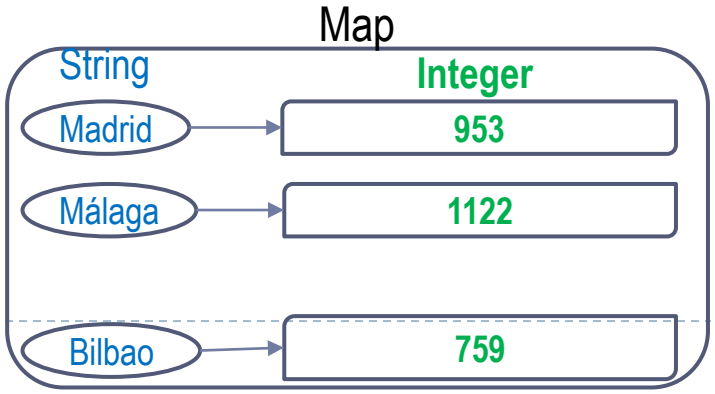
El tipo Collectors *groupingBy* (*con 2 parámetros*):

Para construir un mapa en el que los valores sean las sumas de determinada propiedad numérica, dependiendo del tipo de dicha propiedad, se usan como segundo parámetro de *groupinBy* los métodos:

- *Collectors.summingInt* (Function <T, Integer>): los valores son Integer.
- *Collectors.summingLong*(Function <T, Long>): los valores son Long.
- *Collectors.summingDouble*(Function <T, Double>): los valores son Double.

Ejemplo: Un diccionario que devuelva *la suma de pasajeros* a cada *destino*

```
Map<String,Integer>➔sv.collect(Collectors.groupingBy
(Vuelo::destino,Collectors.summingInt(Vuelo::númeroPasajeros)));
```



Tipo Stream

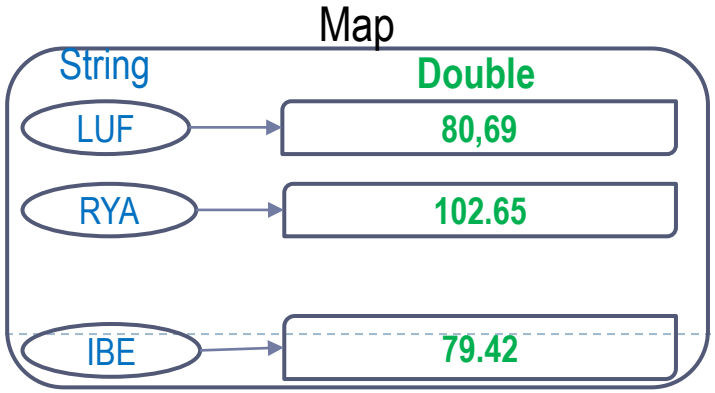
El tipo Collectors *groupingBy* (*con 2 parámetros*):

Para construir un mapa en el que los valores sean los promedios de determinada propiedad numérica, dependiendo del tipo de dicha propiedad, se usan como segundo parámetro de *groupinBy* los métodos:

- *Collectors.averagingInt*(Function <T, Integer>): los valores son *Double*
- *Collectors.averagingLong*(Function <T, Long>): los valores son *Double*
- *Collectors.averagingDouble*(Function <T, Double>): los valores son *Double*

Ejemplo: Un diccionario que devuelva *el promedio de precios* por compañía

```
Map<Compañía,Double>➔sv.collect(Collectors.groupingBy
(Vuelo::compañía,Collectors.averagingDouble(Vuelo::precio)));
```



Tipo Stream

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

Para construir un mapa en el que los valores ***sean el objeto*** que tenga el mayor o menor valor según determinada propiedad, se usan como segundo parámetro de *groupinBy* los métodos

- *Collectors.maxBy* (Comparator<T>): devuelve un *Optional<T>*
- *Collectors.minBy* (Comparator<T>): devuelve un *Optional<T>*

Ejemplo: Un diccionario que devuelva el ***vuelo*** con mayor número de plazas en cada fecha de salida.

Aparentemente la solución sería:

Map<LocalDate,Vuelo>➔

```
sv.collect(Collectors.groupingBy
(v->v.fechaHoraSalida().toLocalDate(),
Collectors.maxBy(Comparator.comparing(Vuelo::númeroPlazas)));
```

¡Pero hay que tratar el resultado Optional<T>!

Tipo Stream

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

Por ello, *maxBy* y *minBy* son el primer parámetro del método:

Collectors.collectingAndThen (método 1, método2)

La solución correcta sería (aplicar el máximo como método 1 y get como método2), quedando el ejercicio anterior:

Map<LocalDate,Vuelo>➔

```
sv.collect(Collectors.groupingBy(
 v->v.fechaHoraSalida().toLocalDate(),
 Collectors.collectingAndThen(
   Collectors.maxBy(Comparator.comparing(Vuelo::númeroPlazas)),
   m->m.get())));
```

(En este caso, en la expresión lambda "m" representa el resultado del maxBy/minBy)

Nota.- Si se busca una *propiedad concreta* del objeto (p.e. el código)

`m->m.get().código()));` y el mapa sería `Map<LocalDate,String>`

Tipo Stream

El tipo Collectors *groupingBy* (*con 3 parámetros*):

- Es una extensión de ***groupingBy*** con dos parámetros. El primero sigue jugando el mismo papel (*genera las claves*). ***El tercero*** juega el papel que antes jugaba el segundo (*genera los valores*) y tenemos un nuevo segundo parámetro que permite construir un ***SortedMap*** en lugar de un ***Map***:

```
collect(Collectors.groupingBy(Function,
                              ContructorMap,
                              Collección/objeto))
```

Tipo Stream

El tipo Collectors *groupingBy*(*con 3 parámetros*):

collect(*Collectors*.*groupingBy*(Function, ContructorMap, Collección/objeto))

Ejemplo: Obtener un mapa con la duración de los vuelos agrupados por fechas de llegada. Los valores serán conjuntos ordenados de mayor a menor duración de los vuelos. *Asimismo, las claves del Map deben estar a su vez ordenadas de menor a mayor fecha* (es decir un *SortedMap*).

SortedMap <LocalDate,SortedSet<Duration>>→

```
sv.collect(Collectors.groupingBy(
    v->v.fechaHoraLLegada().toLocalDate(),
    TreeMap::new,
    Collectors.mapping(Vuelo::duracion,
                       Collectors.toCollection(
              ()->new TreeSet(Comparator.reverseOrder()))));
```

Tipo Stream

El tipo Collectors *groupingBy*(*con 3 parámetros*):

Ejemplo: Obtener un mapa que a cada fecha de salida le indexe/asocie el vuelo con menor número de plazas de dicha fecha. *Asimismo, las claves del Map –las fechas- deben estar a su vez ordenadas desde más actuales a las más antiguas –en orden inverso-*.

SortedMap<LocalDate, Vuelo>→

```
sv.collect(Collectors.groupingBy(
   v->v.fechaHoraSalida().toLocalDate(),()->new
   TreeMap<LocalDate, Vuelo> (Comparator.reverseOrder()),
   Collectors.collectingAndThen(Collectors.minBy(
                  Comparator.comparing(Vuelo::númeroPlazas)),
                  min->min.get())));
```

(En este caso, en la expresión lambda "min" representa el resultado de minBy)

Tipo Stream

Ejercicios que combinan la realización de un Map (*groupingBy*) con *entrySet*

A veces se plantean ejercicios cuyo resultado no es un mapa, sino un valor que se obtiene haciendo un tratamiento posterior al resultado de un mapa.

Por ejemplo:

a) *¿Cuál es el destino con más vuelo*?
b) *¿En qué fecha de salida se puede volar a más destinos diferentes*?
c) *Obtener una lista en la que cada elemento sean los promedios de pasajeros de cada fecha de salida. Dicha lista debe estar ordenada de fechas recientes a más antiguas -orden inverso-*.

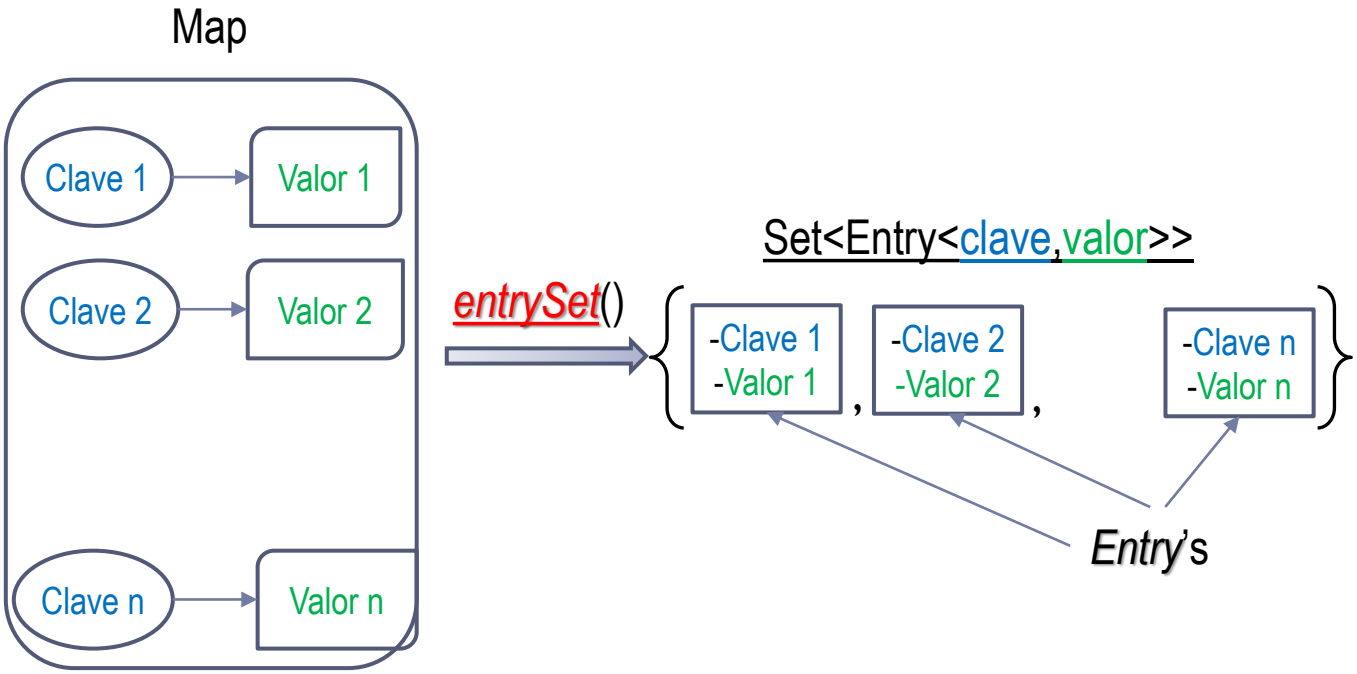
¿Qué tienen en común estos ejercicios?:

Que hay que agrupar una propiedad: a) por destino; b) y c) por fecha de salida. Por ello, primero habría que hacer un *map* que obtenga esas claves y sus correspondientes valores y después hacer un *tratamiento* de las parejas clave-valor del mapa.

Tipo Stream

RECORDAMOS que el método *entrySet* ():

- El método ***entrySet*** aplicado a un Map devuelve un conjunto (*Set*) con los pares: ***clave-valor***. (equivalente al método ítems() de Python). Los objetos del conjunto son de tipo: *Entry*<clave, valor>

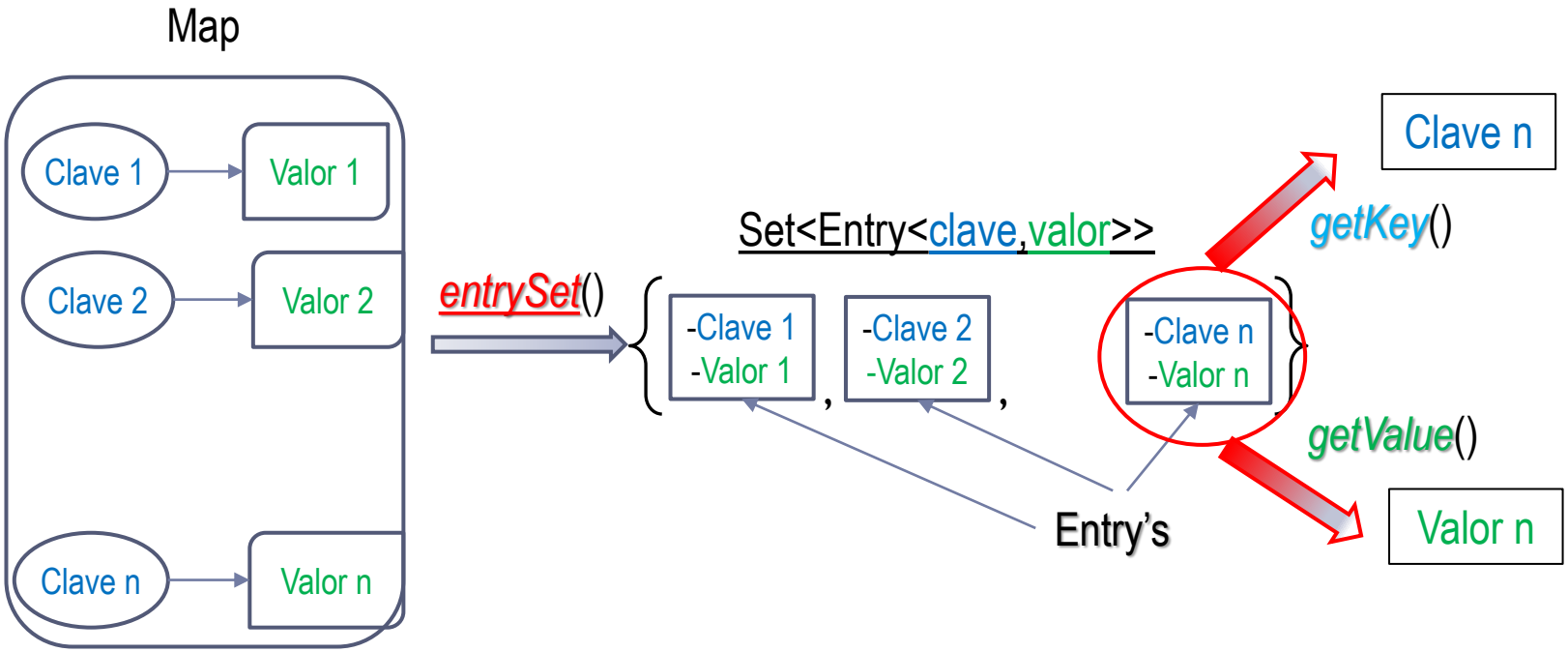


Tipo Stream

2023/24

El método *entrySet* ():

- A cada objeto de tipo *Entry*<clave, valor> se les puede aplicar los métodos:
- *getKey*() → devuelve la clave
- *getValue*()→ devuelve el valor.



Tipo Stream

Ejercicios que combinan *map* y *entrySet* ():

Ejemplo: *¿Cuál es el destino con más vuelos?.*

1) Habría que contar cuantos vuelos hay a cada destino. Esto se hace creando un Map<String,Long> de la siguiente forma:

   Map<String,Long>→`sv.collect(Collectors.groupingBy(Vuelo::destino, Collectors.counting()));`

2) Ahora se trata de obtener la clave cuyo valor es el máximo. Si convertimos las parejas clave-valor del mapa con *entrySet()*. en una colección tendremos un ejercicio sencillo de "Stream" de obtención de un máximo de objetos, por la segunda propiedad, de las dos que tienen.

```
entrySet().stream()
{.max(Comparator.comparing(p->p.getValue())).get().getKey();
 .max(Comparator.comparing(Entry::getValue)).get().getKey();
```

Tipo Stream

Ejercicios que combinan *map* y *entrySet* ():

3) ¡El ejercicio completo es!:

Aquí tenemos un Map

Aquí tenemos un Set

```
String→sv.collect(Collectors.groupingBy(
                                    Vuelo::destino,
                                    Collectors.counting()))
    .entrySet().stream()
    .max(Comparator.comparing(p->p.getValue()))
    .max(Comparator.comparing(Entry::getValue))
    .get().getKey();
```

Tipo Stream

Ejercicios que combinan *map* y *entrySet* ():

Ejemplo: *¿En qué fecha de salida se puede viajar a más destinos* ***diferentes****?*

1) Si se construye un mapa que a cada fecha le haga corresponder el número de destinos diferentes, después sólo habrá que convertir las parejas clave-valor en una colección y actuar como en el ejercicio anterior.
   Para que no haya destinos repetidos, usaremos un Set para guardar los diferentes destinos antes de contar cuantos elementos hay en él.

Map<LocalDate, Integer>→

```
sv.collect(Collectors.groupingBy(
      v->v.fechaHoraSalida().toLocalDate(),
      Collectors.collectingAndThen(
                  Collectors.mapping(Vuelo::destino,
                                     Collectors.toSet()),
                  conj->conj.size()))).
```

Tipo Stream

Ejercicios que combinan *Map* y *EntrySet* ():

2) Ahora tenemos un Map con las parejas clave-valor (fechas-número de destinos diferentes) que lo convertimos en una colección y se trata de un ejercicio sencillo de buscar el máximo de una colección.

```
entrySet().stream()
.max(Comparator.comparing(par->par.getValue()))
.get()
.getKey();
```

Tipo Stream

2023/24

Ejercicios que combinan *Map* y *EntrySet* ():

3) ¡El ejercicio completo es!:

LocalDate→

```
sv.collect(
  Collectors.groupingBy(v->v.fechaHoraSalida().toLocalDate(),
      Collectors.collectingAndThen(
      Collectors.mapping(Vuelo::destino, Collectors.toSet()),
      conj->conj.size())))
.entrySet().stream()
  .max(Comparator.comparing(par->par.getValue()))
  .get()
  .getKey();
```

Tipo Stream

Ejercicios que combinan *map* y *entrySet* ():

Ejemplo: Obtener una lista en qué cada elemento sean los promedios de pasajeros de los vuelos de cada fecha de salida. Dicha lista debe estar ordenada desde fechas recientes a lejanas -orden inverso-.

1) En primer lugar, la frase "*los promedios de pasajeros de los vuelos de cada fecha de salida*" invita a realizar un mapa con la clave la fecha y los promedios como valores.

   Map<LocalDate,Double>→

```
sv.collect(Collectors.groupingBy(
      v->v.fechaHoraSalida().toLocalDate(),
      Collectors.averagingInt(Vuelo::númeroPasajeros)));
```

Tipo Stream

Ejercicios que combinan *map* y *entrySet* ():

2) Si aplicamos el método *entrySet* al map convertimos el map en una colección *(un Set)* de objetos *Entry* con las parejas "fechas-promedios".

   Se trata de un ejercicio sencillo de Stream en el que a partir de una colección (*un Set*) con dos propiedades (getKey y getValue) ordenemos por una de ellas y obtengamos una colección (*una lista*) de la otra.

```
entrySet().stream()
    .sorted(Comparator.comparing(Entry::getKey).reversed())
    .map(Entry::getValue)
    .collect(Collectors.toList());
```

Tipo Stream

2023/24

Ejercicios que combinan *map* y *entrySet* ():

3) ¡El ejercicio completo es!:

List<Double>→

```
sv.collect(Collectors.groupingBy(
      v->v.fechaHoraSalida().toLocalDate(),
      Collectors.averagingInt(Vuelo::númeroPasajeros)))
.entrySet().stream()
    .sorted(Comparator.comparing(Entry::getKey).reversed())
    .map(Entry::getValue)
    .collect(Collectors.toList());
```

Ejercicio Aeropuerto

- Realice los ejercicios del ***EnunciadoAeropuerto09***

  Apartados 21 y 22