Índice

Introducción

El avance que representa Java 8 con respecto a versiones anteriores es muy significativo y nos va a obligar a modificar la perspectiva de cómo enseñar a programar a nuestros estudiantes. Estos apuntes intentan ser un resumen de las principales características que hemos encontrado en un apresurado estudio de las nuevas funcionalidades del tipo Stream. Todo el código aquí presentado ha sido implementado usando Eclipse Versión: Luna (4.4) Build id: I20140501-0200.

Para ejemplificar los casos de uso partimos de un tipo Vuelo con la siguiente interfaz:

```
public interface Vuelo  extends Comparable<Vuelo>{
	String getCodigo() ;
	String getDestino() ;
	Fecha getFecha();
	Integer getNumPasajeros();
	Integer getNumPlazas() ;
	Duracion getDuracion();
	Double getPrecio() ;
	Double getOcupacion(); //opcional
	Double getRecaudacion(); //opcional
	void setPrecio(Double p);
	void setDuracion(Duracion d);
	void setNumPasajeros(Integer np) ;
}
```

El tipo Duracion implementa el tiempo de un Vuelo y tiene la funcionalidad siguiente

```
public interface Duracion extends Comparable<Duracion>{
	Integer getMinutos();
	Integer getHoras();
	Duracion suma(Duracion d);
}
```

El objetivo es implementar un tipo Aeropuerto que manipula como atributo una colección de objetos de tipo Vuelo que hemos denominado `vuelos` y requiere una serie de funcionalidades que se irán presentado en cada uno de los siguientes secciones en que se ha dividido el texto para facilitar su comprensión. La dificultad del código es creciente y en esta primera versión no se explican los objetos que se manipulan como Predicate, Function, Supplier etc que serán tratados en apuntes aparte en una segunda fase.

Nos centramos en este documento en la estructura Stream que es un iterable virtual sobre objetos. El método stream() aplicado a una Collection permite obtener un Stream a partir de los objetos de la Collection. Por eso siempre el código de las siguientes secciones comienza con una sentencia `vuelos.stream()`. A continuación se describen las principales funcionalidades del tipo Stream.

1. filter(Predicate)

Sirve para devolver otro Stream con sólo aquellos elementos del que invoca que cumplen el predicado. Se suele componer con métodos como count para calcular el número de elementos de una colección que cumplen una determinada condición. Por ejemplo, cuántos vuelos hay en una fecha determinada:

```
	public Long getNumVuelosDia(Fecha f) {
		return vuelos.stream().
			filter(x->x.getFecha().equals(f)).
			count();
	}
```

O cuántos vuelos completos hay:

```
	public Long getNumVuelosCompletos() {
		return vuelos.stream().
			filter(x->x.getNumPasajeros().equals(x.getNumPlazas())).
			count();
	}
```

Si un objeto de tipo Predicate se va a usar a menudo, debe definirse previamente y después invocarse. Por ejemplo, la condición de si un vuelo es de una determinada fecha, se definiría:

```
	Predicate<Vuelo> fechaIgual(Fecha f){
		return x -> x.getFecha().equals(f);
	}
```

Y después se invocaría:

```
	public Long getNumVuelosDia(Fecha f {
		return vuelos.stream().
			filter(fechaIgual(f))
			count();
	}
```

También se puede combinar con métodos como findFirst que devuelve el primer objeto de un stream o limit(long) que devuelve un stream limitado al número de elementos que se recibe como argumento.

2. allMatch(Predicate)

Nos devuelve un valor de cierto si todos los elementos de una colección cumplen una condición. Por ejemplo si nos preguntamos si todos los vuelos de una fecha están completos:

```
	public Boolean todosCompletos(Fecha f){
		return vuelos.stream().
			filter(fechaIgual(f)).
			allMatch(x->x.getNumPasajeros().equals(x.getNumPlazas()));
	}
```

En este ejemplo el Predicate del filter se podía haber añadido al Predicate del método allMatch con un método and del tipo Predicate.

```
	Predicate<Vuelo> vueloCompleto = x->
				x.getNumPasajeros().equals(x.getNumPlazas());

	public Boolean todosCompletos(Fecha f){
		return vuelos.stream().
				allMatch(fechaIgual(f).and(vueloCompleto()));
	}
```

3. anyMatch(Predicate)

Nos devuelve un valor de cierto si algún elemento de una colección cumple una condición. Por ejemplo si nos preguntamos si hay algún vuelo a un destino en una fecha concreta:

```
	public Boolean hayVueloDestinoFecha(Fecha f, String d){
		return vuelos.stream().
			anyMatch(x->x.getFecha().equals(f) &&
					x.getDestino().equals(d));
	}
```

O si hay algún vuelo no completo un determinado día, definiendo previamente los objetos Predicate fechaIgual y vueloCompleto:

```
	public Boolean hayVueloFechaNoCompleto(Fecha f){
		return vuelos.stream().
			anyMatch(fechaIgual(f).and(vueloCompleto().negate()));
	}
```

4. min(Comparator)

Devuelve el mínimo elemento de una colección según el orden establecido por un Comparator. Para definir el Comparator de un tipo por alguno de sus atributos es muy útil el método comparing de la clase Comparator. Por ejemplo, el vuelo más barato a un determinado destino se calcula comparando dos objetos de tipo Vuelo por su precio usando la Function Vuelo::getPrecio:

```
public Vuelo getVueloMasBaratoDestino(String d){
	return vuelos.stream().
		filter(x->x.getDestino().equals(d)).
		min(Comparator.comparing(Vuelo::getPrecio)).
		get();
}
```

La clase Comparator permite especificar el tipo que se está comparando. Así en el ejemplo anterior se debería hacer usado el método comparingDouble ya que ese es el tipo devuelto por getPrecio. Igualmente existen los métodos comparingInt o comparintLong. También se puede usar el orden natural de Vuelo (dado por su compareTo) mediante la invocación:

```
		min(Comparator.naturalOrder());
```

Hay que tener en cuenta que el método min devuelve lo que en Java 8 se denomina un tipo opcional, ya que podría no existir, si por ejemplo, estuviéramos calculando el mínimo de una colección vacía que podría suceder si en el ejemplo anterior no hubiera ningún vuelo al destino que se pasa como argumento. Por eso se invoca después el método get, que devolvería el objeto menor si existe y sino lanza la excepción NoSuchElementException. Esta posibilidad puede ser controlada con otros métodos disponibles después de invocar a min. Como por ejemplo el método ifPresent que nos devuelve true si efectivamente existe mínimo orElse que permite crear y devolver un objeto si el mínimo no existe.

5. max(Comparator)

Similar al método anterior pero con el máximo en vez del mínimo. Por ejemplo, si quisiéramos devolver el Vuelo de mayor ocupación un determinado día, implementaríamos el método:

```
public Vuelo getVueloMayorOcupacion(Fecha f) {
	return vuelos.stream().
		filter(x->x.getFecha().equals(f)).
		max(Comparator.comparingDouble(
				x->x.getNumPasajeros()/x.getNumPlazas())).
		get();
}
```

En el caso anterior la FunctionDouble se ha definido como argumento del método comparingDouble. Otra solución distinta hubiera sido si Vuelo tuviera un método que devolviera su ocupación. Entonces el código quedaría:

```
public Vuelo getVueloMayorOcupacion(Fecha f) {
	return vuelos.stream().
		filter(x->x.getFecha().equals(f)).
		max(Comparator.comparingDouble(Vuelo::getOcupacion))
		get();
}
```

Lógicamente si el tipo Vuelo no tuviera esa funcionalidad se podría definir una Function para ello:

```
Function<Vuelo, Double> getOcupacion = x->
                                1.*x.getNumPasajeros()/x.getNumPlazas();
```

O una ToDoubleFunction:

```
ToDoubleFunction<Vuelo> getOcupacion = x->
                                1.*x.getNumPasajeros()/x.getNumPlazas();
```

Quedando entonces el método

```
public Vuelo getVueloMayorOcupacion(Fecha f) {
	return vuelos.stream().
		filter(x->x.getFecha().equals(f)).
		max(Comparator.comparingDouble(getOcupacion))
		get();
} //getOcupacion se refiere a la definida como ToDoubleFunction
// para usar getOcupacion como Function debería invocarse desde comparing
```

6. map(Function)

Este método es realmente una familia de métodos con sus adaptaciones mapToInt (ToIntFunction), mapToLong (ToLongFunction) y mapToDouble (ToDoubleFunction) devolviendo un Stream de objetos de otro tipo obtenidos a partir del tipo base aplicando una Function. Si el tipo devuelto es Integer, Long o Double existen los métodos específicos con las correspondientes funciones como argumentos. Así si en el ejemplo anterior no nos interesara cuál es el Vuelo más completo sino cuál es la ocupación máxima, se podría implementar el método:

```
public Double getMayorOcupacion(Fecha f) {
	return vuelos.stream().
		filter(x->x.getFecha().equals(f)).
		mapToDouble(x->x.getNumPasajeros()/x.getNumPlazas()).
		max().
		getAsDouble();
}
```

O usando la ToDoubleFunction anterior:

```
public Double getMayorOcupacion(Fecha f) {
	return vuelos.stream().
		filter(x->x.getFecha().equals(f)).
		mapToDouble(getOcupacion()).
		max().getAsDouble();
}
```

Los métodos map también nos permiten facilitar una serie de operaciones sobre los valores devueltos. Por ejemplo si quisiéramos calcular la recaudación total de los vuelos a un determinado destino escribiríamos:

```
public Double sumaRecaudacionDestino(String d) {
	return vuelos.stream().
		filter(x->x.getDestino().equals(d)).
		mapToDouble(x->x.getNumPasajeros()*x.getPrecio()).
		sum();
}
```

Igual que se calcula la suma se podría devolver la media con el método average. Por ejemplo definiendo una ToDoubleFunction:

```
ToDoubleFunction<Vuelo> getRecaudacion = x->
                                        x.getNumPasajeros()/x.getPrecio();
```

La recaudación media de un determinado día quedaría:

```
	public Double mediaRecaudacionFecha(Fecha f) {
		return vuelos.stream().
				filter(fechaIgual(f)).
				mapToDouble(getRecaudacion).
				average().
				getAsDouble;
	}// average devuelve un OptionalDouble por si no hubiera elementos
```

7. reduce(T, BiFunction)

El método reduce produce una “reducción” de un stream, mediante la operación dada por la Bifunction usando como elemento neutro el primer argumento. Las operaciones sum o average vistas anteriormente son casos particulares de reduce para tipos numéricos. Para otros tipos creados por el programador como Duracion, con el método reduce se puede sumar las duraciones de los vuelos de un determinado día:

Para ello el Tipo Duracion tiene definido un método suma capaz de devolver la suma de dos objetos.

```
public Duracion suma(Duracion d) {
	Integer min = getMinutos() + d.getMinutos();
	Integer hor = getHoras() + d.getHoras();
	return new DuracionImpl(hor+min/60,min%60);
}
```

El método reduce se invoca con dos argumentos: el primero es el elemento neutro que inicializa el acumulador y el segundo la operación que acumula:

```
public Duracion getDuracionVuelosFecha(Fecha f){
	return vuelos.stream().
		filter(x->x.getFecha().equals(f)).
		map(Vuelo::getDuracion).
		reduce(new DuracionImpl(0,0), Duracion::suma);
```

```
}
```

Otra forma de hacerlo es definir la BiFunction en el código. En este caso como los tres tipos que intervienen son de tipo Duracion es un caso particular de BinaryOperator que se podría definir:

```
	BinaryOperator<Duracion> sumadur = (x,y)-> x.suma(y);
```

Incluso si el tipo Duracion no implementara la funcionalidad suma se podría hacer en el código:

```
BinaryOperator<Duracion> sumadur = (x,y)-> {
	Integer min = x.getMinutos() + y.getMinutos();
	Integer hor = x.getHoras() + y.getHoras();
	return new DuracionImpl(hor+min/60,min%60);
};
```

De esta manera la llamada al método reduce quedaría:

```
	reduce(new DuracionImpl(0,0), sumadur);
```

8. forEach(Consumer)

El método forEach sirve para llevar a cabo una acción definida por el objeto Consumer que se pasa como argumento sobre todos los elementos del stream. Por ejemplo para cambiar las duraciones de los vuelos a un determinado destino en un número determinado de minutos, se construiría un método como el siguiente:

```
public void incrementaDuracionDestino(String d, Integer min){
	vuelos.stream().
	filter(x->x.getDestino().equals(d)).
	forEach(x->x.setDuracion(x.getDuracion().suma(new
                                                DuracionImpl(0,min))));
}
```

O para incrementar los precios de los vuelos un 10% a partir de un día determinado:

```
Consumer<Vuelo> incrementaPrecio10p =  x->x.setPrecio(x.getPrecio()*1.1);

public void incrementaPrecios10pAPartirFecha(Fecha f) {
	vuelos.stream().
	filter(x->x.getFecha().compareTo(f)>0).
	forEach(incrementaPrecio10p);
}
```

También serviría para hacer un método estático que escriba en un fichero de texto los elementos de un stream uno por línea:

```
public static <T> void escribeFichero(Stream<T> it, String filename){
	File file = new File(filename);
	try {
		PrintWriter ps = new PrintWriter(file);
		it.forEach(x->ps.println(x));
```

```
			ps.close();
		} catch (FileNotFoundException e) {
			System.out.println("Fichero no encontrado "+filename);
		}
}
```

9. sorted(Comparator)

El método sorted devuelve el stream ordenado por el Comparator que se pasa como argumento. Sin el argumento el orden sería el natural. Por ejemplo, reutilizando el método estático anterior para escribir un Stream en un fichero, podríamos implementar el siguiente método que escribe un Stream ordenado por Fecha y Duracion.

```
	public void escribeVuelosOrdenadosFechaDuracion (String fileName){
		Util.escribeFichero(vuelos.stream().
			sorted(Comparator.comparing(Vuelo::getFecha).
			   thenComparing(Vuelo::getDuracion)), fileName);
	}
```

10. collect(Collector)

La funcionalidad collect es un potente mecanismo para reducir un stream en otra colección, estructura Map o dato. Alguna de estas funcionalidades puede ser realizada por métodos vistos anteriormente, como reduce. Como las posibilidades son muy numerosas vamos a dividirlas según el objetivo.

10.1 Reducción a List o Set. El método collect proporciona un mecanismo muy potente para transformar un stream en distinto tipo de colecciones de datos como List y Set. Por ejemplo, si se quisiera devolver una lista con las duraciones de los vuelos a un determinado destino, se escribiría:

```
public List<Duracion> getDuracionesDestino(String d){
	return vuelos.stream().
		filter(x->x.getDestino().equals(d)).
		map(x->x.getDuracion()).
		collect(Collectors.toList());
}
```

La clase Collectors proporciona dos métodos toList y toSet de forma que el Stream de objetos Duracion devuelto por la operación map puede ser convertida en una lista o un conjunto respectivamente. Por ejemplo, el conjunto de los destinos posibles desde un aeropuerto viene dado por el método:

```
public Set<String> getDestinos(){
	return vuelos.stream().
		map(x->x.getDestino()).
		collect(Collectors.toSet());
```

```
}
```

Los métodos toList y toSet son casos particulares del método toCollection que recibe un argumento de tipo Supplier con, por ejemplo, una invocación al constructor del tipo. De esta manera podemos devolver un SortedSet con los destinos de un determinado día:

```
public SortedSet<String> getDestinosEnFecha(Fecha f){
	return vuelos.stream().
		filter(x->x.getFecha().equals(f)).
		map(x->x.getDestino()).
		collect(Collectors.toCollection(TreeSet::new));
}
```

También se pueden transformar los objetos antes de su recolección. Por ejemplo, si quisiéramos obtener una lista con las duraciones de los vuelos a un determinado destino incrementadas en m minutos, se implementaría el siguiente método:

```
public List<Duracion> getDuracionesIncrementadasDestino(String d, Integer m){
	return vuelos.stream().
		filter(x->x.getDestino().equals(d)).
		map(x->x.getDuracion().suma(new DuracionImpl(0,m))).
		collect(Collectors.toList());
}
```

10.2 Estadísticas. Si solo quisiéramos contar el número de elementos del stream se puede usar también Collectors.counting(). Igualmente si el stream es de datos numéricos se pueden usar los colectores Collectors.summingDouble (summingInt o summingLong) para devolver la suma del stream en el mismo tipo y Collectors.averagingDouble (averagingInt o averagingLong) que devuelven la media aritmética siempre en tipo Double.

También existe el tipo DoubleSummaryStatistics a partir del cual es posible obtener los valores de la suma, máximo, mínimo y media de una serie de valores numéricos. Un objeto de este tipo es devuelto por Collectors.summarizingDouble que recibe una ToDoubleFunction (también existen igualmente summarizingInt y summarizinLong). Por ejemplo, para obtener la media de los precios de los vuelos a un determinado destino pondríamos:

```
public Double getMediaPreciosDestino(String d){
	return vuelos.stream().
		filter(x->x.getDestino().equals(d)).
		collect(Collectors.summarizingDouble(Vuelo::getPrecio)).
		getAverage();
}
```

Teniendo en cuenta que la varianza de una muestra es la media de los valores al cuadrado menos la media aritmética al cuadrado, se podría calcular la varianza de todos los precios del aeropuerto con el siguiente método:

```
public Double getVarianzaPrecios(){
	Double medcuad = vuelos.stream().
			collect(Collectors.summarizingDouble(
					x->x.getPrecio()*x.getPrecio())).
```

```
                getAverage();
        Double media = vuelos.stream().
                collect(Collectors.summarizingDouble(Vuelo::getPrecio)).
                getAverage();
        return medcuad-media*media;
        }
```

Cuando no podemos emplear las operaciones aritméticas habituales de suma o media porque el tipo base no es un tipo numérico (Double, Integer, etc), se debe emplear la funcionalidad reducing de Collectors, para implementar nuevas operaciones sobre este tipo base. Por ejemplo si quisiéramos devolver la suma de las duraciones de todos los vuelos a un destino incrementadas en m minutos:

```
public Duracion incrementaySumaDuracionDestino(String d, Integer m){
        return vuelos.stream().
                filter(x->x.getDestino().equals(d)).
                map(x->x.getDuracion().suma(new DuracionImpl(0,m))).
                collect(Collectors.reducing(
                                new DuracionImpl(0,0), Duracion::suma));
}
```

10.3 Reducción a Map. La clase Collectors proporciona los métodos partitioningBy y groupingBy para organizar la información de un stream en un Map. Las posibilidades son numerosas ya que groupingBy puede recibir distintos argumentos.

10.3.1 groupingBy(Function) y partitioningBy(predicate). En este primer caso, la funcionalidad groupingBy recibe una Function, mientras que partitioningBy es un caso particular en el que la Function es sustituida por un Predicate. En este caso, el conhunto final es un List con los objetos del stream. Veamos distintos ejemplos de uso, si se quisiera construir un Map<Boolean, List<Vuelo>> para clasificar los vuelos del aeropuerto en completos o no, escribiríamos usando el Predicate declarado en el punto 2:

```
public Map<Boolean, List<Vuelo>> getMapVuelosCompletos(){
        return vuelos.stream().
                collect(Collectors.partitioningBy(vueloCompleto()));
}
```

Un poco más complejo es si quisiéramos obtener un Map<Boolean,List<Vuelo>> para clasificar los vuelos según hubieran pasado o no un determinado umbral de ocupación que se pasa como argumento:

```
public Map<Boolean, List<Vuelo>> getMapVuelosPorcentaje(Double p){
        return vuelos.stream().
                collect(Collectors.partitioningBy(
                                x->x.getOcupacion().compareTo(p/100)>=0));
}
```

Para obtener un Map con más de dos valores en el conjunto inicial hay que recurrir a groupingBy. De esta manera para obtener un Map que organize los vuelos por fecha, escribiríamos:

```
public SortedMap<Fecha,List<Vuelo>> getMapVuelosXFecha(){
	return vuelos.stream().
		collect(Collectors.groupingBy(Vuelo::getFecha));
}
```

10.3.2 GroupingBy(Function, Collector). Por supuesto es posible que el tipo del conjunto final del Map no sea un List de objetos del stream sino que también podría ser un Set, pasando como argumentos a groupingBy además de la Function para el conjunto inicial un Collector para el conjunto final:

```
public Map<String, Set<Vuelo>> getMapSetVuelosXDestino(){
	return vuelos.stream().
		collect(Collectors.groupingBy(Vuelo::getDestino,
							Collectors.toSet()));
}
```

Si la información del conjunto final es una reducción u operación sobre los objetos del stream es necesario acudir a otros objetos de tipo Collector para pasarle como argumento a groupingBy. De esta forma, usando los métodos de Collectors counting, averaging o summing se puede obtener una reducción del tipo del conjunto final del Map. Así para obtener el número de vuelos por destino el método sería:

```
public Map<String, Long> getMapNumVuelosXDestino(){
	return vuelos.stream().
		collect(Collectors.groupingBy(Vuelo::getDestino,
					Collectors.counting()));
}
```

Igualmente si se quiere devolver un Map con los precios medios por destino:

```
public Map<String, Double> getMapPrecioMedioXDestino(){
	return vuelos.stream().
		collect(Collectors.groupingBy(Vuelo::getDestino,
			Collectors.averagingDouble(Vuelo::getPrecio)));
}
```

O un Map con la recaudación por Fecha:

```
public Map<Fecha, Double> getMapRecaudacionXFecha(){
	return vuelos.stream().
		collect(Collectors.groupingBy(
			Vuelo::getFecha,
			Collectors.summingDouble(
			(Vuelo x)->x.getNumPasajeros()*x.getPrecio())));
}
```

Que se podría escribir más sencillo usando la ToDoubleFunction getRecaudacion del punto 6:

```
public Map<Fecha, Double> getMapRecaudacionXFecha(){
	return vuelos.stream().
		collect(Collectors.groupingBy(Vuelo::getFecha,
```

```
                                    Collectors.summingDouble(getRecaudacion())));
}
```

Otras veces se requiere que el tipo del conjunto final sea una transformación del tipo del stream. En ese caso el Collector que se pasa como segundo argumento de groupingBy es un Collectors.mapping que debe recibir una Function y otro Collector. Por ejemplo, para construir un Map con destinos en el conjunto inicial y una lista de los precios por destino en el conjunto final se escribiría:

```
public Map<String, List<Double>> getMapPreciosXDestino(){
	return vuelos.stream().
		collect(Collectors.groupingBy(Vuelo::getDestino,
					Collectors.mapping(Vuelo::getPrecio,
					Collectors.toList())));
	}
```

10.3.3 GroupingBy(Function, Supplier, Collector). Proporcionar un Supplier permite, entre otras posibilidades devolver SortedMap como resultado de groupingBy. De esta manera, para devolver un SortedMap con una lista de vuelos por Fecha, escribiríamos:

```
public SortedMap<Fecha,List<Vuelo>> getSortedMapVuelosXFecha(){
	return vuelos.stream().
		collect(Collectors.groupingBy(Vuelo::getFecha,
						TreeMap::new,
						Collectors.toList()));
}
```

Un ejercicio más complicado es invertir un Map. Es decir, si tenemos un Map<K,V> obtener un Map<V,List<K>> o Map<V,Set<K>> donde los elementos de los conjuntos inicial y final del Map original han intercambiado sus papeles en el Map inverso. Lógicamente como la aplicación original puede ser sobreyectiva es necesario asignar en el Map inverso a cada elemento del conjunto inicial varios del conjunto inicial. Por ejemplo, hemos hallado anteriormente un Map<String, Long> con el número de vuelos por destino. Si quisiéramos saber el destino con más vuelos, una solución sería invertir el Map anterior en un SortedMap<Long,List<String>> de manera que la clave mayor nos proporciona la lista de destinos con mayor número de vuelos. En Java 7 esté método invierteMap se podía escribir como método estático así:

```
public static <T,K> SortedMap<T, List<K>> invierteMap(Map<K, T> m) {
	SortedMap<T, List<K>> res = new TreeMap<T,List<K>>();
	Set<K> sp = m.keySet();
	for(K elem: sp){
			T val =m.get(elem);
			if (res.containsKey(val)) {
				res.get(val).add(elem);
			}
			else {
				List<K> lista = new LinkedList<K>();
				lista.add(elem);
				res.put(val, lista);
			}
	}
```

```
        return res;
}
```

En Java 8 se usaría un código más compacto, dejando al lector si más fácil o no de entender. Se trata de convertir el Map original en un stream mediante el método entrySet que devuelve el conjunto de pares clave-valor. Una vez que tenemos el Map convertido en un stream de pares, se invoca a collect agrupando los pares por los valores (es decir, los valores pasan a ser las claves del nuevo Map) y haciendo que en el conjunto final estén List de las claves originales mediante el mapping correspondiente. El constructor TreeMap nos permite que el resultado sea un Sortedmap:

```
public static <K,T> SortedMap<K, List<T>> invierteMapAList(Map<T, K> m){
        return m.entrySet().stream().
                collect(Collectors.groupingBy(
                                Map.Entry<T,K>::getValue,
                                TreeMap::new,
                                Collectors.mapping(Map.Entry<T,K>::getKey,
                                                Collectors.toList())));
}
```

Cambiando el toList() final por un toSet() obtendríamos el método invierteMapASet. Una aplicación inmediata de esta estructura de código para el caso particular de obtener el número de vuelos relacionado con el conjunto de destinos es el siguiente método:

```
public SortedMap<Long, Set<String>> getMapDestinosXNumVuelos(){
        return getMapNumVuelosXDestino().
                entrySet().stream().
                Collect(Collectors.groupingBy(
                                Map.Entry<String,Long>::getValue,
                                TreeMap::new,
                                Collectors.mapping(
                                                Map.Entry<String,Long>::getKey,
                                                Collectors.toSet())));
}
```

De esta manera responder a la pregunta cuáles son los destinos con mayor número de vuelos se resuelve de la siguiente manera:

```
public Set<String> getDestinosMasVuelos(){
        SortedMap<Long,Set<String>> m = getMapDestinosXNumVuelos());
        return m.get(m.lastKey());
}
```

O empleando el método estático invierteMapASet:

```
public Set<String> getDestinosMasVuelos(){
SortedMap<Long,Set<String>>m=Util.invierteMapASet(getMapNumVuelosXDestino());
return m.get(m.lastKey());
}
```

Una versión más simple de este problema se puede resolver si sabemos que sólo hay un máximo y por tanto no es necesario devolver un Set con los posibles valores donde se da. Por ejemplo, si sólo hubiera un destino con el máximo de vuelos una solución sería partir del Map

que asigna a cada destino su número de vuelos, convertir a stream su conjunto de pares valor, y después obtener el máximo mediante un comparator sólo con los valores y devolver el par que le corresponde al máximo mediante get y la clave de ese par mediante getKey:

```
public String getUnDestinoMasVuelos(){
	return getMapNumVuelosXDestino().
		entrySet().stream().
		max(Comparator.comparing(x->x.getValue())).
		get().
		getKey();
}
```