Tema 8. Tratamientos Secuenciales II

Autores: Miguel Toro, José Riquelme y Mariano González
Revisión: profesores de la asignatura
Tiempo estimado: 8 horas

1. Comparadores

Resumen: en este apartado se introduce la interfaz Comparator como forma de implementar un orden alternativo sobre un tipo. Se muestran varias opciones para declarar objetos de tipo Comparator, y se realizan varios ejemplos de uso de comparadores con los métodos de Collections y con los tipos SortedSet y SortedMap. Además, se introducen las interfaces funcionales y las expresiones lambda.

Nota: en los ejemplos de este tema utilizaremos el tipo Vuelo, cuya definición es la siguiente:

```
public interface Vuelo extends Comparable<Vuelo> {
	String getCodigo();
	String getOrigen();
	String getDestino();
	void setDestino(String destino);
	LocalDate getFechaSalida();
	void setFechaSalida(LocalDate fsal);
	LocalDate getFechaLlegada();
	void setFechaLlegada(LocalDate flleg);
	Integer getNumPlazas();
	Integer getNumPasajeros();
	List<Pasajero> getPasajeros();
}
```

1.1 Orden natural

Hemos visto cómo definir un orden natural en un tipo. La interfaz del tipo T debe extender a la interfaz `Comparable<T>`, y en la clase hay que implementar el método `compareTo`.

Veamos como ejemplo el tipo Vuelo. Los vuelos se ordenan de forma natural por su código y, a igualdad de éste, por su fecha de salida. Para implementar este orden hacemos lo siguiente:

En la interfaz:

```
public interface Vuelo extends Comparable<Vuelo> {
	...
}
```

En la clase:

```
public int compareTo(Vuelo v) {
	int res = getCodigo().compareTo(v.getCodigo());
	if (res == 0) {
		res = getFechaSalida().compareTo(v.getFechaSalida());
	}
	return res;
}
```

Una vez definido el orden natural, si queremos ordenar una lista de vuelos según este orden podemos hacer lo siguiente:

```
List<Vuelo> vuelos = new LinkedList<Vuelo>();
...
Collections.sort(vuelos);
System.out.println("Vuelos por orden de código y fecha de salida: " + vuelos);
```

1.2 Órdenes alternativos

Muchas veces necesitamos ordenar por otros criterios distintos al orden natural. Por ejemplo, nos puede interesar ordenar vuelos por su destino, su fecha de salida o su porcentaje de ocupación.

El método `sort` permite especificar una forma alternativa de ordenación, añadiendo un segundo parámetro a la llamada. Por ejemplo, para ordenar los vuelos por su fecha de salida, haríamos lo siguiente:

```
Collections.sort(vuelos, orden alternativo por fecha de salida);
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

¿Cómo le indicamos al método `sort` que debe ordenar los vuelos por su fecha de salida? Lo hacemos pasando como parámetro un objeto que representa un orden sobre el tipo, en este caso el orden según la fecha de salida.

Estos objetos que representan órdenes se denominan comparadores, porque implementan la interfaz de Java `Comparator`. El tipo asociado a esta interfaz se puede representar de la forma **T x T -> int**, ya que un objeto comparator toma dos objetos de un mismo tipo y devuelve un valor de tipo int que indica la relación de orden entre ambos objetos.

Por tanto hay que definir un objeto de tipo `Comparator` y pasarlo como parámetro al método:

```
Comparator<Vuelo> comparadorVueloFechaSalida = . . .
Collections.sort(vuelos, comparadorVueloFechaSalida);
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

Hay varias formas de definir objetos de tipo `Comparator`. Veamos tres de ellas, que serán las que utilizaremos en nuestra asignatura. Pero antes, hablemos un poco más de la interfaz `Comparator` y, generalizando, de lo que se conoce como interfaces funcionales.

1.3 Interfaces funcionales. La interfaz Comparator

La interfaz `Comparator` es una **interfaz funcional**, ya que mediante ella creamos un tipo que representa una función. En este caso, cuando construimos un objeto de tipo `Comparator`, lo que estamos creando es una función para comparar dos objetos. Y esta función se puede pasar como parámetro a ciertos métodos, como es el caso del método `sort`. Este método espera como segundo parámetro un tipo funcional que le indique cómo debe ordenar los elementos de la lista. El objeto comparator cumple con este requisito.

Existen más interfaces funcionales en Java, que iremos viendo poco a poco. Pero antes de saber qué requisito debe cumplir una interfaz funcional, es preciso conocer dos novedades importantes de las interfaces en Java 8.

La primera novedad es que en Java 8 las interfaces pueden contener métodos de tipo `static`. Estos métodos se implementan en la propia interfaz, y pueden ser invocados desde las clases que implementan la interfaz.

La segunda novedad de Java 8 es que permite incluir en una interfaz implementaciones por defecto (`default`) para los métodos. Si una clase que implementa la interfaz no redefine este método, utilizará la implementación por defecto.

Pues bien, las interfaces funcionales son aquellas que sólo incluyen un método que no es de tipo `static` o `default`. Es el caso de la interfaz `Comparator`. Esta interfaz tiene, entre otros, los siguientes métodos:

```
public interface Comparator<T> {
      int compare(T o1, T o2);
      static <T,U extends Comparable<? super U>>
         Comparator<T> comparing(Function<? super T,? extends U> keyExtractor);
      static <T extends Comparable<? super T>> Comparator<T> naturalOrder();
      default Comparator<T> reversed();
      static <T extends Comparable<? super T>> Comparator<T> reverseOrder();
      default Comparator<T> thenComparing(Comparator<? super T> other);
      . . .
}
```

En el resto del tema iremos viendo qué hacen y cómo se utilizan algunos de estos métodos.

1.4 Comparadores. Definición mediante una clase que implementa la interfaz Comparator

La opción clásica en Java, hasta la introducción de la versión Java 8, es crear una clase que implemente la interfaz `Comparator`. Esta clase implementará el método `compare` de la interfaz:

```
int compare(T o1, T o2);
```

Este método recibe dos objetos y compara los valores de la propiedad o propiedades según las cuales queremos ordenar los objetos. El valor devuelto es similar al devuelto por el método `compareTo` ya visto: negativo si el primer objeto es anterior al segundo, 0 si están en el mismo orden y positivo si es posterior.

En nuestro ejemplo, creamos una clase `ComparadorVueloFechaSalida` que implementa la interfaz Comparator. La clase tiene un método `compareTo` que recibe dos objetos de tipo Vuelo y devuelve un valor que es < 0 si el primer vuelo tiene una fecha de salida anterior al segundo, 0 si ambos salen en la misma fecha y > 0 si el primero tiene una fecha de salida posterior:

```
public class ComparadorVueloFechaSalida implements Comparator<Vuelo> {
	public int compare(Vuelo v1, Vuelo v2) {
		int res = v1.getFechaSalida().compareTo(v2.getFechaSalida());
		return res;
	}
}
```

Una vez creada la clase, podemos utilizarla para crear el objeto de tipo `Comparator` y usarlo en la llamada al método `sort`:

```
Comparator<Vuelo> comparadorVueloFechaSalida = new ComparadorVueloFechaSalida();
Collections.sort(vuelos, comparadorVueloFechaSalida);
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

1.5 Comparadores. Definición mediante el método comparing

La forma más simple de crear un objeto comparator es indicar directamente la propiedad según la cual queremos ordenar los objetos. Para ello usamos un método de la interfaz `Comparator`, el método `comparing`. En el ejemplo de la ordenación de vuelos por su fecha de salida se haría de la siguiente manera:

```
Comparator<Vuelo> comparadorVueloFechaSalida =
	Comparator.comparing(Vuelo::getFechaSalida);
Collections.sort(vuelos, comparadorVueloFechaSalida);
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

Fíjese que el método `comparing` es un método `static`, de ahí que haya que invocarlo anteponiendo a su nombre el de la interfaz `Comparator`.

Por otro lado, la expresión

```
Vuelo::getFechaSalida
```

Se conoce en Java como **referencia a método**, y toma la forma **T :: m**, donde **T** es un tipo y **m** el nombre de un método. Por ejemplo, Vuelo::getFechaSalida para hacer referencia a la fecha de salida del vuelo, Vuelo::getDestino para hacer referencia al destino del vuelo, o Vuelo::new cuando queremos referirnos al constructor del tipo.

En este caso, la expresión que indica el orden que deseamos tiene la forma <Nombre del tipo>::<Nombre del método>, siendo el método el getter que devuelve la propiedad por la cual queremos ordenar.

Esto se puede hacer con cualquier propiedad del tipo. Por ejemplo, para ordenar los vuelos por su destino sería:

```
Comparator<Vuelo> comparadorVueloDestino =
	Comparator.comparing(Vuelo::getDestino);
```

Como vemos, definir el orden es tan simple como indicar la propiedad por la cual queremos ordenar. Claro que esto no siempre podremos hacerlo. Por ejemplo, ¿qué pasa si queremos ordenar los vuelos por su porcentaje de ocupación? Como no existe una propiedad que nos dé directamente esta información, no podemos usar la forma anterior. No obstante, hay una alternativa para indicar el orden en la llamada al método `comparing`. Se trata de pasar como parámetro una expresión que indique el orden. Se haría de la siguiente forma:

```
Comparator<Vuelo> comparadorVueloOcupacion =
      Comparator.comparing(x -> 100. * x.getNumPasajeros() / x.getNumPlazas());
Collections.sort(vuelos, comparadorVueloOcupacion);
System.out.println("Vuelos por orden de % de ocupación: " + vuelos);
```

La expresión

```
x -> 100. * x.getNumPasajeros() / x.getNumPlazas()
```

se conoce como **expresión lambda**, y viene a ser una simplificación de un método en el que el parámetro de entrada y la expresión que produce el valor de salida se separan por el operador flecha '->'. Sería equivalente al método

```
public Double getPorcentajeOcupacion() {
      return 100. * getNumPasajeros() / getNumPlazas();
}
```

pero con la ventaja de que puede utilizarse como parámetro en la llamada a un método que espera un tipo funcional, como es el caso del método `comparing`.

En una expresión lambda normalmente no es necesario definir los tipos de los argumentos porque Java es capaz de deducirlos del contexto en el que se definen. Así, en este caso la variable x se toma como un objeto de tipo Vuelo.

Como puede imaginar, también podemos usar una expresión lambda para ordenar los vuelos por su fecha de salida, aunque ya exista una propiedad para ello. Se haría así:

```
Comparator<Vuelo> comparadorVueloFechaSalida =
      Comparator.comparing(x -> x.getFechaSalida());
Collections.sort(vuelos, comparadorVueloFechaSalida);
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

En este caso la expresión lambda es:

```
x -> x.getFechaSalida()
```

1.6 Comparadores. Definición mediante expresiones lambda

Una tercera forma de definir el comparador es utilizar directamente una expresión lambda, sin necesidad de recurrir al método `comparing`. Para el ejemplo de ordenación por la fecha de salida del vuelo, sería de la siguiente forma:

```
Comparator<Vuelo> comparadorVueloFechaSalida =
      (x, y) -> x.getFechaSalida().compareTo(y.getFechaSalida());
Collections.sort(vuelos, comparadorVueloFechaSalida);
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

La expresión lambda es:

```
(x, y) -> x.getFechaSalida().compareTo(y.getFechaSalida());
```

En este caso tiene dos parámetros, x e y, que representan los dos objetos que hay que comparar. Al ser dos, deben ir entre paréntesis y separados por una coma. La expresión a la derecha de la flecha es la que determina el orden de los objetos, y para ello se hace una llamada al método `compareTo` con la propiedad que determina el orden, la fecha de salida.

Es posible escribir directamente la expresión en la llamada al método `sort`, sin necesidad de declarar previamente la variable comparator. Quedaría así:

```
Collections.sort(vuelos,
      (x, y) -> x.getFechaSalida().compareTo(y.getFechaSalida()));
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

Veamos otro ejemplo. Si queremos ordenar vuelos por su destino, sería:

```
Collections.sort(vuelos, (x, y) -> x.getDestino().compareTo(y.getDestino()));
```

1.7 Combinación de comparadores

En ocasiones nos interesará ordenar según un criterio y, a igualdad de éste, según un segundo criterio. Por ejemplo, queremos ordenar los vuelos por fecha de salida, y a igualdad de ésta por su destino. Para ello creamos dos objetos de tipo comparator y los combinamos usando el método `thenComparing` de la interfaz `Comparator`:

```
Comparator<Vuelo> comparadorVueloFechaSalida =
      Comparator.comparing(Vuelo::getFechaSalida);
Comparator<Vuelo> comparadorVueloDestino =
      Comparator.comparing(Vuelo::getDestino);
Collections.sort(vuelos,
      comparadorVueloFechaSalida.thenComparing(comparadorVueloDestino));
System.out.println("Vuelos por orden de fecha de salida y destino: " + vuelos);
```

1.8 Ordenación inversa

En todos los casos vistos hasta ahora, el orden de los elementos viene dado por el orden natural de la propiedad según la cual estamos comparando. Por ejemplo, si comparamos por fecha de salida, estamos ordenando de menor a mayor fecha, y si comparamos por número de pasajeros estamos ordenando de menor a mayor número de pasajeros. ¿Qué haríamos si queremos obtener una lista en orden inverso, por ejemplo de mayor a menor número de pasajeros? Pues utilizamos el método `reversed` de la interfaz `Comparator`. Sería así:

```
Comparator<Vuelo> comparadorVueloNumPasajeros =
      Comparator.comparing(Vuelo::getNumPasajeros);
Collections.sort(vuelos, comparadorVueloNumPasajeros.reversed());
System.out.println("Vuelos por orden inverso del número de pasajeros: " + vuelos);
```

1.9 Ejemplos de uso de comparadores

Veamos más ejemplos del uso de comparadores.

Ejemplo 1. Obtener el vuelo con más pasajeros:

```
Vuelo masPasajeros = Collections.max(vuelos,
      Comparator.comparing(Vuelo::getNumPasajeros));
System.out.println("Vuelo con más pasajeros: " + masPasajeros);
```

El método `max` obtiene el mayor elemento de una colección según el orden indicado por el comparador que recibe como parámetro. En este caso, nos da el vuelo de la lista con mayor número de pasajeros.

Ejemplo 2. Obtener el vuelo que sale antes:

```
Vuelo primerVuelo = Collections.min(vuelos,
      Comparator.comparing(Vuelo::getFechaSalida));
System.out.println("Vuelo que sale antes: " + primerVuelo);
```

Ejemplo 3. Crear un conjunto de vuelos ordenado por el número de pasajeros:

```
SortedSet<Vuelo> vuelosOrdenados =
      new TreeSet<Vuelo>(Comparator.comparing(Vuelo::getNumPasajeros));
```

En este ejemplo se nos plantea un problema: dos vuelos con el mismo número de pasajeros son iguales según nuestro orden, y por lo tanto no se podrían añadir los dos elementos al `SortedSet`. Para solucionar esto es preciso recurrir al orden natural del tipo en caso de empate por el orden alternativo. Para ello tenemos el método `naturalOrder` de la interfaz `Comparator`, y lo usaríamos de la siguiente forma:

```
SortedSet<Vuelo> vuelosOrdenados =
      new TreeSet<Vuelo>(Comparator.comparing(Vuelo::getNumPasajeros)
            .thenComparing(Comparator.naturalOrder()));
```

Ejemplo 4. Crear un `SortedMap` que relacione los vuelos con su número de pasajeros, ordenado por el destino del vuelo y a igualdad de éste por su orden natural.

```
SortedMap<Vuelo, Integer> pasajerosVuelos = new TreeMap<Vuelo, Integer>(
      Comparator.comparing(Vuelo::getDestino)
            .thenComparing(Comparator.naturalOrder()));
```

2. El tipo Stream

Resumen: en este apartado se explica la forma de realizar tratamientos secuenciales en Java 8. Se introduce el tipo `Stream`, sus métodos y las interfaces funcionales que se utilizan en la llamada a los mismos.

2.1 Streams

Java 8 introduce una nueva y potente forma de realizar operaciones con agregados de datos. Para ello nos proporciona varios elementos que podemos combinar para realizar cualquier tratamiento que necesitemos sobre un agregado de datos.

Un concepto fundamental es el de **Stream**. Un stream es un agregado de elementos. Se podría traducir como 'flujo de datos'. Por ejemplo, tenemos los siguientes streams:

- Vuelos que se dirigen a Londres.
- Libros de tipo Novela y con más de 500 páginas.
- Asignaturas optativas de primer cuatrimestre.

Estos streams se obtienen a partir de una lista o conjunto, es decir, a partir de una fuente real. Pero también se puede obtener un stream a partir de una fuente virtual, como por ejemplo:

- Números primos menores de 1000.
- Palabras de longitud 3.
- Números naturales (stream infinito)

Un stream, en definitiva, representa un agregado de elementos sobre los cuales se pueden realizar tratamientos diversos. En este tema veremos cuáles son esos tratamientos y cómo realizarlos.

2.2 Un primer ejemplo

Vamos a recordar cómo hemos realizado los tratamientos secuenciales hasta ahora, y vamos a ver cómo se hace con el uso de streams. Para ello tomemos un ejemplo.

Supongamos que tenemos una lista de vuelos y queremos obtener el número de ellos que parten en una fecha determinada. Se trata de un tratamiento secuencial de tipo 'contador', tal como vimos en su día. Según el esquema visto para este tratamiento, el problema se resolvería así:

```
public Integer numeroVuelosSalidaFecha(LocalDate fecha) {
	Integer num = 0;
	for (Vuelo v : getVuelos()) {
		if (v.getFechaSalida().equals(fecha)) {
			num++;
		}
	}
	return num;
}
```

Ahora veamos cómo se hace en Java 8:

```
public Long numeroVuelosSalidaFecha(LocalDate fecha) {
// Se declara Long porque es el tipo que devuelve count
	return getVuelos().stream()
		.filter(x -> x.getFechaSalida().equals(fecha))
		.count();
}
```

Analicemos los elementos que vemos aquí.

En primer lugar tenemos una llamada al método stream, que convierte la lista en un objeto de tipo `Stream`. Este será siempre el primer paso que debemos dar para poder aplicar el tratamiento a una colección de elementos.

```
getVuelos().stream()
```

En segundo lugar se aplica una llamada al método `filter`, que filtra los elementos de la lista que queremos contar.

```
.filter(x -> x.getFechaSalida().equals(fecha))
```

En este caso son los vuelos que tienen una fecha de salida determinada. Para indicar esto, utilizamos una expresión lambda.

```
x -> x.getFechaSalida().equals(fecha)
```

Finalmente, se hace una llamada al método count, que cuenta el número de elementos de la lista filtrada.

```
.count();
```

Como vemos, en primer lugar se convierte el agregado en un stream, y a partir de ahí se trata de ir realizando operaciones una tras otra en secuencia sobre el stream. Cada operación se realiza sobre el resultado de la anterior. En el ejemplo, primero se filtran los vuelos con una fecha de salida determinada y luego se cuentan.

2.3 Un segundo ejemplo

Veamos un nuevo ejemplo. En este caso, queremos obtener el número de vuelos que tienen un destino concreto. Nuevamente se trata de un tratamiento secuencial de tipo ‘contador’. Según el esquema visto para este tratamiento, sería así:

```
public Integer numeroVuelosDestino(String destino) {
	Integer num = 0;
	for (Vuelo v : getVuelos()) {
		if (v.getDestino().equals(destino)) {
			num++;
		}
	}
	return num;
}
```

Ahora veamos cómo sería en Java 8:

```
public Long numeroVuelosDestino(String destino) {
	return getVuelos().stream()
		.filter(x -> x.getDestino().equals(destino))
		.count();
}
```

Se podrían hacer más operaciones en cascada sobre el stream. Por ejemplo, filtrar los vuelos que salen en una fecha, a continuación quedarse con los que tienen un destino determinado, y por último contarlos. Así podríamos responder por ejemplo a la pregunta ¿Cuántos vuelos salen hoy hacia París?

```
public Long numeroVuelosSalidaFechaYDestino(LocalDate fecha, String destino) {
	return getVuelos().stream()
		.filter(x->x.getFechaSalida().equals(fecha))
		.filter(x->x.getDestino().equals(destino))
		.count();
}
```

Ejercicio: responder a la pregunta ¿Cuántos vuelos con plazas libres salen hoy hacia París? Solución: filtrar los vuelos que salen en una fecha, a continuación quedarse con los que tienen un destino determinado, luego descartar los que están completos, y por último contarlos.

El esquema es pues siempre el mismo. Dependiendo de la operación que queramos hacer utilizaremos los métodos correspondientes, que iremos viendo a lo largo del tema.

2.4 El tipo Stream

Según hemos visto, un stream es un agregado de elementos, que puede ser real o virtual.

En Java 8 existe el tipo `Stream<T>` para representar estos agregados. Este tipo soporta operaciones secuenciales (y también en paralelo, aunque esto no se verá en este curso). Por ejemplo, `filter` y `count`, vistas anteriormente.

Para obtener un stream a partir de un conjunto o lista se utiliza el método `stream` de la interfaz `Collection`. Por ejemplo:

```
List<Vuelo> vuelos = new LinkedList<Vuelo>();
Stream<Vuelo> s = vuelos.stream();
```

También se puede construir un stream a partir de varios objetos. Por ejemplo, dados tres vuelos v1, v2 y v3, podemos crear un stream con ellos de la siguiente forma:

```
Stream<Vuelo> vuelos = Stream.of(v1, v2, v3);
```

Un stream se puede manipular con una serie de métodos para realizar diferentes operaciones sobre él, que incluyen todos los tratamientos secuenciales que hemos visto y algunas operaciones más.

2.5 Operaciones sobre streams

Las operaciones sobre streams pueden ser de dos tipos: operaciones intermedias y terminales. Las operaciones intermedias producen un stream a partir de otro. Es el caso, por ejemplo, de la operación `filter`. Las operaciones terminales producen un objeto que no es de tipo stream. Por ejemplo, el número de elementos del stream devuelto por el método `count`.

Hay muchas operaciones aplicables sobre streams. En este tema veremos las siguientes:

- Operaciones intermedias:
  - distinct
  - filter
  - flatMap
  - map
  - sorted

- Operaciones terminales:
  - allMatch
  - anyMatch
  - average
  - collect
  - count
  - forEach
  - max
  - min
  - reduce
  - sum

Cada uno de estos métodos tiene como parámetro un objeto de un tipo funcional. Ya hemos visto uno de estos tipos, `Comparator`. En el resto del tema veremos algunos tipos funcionales más a la vez que se van introduciendo los métodos que los utilizan.

2.6 La interfaz Predicate

En los ejemplos del inicio del tema hemos usado el método `filter`. Este método toma un stream y devuelve otro stream que contiene únicamente los elementos del primer stream que cumplen una determinada condición. La forma de invocarlo es:

```
s.filter(expresión)
```

En el ejemplo vimos cómo obtener el número de vuelos que tienen un destino concreto. El método era el siguiente:

```
public Long numeroVuelosDestino(String destino) {
	return getVuelos().stream()
		.filter(x -> x.getDestino().equals(destino))
		.count();
}
```

La expresión que pasamos como parámetro al método `filter`,

```
x -> x.getDestino().equals(destino)
```

es una expresión que toma valor true si el vuelo x al que se aplica tiene el destino deseado. Por tanto, el método `filter` espera como parámetro una función que devuelve un tipo lógico. Estas funciones que devuelven un valor lógico se conocen como predicados, ya que son objetos que implementan la interfaz `Predicate`:

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

De esta forma, el método anterior se podría reescribir de la siguiente forma:

```
public Long numeroVuelosDestino(String destino) {
	Predicate<Vuelo> vueloDestino = x -> x.getDestino().equals(destino);
	return getVuelos().stream().filter(vueloDestino).count();
}
```

Nótese la similitud con lo que hacíamos en el apartado anterior al definir un comparador. En aquel caso se utilizaba la interfaz `Comparator`, y ahora la interfaz `Predicate`. Ambas son interfaces funcionales, que sirven para definir tipos funcionales que posteriormente se utilizan como parámetros en la llamada a un método.

Cuando un predicado se va a utilizar con frecuencia, puede implementarse en un método independiente. Por ejemplo, en una clase de utilidad `PredicadosVuelo` podríamos tener el siguiente método:

```
public static Predicate<Vuelo> vueloDestino(String destino) {
	return x -> x.getDestino().equals(destino);
}
```

Usando este método, el código quedaría de la siguiente manera:

```
public Long numeroVuelosDestino(String destino) {
	return getVuelos().stream().filter(vueloDestino(destino)).count();
}
```

Los métodos `or`, `and` y `negate` de la interfaz `Predicate` se utilizan para combinar predicados. Por ejemplo, dados los predicados `vueloDestino`, que indica si un vuelo tiene un destino dado; `vueloFecha`, que indica si un vuelo sale en una fecha dada, y `vueloCompleto`, que indica si un vuelo está completo, podemos combinarlos para crear nuevos predicados, como el que nos indica si existen vuelos a un destino dado en una fecha dada,

```
vueloDestino(destino).and(vueloFecha(fecha));
```

o el que nos indica si existen vuelos a un destino dado con plazas libres:

```
vueloDestino(destino).and(vueloCompleto().negate());
```

2.7 allMatch(Predicate)

El método `allMatch` permite realizar el tratamiento 'Para todo'. `allMatch` se aplica a un stream y devuelve un valor true si todos los elementos del stream cumplen una determinada condición (es decir, un predicado) y false si hay al menos un elemento que no lo cumple.

Con este método podríamos responder, por ejemplo, a la pregunta ¿Todos los vuelos que salen en una fecha dada están completos? Lo haríamos así:

```
public Boolean todosVuelosFechaCompletos(LocalDate fecha) {
	return getVuelos().stream()
		.filter(x -> x.getFechaSalida().equals(fecha))
		.allMatch(x -> x.getNumPlazas().equals(x.getNumPasajeros()));
}
```

O bien, utilizando predicados,

```
public Boolean todosVuelosFechaCompletos(LocalDate fecha) {
	Predicate<Vuelo> vueloFecha = x -> x.getFechaSalida().equals(fecha);
	Predicate<Vuelo> vueloCompleto =
		x -> x.getNumPlazas().equals(x.getNumPasajeros());
	return getVuelos().stream().filter(vueloFecha).allMatch(vueloCompleto);
}
```

2.8 anyMatch(Predicate)

El método `anyMatch` permite realizar el tratamiento 'Existe'. `anyMatch` se aplica a un stream y devuelve un valor `true` si existe al menos un elemento del stream que cumpla una determinada condición (es decir, un predicado) y `false` si ningún elemento del stream la cumple.

Con este método podríamos responder, por ejemplo, a la pregunta ¿Existe algún vuelo con una fecha y un destino determinados? Lo haríamos así:

```
public Boolean existeVueloFechaYDestino(LocalDate fecha, String destino) {
	return getVuelos().stream().anyMatch(x -> x.getDestino().equals(destino)
		&& x.getFechaSalida().equals(fecha));
}
```

O bien, utilizando predicados,

```
public Boolean existeVueloFechaYDestino(LocalDate fecha, String destino) {
	Predicate<Vuelo> vueloFecha = x -> x.getFechaSalida().equals(fecha);
	Predicate<Vuelo> vueloDestino = x -> x.getDestino().equals(destino);
	return getVuelos().stream().anyMatch(vueloFecha.and(vueloDestino));
}
```

Aquí hemos usado el método `and` de la interfaz `Predicate`, que permite combinar dos predicados, de forma que deben cumplirse los dos para que el predicado resultante se cumpla.

2.9 min(Comparator)

El método `min` devuelve el elemento mínimo de un stream de acuerdo con el orden definido por un comparador, que recibe como parámetro. Por ejemplo, para obtener el vuelo con un destino dado que sale más pronto, haríamos lo siguiente:

```
public Vuelo primerVueloDestino(String destino) {
	return getVuelos().stream().filter(x->x.getDestino().equals(destino))
		.min(Comparator.comparing(Vuelo::getFechaSalida))
		.get();
}
```

Hay que tener en cuenta que podría no existir un mínimo, algo que sucedería en el caso de que el stream estuviese vacío, como si en el ejemplo no hubiese ningún vuelo con el destino indicado. Por eso el método `min`, al igual que otros métodos, devuelve un objeto de un tipo **Optional<T>**. A este objeto le aplicamos el método `get`, que devuelve el valor del objeto si éste existe, y en caso contrario lanza la excepción `NoSuchElementException`.

También podemos aplicar a los objetos de tipo `Optional` el método `isPresent`, que devuelve `true` si el objeto contiene un valor. De esta forma podemos saber si el valor existe y evitar el lanzamiento de la excepción.

2.10 max(Comparator)

El método `max` devuelve el elemento máximo de un stream de acuerdo con el orden definido por un comparador, que recibe como parámetro. Por ejemplo, para obtener el vuelo con mayor ocupación que sale en una fecha dada, haríamos lo siguiente:

```
public Vuelo vueloMayorOcupacionFecha(LocalDate fecha) {
	return getVuelos().stream()
		.filter(x -> x.getFechaSalida().equals(fecha))
		.max(Comparator.comparing(Vuelo::getNumPasajeros))
		.get();
}
```

2.11 sorted(Comparator)

El método `sorted` permite, como su nombre indica, ordenar un stream. Para indicar el orden le pasamos como parámetro un comparador. Por ejemplo, supongamos que queremos ordenar los vuelos según su fecha de salida. Lo haríamos de la siguiente forma:

```
getVuelos().stream()
	.sorted(Comparator.comparing(Vuelo::getFechaSalida));
```

O bien, si los queremos ordenar por fecha de salida, y a igualdad de ésta por destino,

```
getVuelos().stream().sorted(Comparator.comparing(Vuelo::getFechaSalida)
	.thenComparing(Comparator.comparing(Vuelo::getDestino)));
```

Finalmente, si no pasamos ningún parámetro, se ordenan los elementos según su orden natural:

```
getVuelos().stream().sorted();
```

2.12 La interfaz Function

Sea la siguiente expresión lambda:

```
x -> x.getFechaSalida()
```

Esta expresión la hemos utilizado para obtener la fecha de salida de un vuelo, y equivale a una función que a partir de un vuelo obtiene su fecha de salida. Es pues un tipo funcional. En Java 8 existe una interfaz funcional, la interfaz **Function**, que permite definir objetos que representan funciones, de la misma forma que la interfaz Predicate permitía definir objetos que representaban condiciones.

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

El tipo asociado a esta interfaz se puede representar de la forma **T -> R**, ya que una función toma un objeto de un tipo T y devuelve otro objeto de un tipo R.

Recordemos un ejemplo donde hemos usado esta expresión.

```
Comparator<Vuelo> comparadorVueloFechaSalida =
	Comparator.comparing(x -> x.getFechaSalida());
Collections.sort(vuelos, comparadorVueloFechaSalida);
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

Usando la interfaz `Function`, podemos reescribir este código de la siguiente forma:

```
Function<Vuelo, LocalDate> funcionVueloFecha = x -> x.getFechaSalida();
Collections.sort(vuelos, Comparator.comparing(funcionVueloFecha));
System.out.println("Vuelos por orden de fecha de salida: " + vuelos);
```

Como podemos ver, se define el objeto `funcionVueloFecha` como Function<Vuelo, LocalDate>, es decir, una función que toma un objeto de tipo vuelo y devuelve un objeto de tipo `LocalDate` que es la fecha de salida del vuelo. Este objeto function se pasa como parámetro al método `comparing` para indicar que la fecha de salida es la propiedad que se debe utilizar para comparar los vuelos.

La función también se puede definir en este caso de la forma siguiente:

```
Function<Vuelo, LocalDate> funcionVueloFecha = Vuelo::getFechaSalida;
```

Veamos otro ejemplo más. En este caso queremos ordenar los vuelos por su porcentaje de ocupación:

```
Function<Vuelo, Double> funcionVueloOcupacion =
	x -> 100. * x.getNumPasajeros() / x.getNumPlazas();
Collections.sort(vuelos, Comparator.comparing(funcionVueloOcupacion));
System.out.println("Vuelos por orden de % de ocupación: " + vuelos);
```

Los métodos `compose` y `andThen` de la interfaz `Function` permiten aplicar dos funciones una tras otra. Así, `f1.andThen(f2)` construye una nueva función que aplica sucesivamente `f1` y luego `f2` a los resultados obtenidos. Por su parte, `f1.compose(f2)` construye una nueva función que aplica sucesivamente `f2` y luego `f1` a los resultados obtenidos.

Por ejemplo, dadas las funciones

```
Function<Vuelo, LocalDate> funcionVueloFecha = Vuelo::getFechaSalida;
Function<LocalDate, Integer> funcionFechaDia = LocalDate::getDayOfMonth;
```

Podemos obtener la siguiente función por composición de ambas:

```
Function<Vuelo, Integer> funcionVueloDiaSalida =
      funcionVueloFecha.andThen(funcionFechaDia);
```

Existen varios métodos de stream que reciben como parámetro un objeto de tipo `Function`. Uno de ellos es el método `map`.

2.13 map(Function)

El método `map` obtiene un stream de elementos de un tipo a partir de un stream de elementos de otro tipo. Para ello recibe como parámetro la función que transforma los objetos de un tipo en otro. Por ejemplo, a partir de un stream de vuelos podríamos obtener un stream con el número de pasajeros de cada vuelo. Fíjese que esto equivale a un `Map<Vuelo, Integer>`, de aquí el nombre de este método.

En realidad existe una familia de métodos `map`, según el tipo de los objetos del stream de salida. Así, existen los métodos `mapToInt`, `mapToLong`, `mapToDouble`.

Veamos como ejemplo un método que obtiene el número total de pasajeros de los vuelos que salen en una fecha determinada.

```
public Integer sumaPasajerosVuelosFecha(LocalDate fecha) {
      return getVuelos().stream()
            .filter(x -> x.getFechaSalida().equals(fecha))
            .mapToInt(x -> x.getNumPasajeros())
            .sum();
}
```

La llamada a `mapToInt` obtiene un stream de objetos de tipo `Integer` que contiene el número de pasajeros de todos los vuelos que salen en la fecha dada. A este stream le aplicamos el método terminal sum, que suma todos los elementos del stream.

Si en lugar de aplicar el método sum aplicamos el método `average`, obtendremos el número medio de pasajeros de los vuelos.

```
public Double mediaPasajerosVuelosFecha(LocalDate fecha) {
      return getVuelos().stream()
            .filter(x -> x.getFechaSalida().equals(fecha))
            .mapToInt(x -> x.getNumPasajeros())
            .average()
            .getAsDouble();
}
```

En este caso hay que aplicar después el método `getAsDouble`, ya que `average` devuelve un objeto de tipo opcional.

2.14 flatMap(Function)

El método `flatMap` concatena varios streams en un nuevo stream. Concretamente, a partir de un stream original, obtiene un nuevo stream formado por la concatenación de los streams obtenidos al aplicar una función a cada uno de los elementos del stream original. Equivale al tratamiento secuencial *aplana*.

Veámoslo con un ejemplo: queremos obtener una lista con todos los pasajeros de los vuelos que salen en una fecha dada. El código sería el siguiente:

```
public List<Pasajero> pasajerosVuelosFecha(LocalDate fecha) {
	return getVuelos().stream()
		.filter(x -> x.getFechaSalida().equals(fecha))
		.flatMap(x -> x.getPasajeros().stream())
		.collect(Collectors.toList());
}
```

2.15 reduce (T, BiFunction)

En los ejemplos anteriores hemos utilizado los métodos `sum` y `average` para calcular la suma y el valor medio, respectivamente, de los elementos de un stream. Estas dos operaciones son sendos casos particulares de una operación genérica consistente en calcular un valor a partir de los elementos de un stream. El método que realiza esta operación es el método `reduce`.

Veamos cómo se puede reescribir el método que obtiene el número total de pasajeros de los vuelos que salen en una fecha determinada usando el método reduce.

```
public Integer sumaPasajerosVuelosFecha(LocalDate fecha) {
	return getVuelos().stream()
		.filter(x -> x.getFechaSalida().equals(fecha))
		.mapToInt(x -> x.getNumPasajeros())
		.reduce(0, (x, y) -> x + y);
}
```

El método `reduce` tiene dos parámetros. El segundo de ellos indica la operación que hay que realizar con los elementos del stream, que en este caso es la suma de ambos. El primer parámetro es el elemento neutro de la operación, que es 0 al tratarse de una suma.

El segundo parámetro,

```
(x, y) -> x + y
```

es un tipo funcional, concretamente un objeto de la interfaz funcional **BiFunction**:

```
public interface BiFunction<T, U, R> {
	. . .
}
```

El tipo asociado a esta interfaz se puede representar de la forma **T x U -> R**, ya que una `BiFunction` toma dos objetos de tipos T y U y devuelve otro objeto de un tipo R. Como se puede deducir, es una extensión de la interfaz `Function`. En nuestro caso el tipo funcional es **T x T -> T**, lo cual corresponde a un subtipo de `BiFunction<T, U, R>` que es **BinaryOperator<T>**.

Veamos otro ejemplo de uso del método `reduce`. En este caso, queremos construir una cadena de caracteres con los nombres de los destinos de todos los vuelos que parten en una fecha determinada.

```
public String destinosVuelosFecha(LocalDate fecha) {
	return getVuelos().stream()
		.filter(x -> x.getFechaSalida().equals(fecha))
		.map(Vuelo::getDestino)
		.reduce("", (x, y) -> x + " " + y);
}
```

Como vemos, el stream de vuelos se filtra y se transforma en un stream de `String` con los destinos de los vuelos. A este stream le aplicamos el método `reduce`. En este caso la operación que se realiza con los elementos es la concatenación, cuyo elemento neutro es la cadena vacía.

Este método tiene un "problema", y es que la cadena de salida puede contener destinos repetidos. Para evitar esto podemos eliminar estos elementos. Para ello usamos el método de stream `distinct`, que elimina los elementos repetidos de un stream. El método quedaría entonces así:

```
public String destinosVuelosFecha(LocalDate fecha) {
	return getVuelos().stream()
		.filter(x -> x.getFechaSalida().equals(fecha))
		.map(Vuelo::getDestino)
		.distinct()
		.reduce("", (x, y) -> x + " " + y);
}
```

2.16 La interfaz Consumer

La interfaz funcional `Consumer` es similar a la interfaz `Function`, salvo por el hecho de que no devuelve ningún valor. Su objetivo es realizar una acción sobre el objeto al cual se aplica, como puede ser modificar el valor de una propiedad del mismo.

```
public interface Consumer<T> {
	. . .
}
```

Un ejemplo sería retrasar un día la fecha de llegada de un vuelo:

```
Consumer<Vuelo> retrasaFechaLlegada =
	x -> x.setFechaLlegada(x.getFechaLlegada().plusDays(1));
```

Y otro ejemplo sería cambiar el destino de un vuelo:

```
Consumer<Vuelo> desviaVuelo = x -> x.setDestino(nuevoDestino);
```

Es habitual usar un consumer para mostrar contenido en la pantalla, sustituyendo a la expresión `System.out.println`. Por ejemplo, el siguiente objeto consumer muestra un vuelo en pantalla:

```
Consumer<Vuelo> muestraVuelo = x -> System.out.println(x);
```

2.17 forEach(Consumer)

Los objetos de tipo consumer se utilizan por ejemplo en el método `forEach`. El método `forEach` ejecuta una acción sobre los elementos de un stream. La acción se define mediante un objeto de tipo `Consumer`. Por

ejemplo, supongamos que un temporal de nieve obliga a desviar todos los vuelos que se dirigen a un determinado aeropuerto hacia un aeropuerto alternativo. Lo haríamos de la siguiente forma:

```
public void desviaVuelosDestino(String destino, String nuevoDestino) {
	getVuelos().stream()
	.filter(x -> x.getDestino().equals(destino))
	.forEach(x -> x.setDestino(nuevoDestino));
}
```

La expresión

```
X -> x.setDestino(nuevoDestino));
```

representa la operación a realizar con los objetos de tipo vuelo, consistente en cambiar su destino al nuevo destino. Como vemos, el método no devuelve nada, sino que modifica los objetos del stream, en este caso los vuelos con el destino afectado por el temporal.

Un uso habitual de `forEach` es mostrar en pantalla los elementos de un stream. Por ejemplo, para mostrar en pantalla los destinos de los vuelos que parten en una fecha dada haríamos lo siguiente:

```
public void muestraDestinosVuelosFecha(LocalDate fecha) {
	getVuelos().stream()
		.filter(x -> x.getFechaSalida().equals(fecha))
		.forEach(x -> System.out.println(x.getDestino()));
}
```

2.18 Método collect. La interfaz Collector

El método `collect` proporciona un mecanismo para convertir un stream en otro tipo de dato. Típicamente se utiliza para generar una colección, como un `List` o un `Set`, o un `Map`. Supongamos por ejemplo que queremos obtener una lista con el número de pasajeros de cada uno de los vuelos que parten en una fecha determinada. Aplicamos en primer lugar un filtro con la fecha de salida. Después aplicamos el método `map` para convertir el stream de vuelos en otro stream con el número de pasajeros de cada vuelo y, por último, convertimos este stream en una lista. Para ello utilizamos el método `collect`. Se haría de la siguiente forma:

```
public List<Integer> numPasajerosVuelosFecha(LocalDate fecha) {
	return getVuelos().stream()
		.filter(x -> x.getFechaSalida().equals(fecha))
		.map(x -> x.getNumPasajeros())
		.collect(Collectors.toList());
}
```

El método `collect` toma como parámetro un objeto del tipo **Collector** de Java. Los objetos de tipo `Collector`, también llamados recolectores, indican cómo transformar el stream. En este caso, usamos el recolector `toList` definido en la clase `Collectors`, que transforma el stream en una lista. Otros recolectores son `toSet` y `toMap`.

Los métodos `toList` y `toSet` son casos particulares del método `toCollection`, que construye una colección a partir de un stream. El método anterior sería equivalente a:

```
public List<Integer> numPasajerosVuelosFecha(LocalDate fecha) {
	return getVuelos().stream()
		.filter(x -> x.getFechaSalida().equals(fecha))
		.map(x -> x.getNumPasajeros())
		.collect(Collectors.toCollection(LinkedList::new));
}
```

El método `toMap` transforma un stream en un `Map`. Este método tiene unas cuantas variantes. La más básica es la que tiene como parámetros dos funciones que determinan cómo se construyen los conjuntos de claves y de valores del `Map`. Por ejemplo, supongamos que queremos obtener un `Map` que relacione los códigos de los vuelos con sus fechas de salida. Sería así:

```
public Map<String, LocalDate> fechaSalidaPorCodigo() {
	return getVuelos().stream()
		.collect(Collectors.toMap(
			x -> x.getCodigo(),
			x -> x.getFechaSalida()));
}
```

En este caso, si los códigos de los vuelos son únicos, no se producen colisiones debido a claves repetidas en el Map. Sin embargo, supongamos ahora que queremos obtener un `Map` que relacione los destinos de los vuelos con sus fechas de salida. Siguiendo el ejemplo anterior, lo escribiríamos de la siguiente manera:

```
public Map<String, LocalDate> fechaSalidaPorDestino() {
	return getVuelos().stream()
		.collect(Collectors.toMap(
			x -> x.getDestino(),
			x -> x.getFechaSalida()));
}
```

En este caso hay más de un vuelo con el mismo destino, por lo cual se van a producir colisiones al insertar las claves en el `Map`, y el método va a lanzar una excepción. Para evitarlo hemos de indicar qué debe hacerse con las claves repetidas. En nuestro caso, vamos a almacenar en el `Map` el primer vuelo que sale hacia cada destino. Además, vamos a hacer que este `Map` esté ordenado por destino. Para ello usamos otra versión del método `toMap`, y tendremos lo siguiente:

```
public SortedMap<String, LocalDate> primeraFechaSalidaPorDestino() {
	return getVuelos().stream()
		.collect(Collectors.toMap(
			x -> x.getDestino(),
			x -> x.getFechaSalida(),
			(f1, f2) -> fechaAnterior(f1, f2),
			TreeMap::new));
}

private LocalDate fechaAnterior(LocalDate f1, LocalDate f2) {
	LocalDate res;
	if (f1.compareTo(f2) <= 0) {
		res = f1;
	} else {
		res = f2;
	}
	return res;
}
```

El método `fechaAnterior` es el que determina que en caso de colisión se almacene en el `Map` el vuelo que tiene una fecha de salida anterior. El último parámetro es una referencia a método, en este caso al constructor de la clase `TreeMap`, y es el que hace que el valor devuelto por el método sea un `SortedMap`.

Veamos otro ejemplo. Queremos construir ahora un `Map` que relacione los destinos de los vuelos con el número de pasajeros que vuelan a cada uno de ellos. Las claves del `Map` serán los destinos, y los valores serán el número total de pasajeros que vuelan a dichos destinos. Si hay dos vuelos con el mismo destino se producirá una colisión, y en este caso lo que haremos será sumar los pasajeros de ambos vuelos. Para ello pasaremos como tercer parámetro al método `toMap` un operador binario que acumule los valores asociados a la misma clave. En este caso, la operación consistirá en sumar dos valores de tipo `Integer`, y para ello usaremos el método `sum` del tipo `Integer`. Quedaría así:

```
	public Map<String, Integer> numeroPasajerosPorDestino() {
		return getVuelos().stream()
			.collect(Collectors.toMap(
				Vuelo::getDestino,
				Vuelo::getNumPasajeros,
				Integer::sum));
}
```

Además de los métodos anteriores, la clase `Collectors` proporciona los métodos `partitioningBy` y `groupingBy` que permiten organizar la información de un stream en un `Map`. Las posibilidades son numerosas ya que `groupingBy` puede recibir distintos parámetros. Veamos los casos básicos.

2.18.1 groupingBy(Function)

Este método crea un `Map` que agrupa los elementos de un stream según el valor resultante de aplicar a los mismos una función determinada. En el caso más simple esta función puede ser el valor de alguna de sus propiedades. Por ejemplo, si queremos agrupar los vuelos según su destino, haremos lo siguiente:

```
	public Map<String, List<Vuelo>> vuelosPorDestino() {
		return getVuelos().stream()
		   .collect(Collectors.groupingBy(Vuelo::getDestino));
	}
```

2.18.2 partitioningBy(Predicate)

Este método es un caso particular del anterior en el que la función es sustituida por un `Predicate`. En este caso, el conjunto de claves del `Map` está formado por los valores `true` y `false`, que son los resultados posibles del predicado. Por ejemplo, queremos clasificar los vuelos en dos grupos: los que están completos y los que no lo están. Lo haríamos así:

```
	public Map<Boolean, List<Vuelo>> completosYConPlazas() {
		return getVuelos().stream()
			.collect(Collectors.partitioningBy(
			x -> x.getNumPasajeros().equals(x.getNumPlazas())));
	}
```

2.19 Otros métodos de Stream

Además de los métodos que hemos visto, existen otros métodos que se pueden aplicar sobre streams, como los siguientes:

- **findAny()**. Método terminal. Devuelve un elemento cualquiera del stream.
- **findFirst()**. Método terminal. Devuelve el primer elemento del stream.
- **limit(Long)**. Método intermedio. Devuelve un stream con el tamaño máximo indicado.
- **noneMatch(Predicate)**. Método terminal. Devuelve true si ningún elemento del stream cumple el predicado.
- **peek(Consumer)**. Método intermedio. Aplica una acción a todos los elementos del stream, devolviendo el stream original. Nótese la diferencia respecto a forEach, que es un método terminal y aplica la acción al stream sin devolver nada.

3. Lectura y escritura de ficheros

Java dispone de una clase `Files` con un conjunto de métodos estáticos para operar con ficheros y directorios. Veamos cómo realizar las operaciones básicas de lectura y escritura utilizando esta clase.

3.1 Lectura de un fichero

Para leer un fichero se utiliza el método `lines`:

```
static Stream<String> lines(Path path);
```

Este método lee las líneas del fichero de texto cuyo nombre recibe como parámetro y las almacena en un stream de objetos de tipo `String`.

Como ejemplo, supongamos que tenemos un fichero de texto `vuelos.txt` con el siguiente contenido:

```
IBE5051, Sevilla, Roma, 15/5/2015, 15/5/2015, 180
IBE5244, Sevilla, Roma, 16/5/2015, 16/5/2015, 180
VLG1212, Sevilla, Amsterdam, 15/5/2015, 15/5/2015, 180
VLG9786, Sevilla, San Francisco, 15/5/2015, 16/5/2015, 340
RYA0505, Jerez, Londres, 19/5/2015, 19/5/2015, 180
```

Para crear a partir de este fichero la lista de vuelos de un aeropuerto, haríamos lo siguiente:

```
public void leeVuelos(String nombreFichero) {
	try {
		vuelos = Files.lines(Paths.get(nombreFichero))
			.map(s -> new VueloImpl(s))
			.collect(Collectors.toList());
	} catch (IOException e) {
		System.out.println("Error en la lectura del fichero");
	}
}
```

Una posible llamada a este método sería

```
a.leeVuelos("res/vuelos.txt");
System.out.println("Vuelos del fichero: " + a.getVuelos());
```

El método `lines` lee las líneas del fichero de texto y construye con ellas un stream de `String`. Este stream se transforma en un stream de vuelos invocando al método `map` y pasándole como parámetro una función que devuelve un objeto de tipo `Vuelo` a partir de su representación como cadena. Finalmente, este stream se transforma en una lista.

Podemos escribir un método genérico que lea un fichero de texto cuyas líneas representan objetos y construya una lista con dichos objetos. Para ello, el tipo debe contar con un constructor a partir de cadena. El método sería el siguiente:

```
public static <T> List<T> leeFichero(String nombreFichero,
            Function<String,T> funcion_deString_aT) {
      List<T> res = null;
      try {
            res = Files.lines(Paths.get(nombreFichero))
                  .map(funcion_deString_aT)
                  .collect(Collectors.toList());
      } catch (IOException e) {
            System.out
                  .println("Error en lectura del fichero: "+nombreFichero);
      }
      return res;
}
```

En el ejemplo anterior, la invocación al método para leer el fichero de vuelos sería la siguiente:

```
List<Vuelo> vuelos = Utiles.leeFichero("res/vuelos.txt", s -> new VueloImpl(s));
```

3.2 Escritura en un fichero

Para la escritura en ficheros utilizaremos el método `write` de la clase `Files`. Este método escribe los elementos de un iterable de `String` en un fichero de texto, a razón de uno por línea. Por ejemplo, vamos a escribir en un fichero de texto los destinos de todos los vuelos que parten en una fecha dada.

```
public void escribeDestinosVuelosFecha(LocalDate fecha, String nombreFichero) {
      List<String> s = getVuelos().stream()
            .filter(x -> x.getFechaSalida().equals(fecha))
            .map(Vuelo::getDestino)
            .collect(Collectors.toList());
      try {
            Files.write(Paths.get(nombreFichero), s);
      } catch (IOException e) {
            System.out.println("Error en la escritura del fichero");
      }
}
```

Una posible llamada a este método sería

```
a.escribeDestinosVuelosFecha(LocalDate.of(2015, 5, 15), "res/destinosVuelos.txt");
```

El método `write` recibe como parámetros el nombre del fichero y el iterable de `String` con los elementos que se van a escribir, en el ejemplo un `List<String>` con los destinos de los vuelos.