Interfaces Funcionales y el Tipo Stream

Fundamentos de Programación
Departamento de Lenguajes y Sistemas Informáticos

2023/2024

Interfaz Function

2023/24

Recordatorio 1: Hemos visto en la sesión anterior como definir funciones a través de una *expresión lambda*.

Por ejemplo:

- `x-> x*x,` que expresa: "dado un valor calcular su cuadrado"
- `v->v.getDuración()` que expresa: "dado un vuelo devuelve su duración"

También, hemos aprendido que una *expresión lambda* puede ser sustituida por la notación "*referencia a método*" cuando se trata sólo de invocar una propiedad de un tipo sin paso de parámetros, de forma que se puede escribir indistintamente:

- `v->v.getDuración()`
- `Vuelo::getDuración`

Interfaz Function

Recordatorio 2: Hemos aprendido a crear y usar comparadores alternativos.

Ejemplo de uso:

Ordenar una lista de vuelos `vuelos` por el número de pasajeros:

```
Collections.sort(vuelos,
                 Comparator.comparing(v->v.númeroPasajeros());
Collections.sort(vuelos,
                 Comparator.comparing(Vuelo::númeroPasajeros);
```

Pero también se puede *crear* una variable '*cmp*' para almacenar el comparador:

Ejemplo de creación y uso:

```
Comparator<Vuelo>cmp=
                 Comparator.comparing(v->v.númeroPasajeros());
Comparator<Vuelo>cmp=
                 Comparator.comparing(Vuelo::númeroPasajeros);
Collections.sort(vuelos, cmp);
```

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>

Interfaz Function

2023/24

Interfaz Function: Se trata de definir un tipo de dato que permita almacenar una función. En el caso de las funciones se dispone de la Interfaz funcional *Function<T,R>* que representa que recibiendo un dato de tipo *T* devolverá otro de tipo *R* (aunque *R* podría ser también *T*):

Por ejemplo:

- `Function<Integer,Integer>cuadrado=x->x*x;` (*T* y *R* son el mismo tipo)
- `Function<Vuelo,Duration>duración=v->v.duración()`
- `Function<Vuelo,Duration>duración=Vuelo::duración`
- `Function<LocalDate, Integer>año=LocalDate::getYear`
- `Function<LocalDate, Integer>año=f->f.getYear()`

Interfaz Function

2023/24

Métodos (más relevantes)

```
public interface Function<T, R> {
   default <V> Function<T,V>
       andThen(Function<? super R, ? super V> after);
   default <V> Function<V, R>
       compose(Function<? super V, ? super T> before);
   …
}
```

Interfaz Function

2023/24

Métodos (más relevantes)

- *andThen*: Permite concatenar dos funciones. Así *f1.andThen(f2)* obtiene una función resultado de aplicar *f1* y después aplica *f2*

Por ejemplo: *obtener la hora (de 0 a 24) en que sale un vuelo.*

```
- Function<Vuelo,Integer>horaDelVuelo=
                              v->v.fechaHoraSalida().getHour();
```

Pero se puede hacer por etapas:

```
- Function<Vuelo,LocalDateTime>función1=Vuelo::fechaHoraSalida;
- Function<LocalDateTime,Integer>función2=
                                   LocalDateTime::getHour;
- Function<Vuelo,Integer>horaDelVuelo=
                                   función1.andThen(función2);
```

Se ha optado por usar notación "referencia a método" para las funciones 1 y 2

Interfaz Function

2023/24

Métodos (más relevantes)

- *compose*: Permite la composición de dos funciones. Así *f1.compose(f2)* obtiene una función resultado de aplicar *f2* primero y *f1* después

El ejemplo anterior: también se obtiene usando ***compose()***

```
- Function<Vuelo,Integer>horaDelVuelo=
                        v->v.fechaHoraSalida().getHour();
```

Se puede realizar como:

```
- Function<Vuelo,LocalDateTime> función1=v->fechaHoraSalida();
- Function<LocalDateTime,Integer> función2=fh->fh.getHour();
- Function<Vuelo,Integer>horaDelVuelo=función2.compose(función1);
```

Se ha optado por usar notación "lambda expresión" para las funciones 1 y 2

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>

Interfaz Predicate

2023/24

Definición *Predicate* <*T*> es un caso particular de la interfaz *Función*<*T*,*R*>, donde *R* es *Boolean*.

Es decir, permite guardar una *función lambda* donde la expresión que aparece a la derecha de la **->** es una condición (siempre devuelve *true* o *false*).

Por ejemplo:

devuelve boolean

- `Predicate<Vuelo>destinoEs=v->v.destino().equals("Málaga");`

Lo anterior es más corto y expresivo que escribir:

- `Función<Vuelo,Boolean>destinoEs=`
  `v->v.destino().equals("Málaga");`

Interfaz Predicate

2022/23

Métodos (más relevantes)

```
public interface Predicate<T> {
    default Predicate<T> and (Predicate<? super T> p);
    default Predicate<T> negate();
    default Predicate<T> or (Predicate<? super T> p);
    …
}
```

Interfaz Predicate

2023z24

Métodos (más relevantes).

- *and: Permite construir un nuevo predicado a partir de otros dos. El nuevo predicado devolverá true si los otros, por separado, también son true. En otro caso devuelve false*

Por ejemplo:

```
-Predicate<Vuelo> destinoEs= v->v.destino().equals("Málaga");
-Predicate<Vuelo> hayPlazasLibres =
                    v->v.númeroPlazas()-v.númeroPasajeros()>0;
-Predicate<Vuelo>hayPlazasADestino=
                    destinoEs.and(hayPlazasLibres);
```

Es equivalente a escribir:

```
-Predicate <Vuelo> hayPlazasADestino=
            v->v.destino().equals("Málaga") &&
            v.númeroPlazas()-v.númeroPasajeros()>0;
```

Interfaz Predicate

2023/24

Métodos (más relevantes)

- *or: Permite construir un nuevo predicado a partir de otros dos. El nuevo predicado devolverá true si alguno de los otros, por separado, es true. En otro caso devuelve false.*

Por ejemplo:

```
- Predicate<Vuelo> esIberia=v->v.compañia().equals("IBE");
- Predicate<Vuelo> esVueling=v->v.compañia().equals("VLG");
- Predicate<Vuelo> esIberiaOVueling=esIberia.or(esVueling);
```

Es equivalente a escribir:

```
- Predicate <Vuelo> esIberiaOVueling =
   v->v.compañia().equals("IBE") || v.compañia().equals("VLG");
```

Interfaz Predicate

2023/24

Métodos (más relevantes)

- *negate: Invierte el resultado del predicado al que se aplica.*

Por ejemplo:

```
- Predicate <Vuelo> esIberia=v->v.compañia().equals("IBE");
- Predicate <Vuelo> noEsIberia = esIberia.negate();
```

Es equivalente a escribir:

```
- Predicate <Vuelo> noEsIberia=v->!v.compañia().equals("IBE");
```

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>

Interfaz Consumer

2023/24

Definición *Consumer* <*T*> es un caso particular de la interfaz *Función*<*T*,*R*>, donde *R* es *Void*.

Es decir, permite guardar una función lambda donde la expresión que aparece a la derecha de la -> no devuelve nada.

Por ejemplo:

- `Consumer<Vuelo> verCódigo=v->System.out.println(v.código());`
- `Consumer<Vuelo> cambiarPrecio=v->v.setPrecio(554.60);` *(esto es un ejemplo: nuestro Vuelo es record y no existen métodos set)*

Lo anterior es más corto y expresivo que escribir:

- `Function<Vuelo,Void>verCódigo=v->System.out.println(v.código());`
- `Function<Vuelo,Void>cambiaPrecio=v->v.setPrecio(554,60);` *(esto es un ejemplo: nuestro Vuelo es record y no existen métodos set)*

Interfaz Consumer

2023/24

Métodos (más relevantes)

```
public interface Consumer<T> {
    default Consumer<T>andThen(Consumer<? super T> after);
    …
}
```

Interfaz Consumer

2023/24

Métodos (más relevantes)

- *andThen*: Permite concatenar dos consumer. Así *f1.andThen(f2)* obtiene un nuevo *consumer* resultado de aplicar *f1* y, al resultado, *f2.*

Por ejemplo:

```
- Consumer<Vuelo>verPrecio=v->System.out.println(v.precio());
- Consumer<Vuelo>cambiaPrecio=v->v.setPrecio(554,60);
- Consumer<Vuelo>verNuevoPrecio=cambiaPrecio.andThen(verPrecio);
```

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>

Interfaz Supplier

2023/24

Definición *Supplier* <*R*> es una interfaz que permite guardar una función lambda donde la expresión no tiene parámetros de entrada y el tipo es el tipo de salida de una *function*<*T*,*R*>

Por ejemplo:

- `Supplier<Vuelo> vuelo1 = ()->new Vuelo (….);`
- `Supplier <List<Vuelo>>vuelos=()-> new ArrayList<Vuelo>();`
- `Supplier <Map<String,Integer>> númeroVuelosPorCompañía= ()->new HashMap<Map<String,Integer>>();`

Estas dos últimas, dado que el constructor no tiene parámetros, se puede escribir con la siguiente notación de referencia a método:

- `Supplier <List<Vuelo>> vuelos = ArrayList<Vuelo>::new;`
- `Supplier <Map<String,Integer>> númeroVuelosPorCompañía = HashMap<Map<String,Integer>>::new;`

Interfaz Supplier

2023/24

Métodos (más relevantes)

```
public interface Supplier<T> {
    … INTENCIONADAMENTE EN BLANCO
}
```

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

2023/24

Definición

El tipo *Stream*<*T*> permite trabajar de una forma cómoda y simple las colecciones de objetos para realizar "analítica de datos".

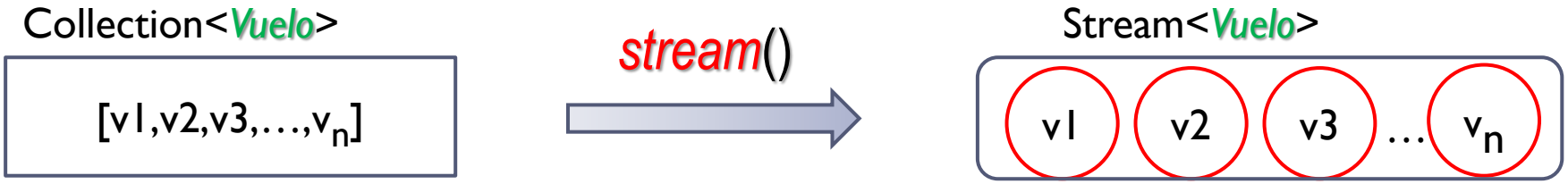
Métodos constructores

1) *Stream*<E> *stream():* permite construir un "flujo" a partir de una colección, al que le podremos aplicar un conjunto de métodos de la clase *Stream* que veremos a continuación.

Ejemplo:

Si tenemos Collection <*Vuelo*> *vuelos*, la expresión:

`vuelos.stream()` devuelve un *Stream*<Vuelo> (*flujo de vuelos*)



Tipo Stream

2023/24

Otros métodos estáticos que construyen flujos:

2) *Stream*<*T*> *of* (obj1, obj2,…): Es un método estático que permite crear un objeto *Stream* con los objetos que se pasan como parámetros (evidentemente todos tienen que ser de tipo *T*).

Ejemplo: Si v1, v2, v3 y v4 son cuatro vuelos:

`Stream<Vuelo> sv=Stream.of (v1, v2, v3, v4);` construye un *Stream* con cuatro vuelos.

3) *Stream*<*T*> *concat* (*Stream* s1, *Stream* s2): Es un método estático que construye un objeto *Stream* concatenando otros dos.

Ejemplo: Si sv1 y sv2 son flujos de vuelos:

`Stream<Vuelo> sv3=Stream.concat(sv1, sv2);` construye un *Stream* sv3 concatenando los vuelos de sv2 a los de sv1.

Tipo Stream

Métodos para operaciones finales

- boolean *allMatch* (Predicate<*T*>): Devuelve *true* si todos los objetos de *stream* que lo invoca cumplen la condición establecida en el predicado, o *false* si alguno no la cumple.

Ejemplo: comprobar si todos los vuelos están completos.

```
vuelos.stream().allMatch(v->v.vueloCompleto());
```

también

```
Predicate<Vuelo> esCompleto=v->v.vueloCompleto();
vuelos.stream().allMatch(esCompleto);
```

Tipo Stream

Métodos para operaciones finales

- boolean *anyMatch* (Predicate<*T*>): Devuelve *true* si alguno de los objetos de *stream* que lo invoca cumplen la condición establecida en el predicado, o *false* si ninguno la cumple.

Ejemplo: existe un vuelo a Sevilla

```
vuelos.stream().anyMatch(v->v.destino().equals("Sevilla"))
```

también

```
Predicate<Vuelo> vaASevilla=v->v.destino().equals("Sevilla");
vuelos.stream().allMatch(vaASevilla);
```

Tipo Stream

Métodos para operaciones finales

- long *count*(): Devuelve el número de objetos que tiene el *stream*.

Ejemplo: devolver el número de vuelos

```
vuelos.stream().count();
```

Tipo Stream

Métodos para operaciones finales

- long *sum*(): Devuelve la suma de los objetos del *stream* que invoca al método. ¡*Ojo!:* hay que aplicarlo después de alguno de los métodos intermedios: *mapToInt, ,mapToDouble, mapToLong (luego se explican)*

Ejemplo: devolver la *suma* de los precios de todos los vuelos

```
vuelos.stream().mapToDouble(Vuelo::precio).sum();
vuelos.stream().mapToDouble(v->v.precio()).sum();
```

Ejemplo: devolver la *suma* del número de plazas de todos los vuelos

```
vuelos.stream().mapToInt(Vuelo::númeroPlazas).sum();
vuelos.stream().mapToInt(v->v.númeroPlazas()).sum();
```

Tipo Stream

Métodos para operaciones finales

- *T findAny*(): Devuelve cualquier elemento del *stream* que lo invoca. Si el *stream* que invoca al método está vacío método (no se puede devolver ninguno), y evitar la excepción por defecto "*NoSuchElementException*", se debe concatenar:

  a) De modo que la sintaxis es:
  findAny().*orElse*(null o cualquier objeto);

  b) Si se quiere lanzar una excepción distinta a la "por defecto"
  findAny().*orElseThrow*(()->**new** *NombreExcepcion*());

Ejemplo: devolver un vuelo de la colección de vuelos. Si no hay vuelos lanzar "*IllegalStateException*".

```
vuelos.stream().findAny().orElseThrow(()->new
                    IllegalStateExcepcion());
```

Tipo Stream

Métodos para operaciones finales

- *T findFirst*(): Devuelve el primer elemento del *stream* que lo invoca. Si el *stream* que invoca al método está vacío método (no se puede devolver ninguno), y evitar la excepción por defecto "*NoSuchElementException*", se debe concatenar:

  a) De modo que la sintaxis es:
  findFirst()().*orElse*(null o cualquier objeto);

  b) Si se quiere lanzar una excepción distinta a la "por defecto"
  findFirst().*orElseThrow*(()->**new** *NombreExcepcion*());

Ejemplo: devolver el primer vuelo de la colección de vuelos. Si no hay vuelos devuelve null.

```
vuelos.stream().findFirst().orElse(null);
```

Tipo Stream

Métodos para operaciones finales

- *OptionalDouble average*(): Devuelve, el *promedio* de los objetos del *stream* que invoca al método. ¡*Ojo!:* hay que aplicarlo después de alguno de los métodos intermedios: *mapToInt, ,mapToDouble, mapToLong.*

  Para que devuelva un *Double* es necesario concatenar *getAsDouble*()
  ```
  average().getAsDouble().
  ```

  Si el stream está vacío lanza la excepción "*NoSuchElementException*" porque no se puede calcular el promedio. Se evita con dos opciones:

  a) Si se añade *average*().*orElse*(*valor*), devuelve el *valor*. *Por ejemplo*:
  ```
  average(). orElse(0.0).
  ```
  b) Si se añade *average*().*orElseThrow*(), se puede lanzar una determinada excepción, *Por ejemplo*:
  ```
  average().orElseThrow(()->new IllegalArgumentException())
  average().orElseThrow(IllegalArgumentException::new)
  ```

Tipo Stream

Métodos para operaciones finales

- *T max*(Comparator<*T*>): Devuelve el máximo inducido por el comparador. Para obtener el máximo por el orden natural del tipo *T*, hay que pasar como parámetro "*Comparator.naturalOrder()*"
  - A continuación de *max*(…) hay que añadir la llamada al método *get*() para que devuelva el objeto. De forma que la sintaxis es *max*(comparador).*get*(). Si se desea devolver una *propiedad* del objeto máximo buscado se añade a continuación el método consultor de que se trate: *max*(comparador).*get*().*getPropiedad*();
  - Si el *stream* está vacío lanza la excepción "*NoSuchElementException*" *Opciones para evitarlo*:
    a) La más común, como en *average*, es sustituir *get()* por *orElse*(null)
    b) O bien, *orElseThrow*(*NombreExcepción*::new).

Tipo Stream

Métodos para operaciones finales

- *T min*(Comparator<*T*>): Devuelve el mínimo inducido por el comparador.
  - Su comportamiento es el mismo del método *max*

- *void forEach*(Consumer<*T*>): Este método permite ejecutar una operación sobre todos los objetos del *stream* de entrada y no devuelve nada. Se limita a recorrer los objetos y aplicarles una operación de tipo *void* (por ejemplo, visualizar o realizar algún set).

Ejemplo: Visualizar los años de la fecha de salida de todos los vuelos

```
vuelos.stream().forEach(v->
        System.out.println(v.fechaHoraSalida().getYear()));
```

Tipo Stream

Métodos para operaciones intermedias

Son métodos invocados por un objeto *Stream* y devuelve otro objeto *Stream*

- *Stream*<*T*> *limit* (long n): Devuelve un *stream* con los n primeros objetos.
- *Stream* <*T*> *skip* (long n): Devuelve un *stream* saltando los n primeros. Si el n es mayor o igual que el número de objetos del *stream* que invoca al método lanza la excepción "*NoSuchElementException*"
- *Stream* <*T*> *distinct* (): Devuelve un *stream* con los mismos objetos, pero sin repetir.
- *Stream* <*T*> *filter* (Predicate<*T*>): Devuelve un *stream* con los objetos que cumplen el predicado.

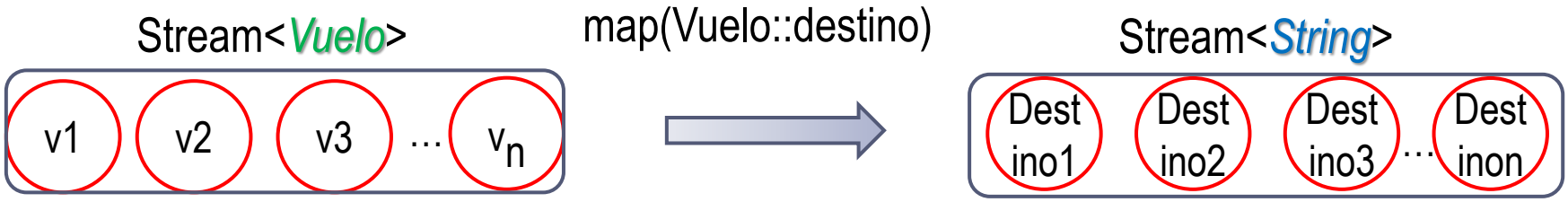
Ejemplo: devolver un *stream* con 5 vuelos distintos a Málaga:

```
vuelos.stream().filter(v->v.destino().equals("Málaga"))
                .distinct().limit(5);
```

Tipo Stream

Métodos para operaciones intermedias

- *Stream*<*R*> *map* (*Function*<*T*, *R*>): Aplica la función a los objetos (*T*) del *stream* que invoca al método, obteniendo otro *stream* de objetos (*R*).



Ejemplo: *devolver un stream con los destinos de todos los vuelos sin repetir*

```
vuelos.stream().map(Vuelo::destino).distinct()
```

Aquí hay un flujo de objetos Vuelo

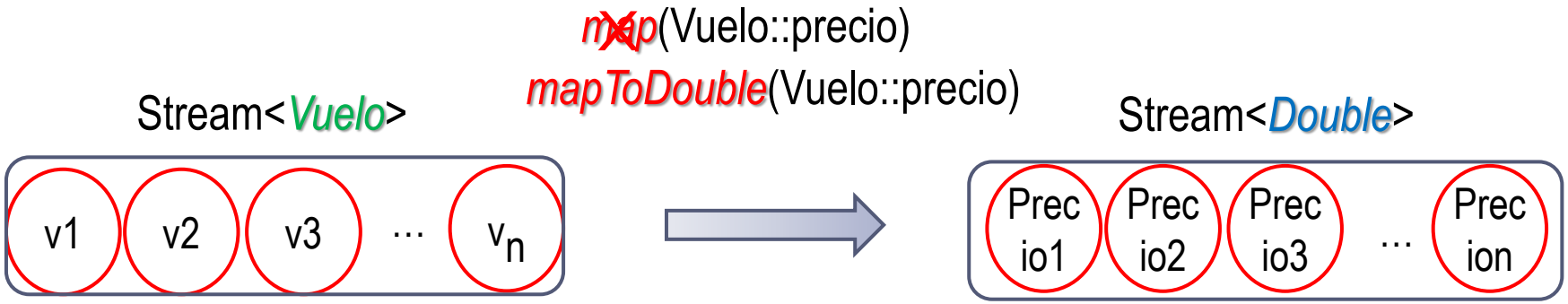
Aquí hay un flujo de objetos String

Aquí hay un flujo de objetos String sin repetir

Tipo Stream

Métodos para operaciones intermedias

- Acabamos de ver que *map* obtiene un *Stream* de un tipo de datos (*R*) a partir de otro tipo (*T*). No obstante, no se usa *map* cuando necesitemos obtener un *Stream* de datos numéricos *para aplicarle un método aritmético* (sumas, promedios, etc) es necesario sustituir *map* por:
  - Stream<*Integer*> *mapToInt*(Function<*T*, *Integer*>)
  - Stream<*Long*> *mapToLong* (Function<*T*, *Long*>)
  - Stream<*Double*> *mapToDouble* (Function<*T*, *Double*>)



Tipo Stream

Métodos para operaciones intermedias

- *Stream*<*T*> *sorted*(): A partir del *stream* que invoca al método, devuelve otro con los mismos elementos ordenados por el orden natural de tipo.

Ejemplo: obtener un *stream* con los vuelos ordenados por el orden natural.

```
vuelos.stream().sorted()
```

- *Stream*<*T*> *sorted* (*Comparator*<*T*>): A partir del *stream* que invoca al método, devuelve otro con los mismos elementos ordenados por el orden inducido por el comparador.

Ejemplo: obtener un *stream* con los vuelos ordenados por la duración.

```
vuelos.stream().sorted(Comparator.comparing(Vuelo::duración))
vuelos.stream().sorted(Comparator.comparing(v->v.duración()))
```

2023/24

Ejercicio Aeropuerto

- Realice los ejercicios del ***EnunciadoAeropuerto07***

  Apartado 18