USO DE
MAP, FLATMAP Y COLLECTOR

1.

MAP

MAP

- Una de las operaciones intermedias más usadas.
- Permite aplicar una transformación a una serie de objetos.
- Recibe como argumento un Function<T,R> para realizar la transformación.
- Se invoca sobre un Stream<T>, y retorna un Stream<R>

MAP

- Se pueden realizar transformaciones sucesivas

```
lista
    .stream()                          Stream<Persona>
    .map(Persona::getNombre)           Stream<String>
    .map(String::toUpperCase)          Stream<String>
    .forEach(System.out::println);
```

2.

FLATMAP

FLATMAP

- Los streams sobre colecciones de *un nivel* (como *List*) se pueden transformar (*map*) fácilmente.
- ¿Qué sucede si tenemos una colección que incluye dentro otra?

Persona 1			Persona 2			Persona 3		
Viaje 1	Viaje 2	Viaje 3	Viaje 4	Viaje 5	Viaje 6	Viaje 7	Viaje 8	Viaje 9

FLATMAP

- En el viejo estilo for, anidaríamos dos bucles:

```
for (Persona p : lista)
     for (Viaje v : p.getViajes())
          System.out.println(v.getPais());
```

FLATMAP

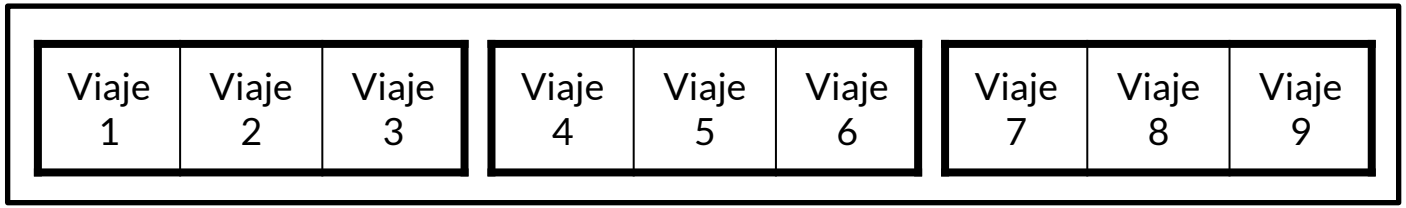
- Podemos observar bien para darnos cuenta los tipos de retorno de los métodos intermedios:

```
// Intento sin flatMap
lista
    .stream()                                   Stream<Persona>
    .map((Persona p) -> p.getViajes())          Stream<Stream<Viaje>>
    //…
    .forEach(System.out::println);
```

FLATMAP

- Necesitamos un método que unifique un *Stream<Stream<T>>* en un solo *Stream<T>*:
- Esa es la funcionalidad de *flatMap*.

Stream<Stream<Viaje>>



Stream<Viaje>

FLATMAP

- Con flatMap podemos transformar los streams y unificarlos en uno:

```
lista
    .stream()
    .map((Persona p) -> p.getViajes())
    .flatMap(viajes -> viajes.stream())
    .map((Viaje v) -> v.getPais())
    .forEach(System.out::println);
```

FLATMAP

- También tenemos la versiones primitivas (flatMapToInt, …):

```
int[][] numeros = { {1, 2, 2, 3, 1, 4},
                    {4, 2, 3, 3, 1, 1} };

Arrays
     .stream(numeros)
     .flatMapToInt(x -> Arrays.stream(x))
     .map(IntUnaryOperator.identity())
     .distinct()
     .forEach(System.out::println);
```

3.

COLLECTORS

COLLECTORS

- Hasta ahora, las operaciones realizadas con streams han acabado con una salida por consola.
- ¿Y si queremos transformar un stream (*inmutable*) y guardar su resultado en una colección (*mutable*)? Operación *collect*.
- Java SE 8 introduce **Collectors**, con métodos estáticos muy usuales (y prácticos).

```
import static java.util.stream.Collectors.*;
```

COLLECTORS “BÁSICOS”

- Nos permite realizar algún tipo de operación y recolectar el valor en uno solo.
- Algunos se solapan con operaciones finales que ya hemos visto; existen para usarse junto con otros colectores.

COLLECTORS “BÁSICOS”

- *counting()*: cuenta el número de elementos.
- *minBy(…)*, *maxBy(…)*: obtiene el mínimo o máximo según un comparador.
- *summingInt*, *summingLong*, *summingDouble*: la suma de los elementos (según el tipo).
- *averagingInt*, *averagingLong*, *averagingDouble*: la media (según el tipo).
- *summarizingInt*, *summarizingLong*, *summarizingDouble*: los valores anteriores, agrupados en un objeto (según el tipo).
- *joinning*: unión de los elementos en una cadena.

COLLECTORS “GROUPING BY”

- Similar a la cláusula GROUP BY de SQL.
- Permiten agrupar los elementos de un stream por un determinado valor.
- Retorna un *Map* con los diferentes grupos, y los elementos de cada grupo.

```
Map<String, List<Empleado>> porDepartamento =
   empleados
     .stream()
     .collect(groupingBy(Empleado::getDepartamento));
```

COLLECTORS “GROUPING BY”

- Se pueden usar uno de los colectores básicos, para realizar algún cáculo

```
Map<String, Long> porDepartamentoCantidad =
   empleados
      .stream()
      .collect(groupingBy(Empleado::getDepartamento, counting()));
```

COLLECTORS “GROUPING BY”

- Se pueden crear varios niveles de agrupamiento:

```
Map<String, Map<Double, List<Empleado>>> porDepartamentoYSalario =
     empleados
     .stream()
     .collect(groupingBy(Empleado::getDepartamento,
                         groupingBy(Empleado::getSalario)));
```

COLLECTORS “PARTITIONING”

- Se pueden agrupar en dos conjuntos, según si cumplen la condición de un predicado.

```
Map<Boolean, List<Empleado>> salarioMayorOIgualque32000 =
   empleados
     .stream()
     .collect(partitioningBy(e -> e.getSalario() >= 32000));
```

COLLECTORS “COLLECTION”

- Producen como resultado una de las colecciones ya conocidas: List y Set
- También puede producir colecciones de tipo key, value, como Map.