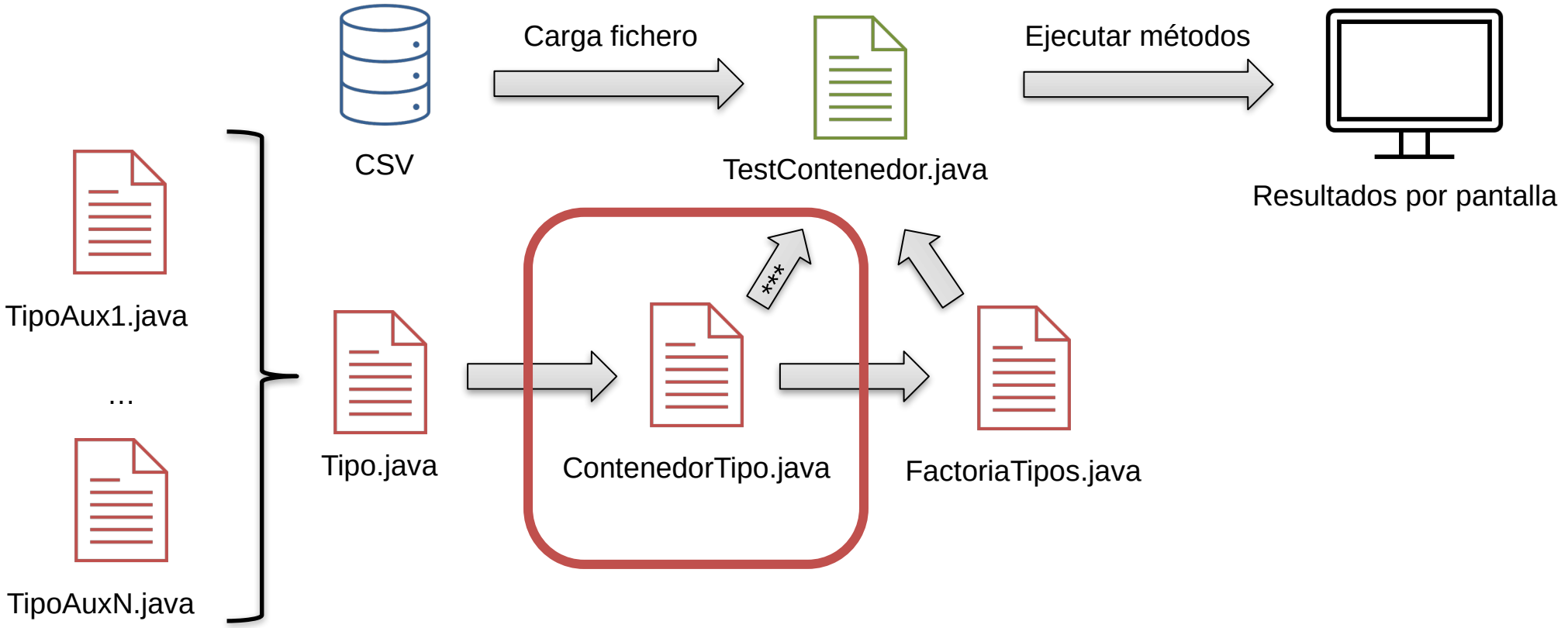
FUNDAMENTOS DE PROGRAMACIÓN

7 – Streams

¿Qué se ve en este tema?



¿Qué se ve en este tema?

Aeropuerto.java

```
public class Aeropuerto {
        private String nombre;
        private List<Vuelo> vuelos;
        public Aeropuerto(String nombre, List<Vuelo> vuelos) {
                this.nombre = nombre;
                this.vuelos = vuelos;
        }
...
         // Total de vuelos de dada una fecha de salida
   public Long numeroVuelosSalidaFecha(LocalDate fecha) {
   Long num = 0l;
   for (Vuelo v : getVuelos()) {
           if (v.getFechaSalida().equals(fecha)) {
                   num++;
           }
   }
   return num;
   }
}
```

¿Qué se ve en este tema?

Aeropuerto.java

```
public class Aeropuerto {
        private String nombre;
        private List<Vuelo> vuelos;

        public Aeropuerto(String nombre, List<Vuelo> vuelos) {
                this.nombre = nombre;
                this.vuelos = vuelos;
        }
...
         // Total de vuelos de dada una fecha de salida
   public Long numeroVuelosSalidaFechaStream(LocalDate fecha) {
           return getVuelos().stream()
                                                    .filter(x ->
   x.getFechaSalida().equals(fecha))
                                                    .count();
   }
}
```

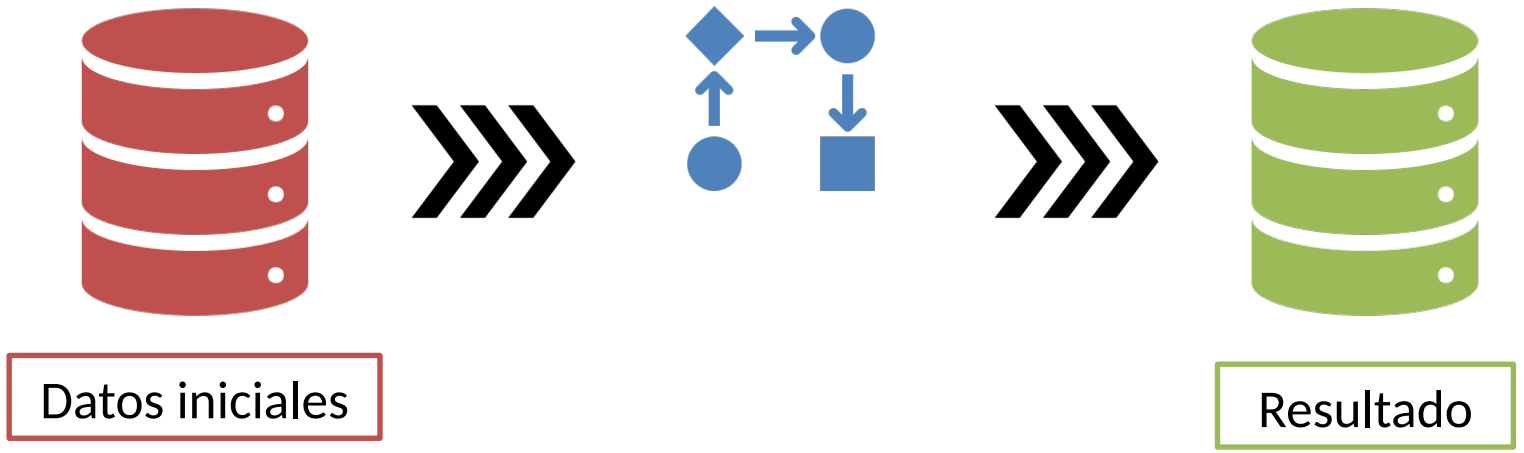
ÍNDICE

ÍNDICE

1. Introducción a Stream

- 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.
- 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.

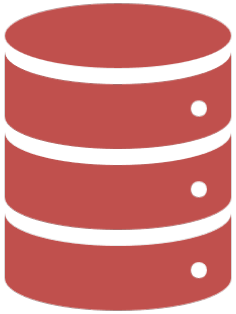
1. Introducción a Stream



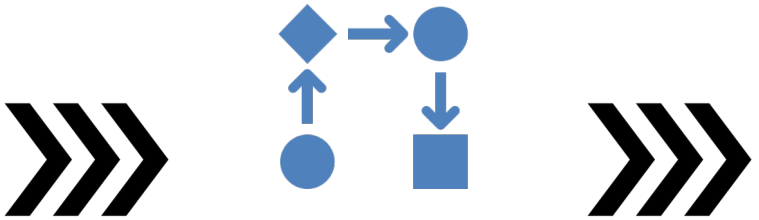
1. Introducción a Stream

Vuelo

```
LocalDate fechaSalida
LocalTime horaSalida
String destino
```



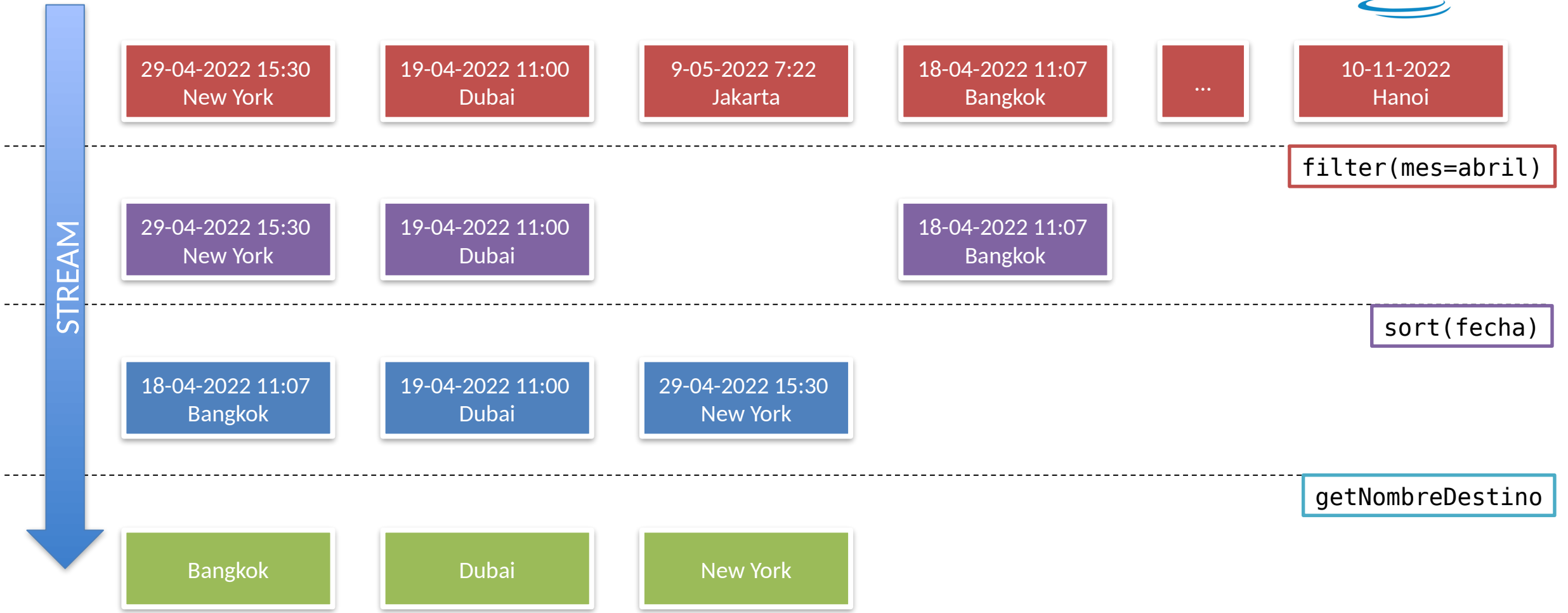
Datos iniciales



Resultado

Lista de los nombres de **destinos** del mes de **abril** ordenados por **fecha de salida**

1. Introducción a Stream



1. Introducción a Stream
Primer ejemplo


```
//Total de vuelos de dada una fecha de salida
public Integer numeroVuelosSalidaFecha(LocalDate fecha) {
   Integer num = 0;
   for (Vuelo v : getVuelos()) {
      if (v.getFechaSalida().equals(fecha)) {
         num++;
      }
   }
   return num;
}
```

```
public Long numeroVuelosSalidaFecha(LocalDate fecha) {
    return getVuelos().stream()
       .filter(x -> x.getFechaSalida().equals(fecha))
       .count();
     }
```

1. Introducción a Stream
Segundo ejemplo


```
        //Lista de los nombres de destinos de una fecha
data
    public List<String> listaDestOrdFecha(LocalDate fecha)
{
        List<Vuelo> vuelos = new ArrayList<Vuelo>();
        for (Vuelo v : getVuelos()) {
            if (v.getFecha().equals(fecha)) {
                vuelos.add(v);
            }
        }

        vuelos.sort(Comparator.comparing(Vuelo::getFecha));

        List<String> resultado = new ArrayList<String>();
        for(Vuelo v: vuelos){
            resultado.add(v.getDestino())
        }
        return resultado;
    }
```

```
public List<String> listaDestOrdFecha(LocalDate
fecha) {
   return getVuelos().stream()
      .filter(x -> x.getFecha().equals(fecha))
      .sorted(Comparator.comparing(Vuelo::getFecha)
)
      .map(Vuelo::getDestino)
      .collect(Collectors.toList());
}
```

ÍNDICE

2. Tipo Stream

- Según hemos visto, un Stream es un agregado de elementos
- En Java 8 existe el tipo Stream<T> para representar estos agregados. Este tipo soporta operaciones secuenciales: 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:

```
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);
```

ÍNDICE

3. Operaciones sobre Streams

- Las operaciones sobre Streams pueden ser de dos tipos:
  - **Intermedias:** producen un Stream a partir de otro.
  - **Terminales**: producen un objeto que **no** es de tipo Stream.

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

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

ÍNDICE

4. Interfaz Predicate
Boolean allMatch(Predicate)

- **allMatch** se aplica a un Stream y devuelve un valor *true* si todos los elementos del Stream cumplen una determinada condición y *false* si hay al menos un elemento que no lo cumple.

¿Todos los vuelos que salen en una fecha dada están completos?

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

4. Interfaz Predicate
Boolean anyMatch(Predicate)

- **anyMatch** se aplica a un Stream y devuelve un valor *true* si existe al menos un elemento del Stream que cumpla una determinada condición y *false* si ningún elemento del Stream la cumple.

¿Existe algún vuelo con una fecha y un destino determinados?

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

ÍNDICE

5. Interfaz Comparator
Optional 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.

Obtener el vuelo con un destino dado que sale más pronto

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

5. Interfaz Comparator
Optional min(Comparator)

- Podría no existir un mínimo (Stream estuviese vacío).
- El método min 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.
- Para evitar el lanzamiento de la excepción, utilizamos el método **orElse**.
  - Este método tiene como parámetro el valor que queremos devolver en el caso de que no se encuentre un mínimo.

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

5. Interfaz Comparator
Optional min(Comparator)

- También podemos obtener directamente un objeto de tipo Optional y aplicar a este objeto el método isPresent, que devuelve true si el objeto contiene un valor. De esta forma podemos saber si el valor existe y en caso contrario especificar el valor de salida alternativo:

```
public Vuelo primerVueloDestino(String destino) {
    Vuelo v = null;
    Optional op = getVuelos().stream()
            .filter(x -> x.getDestino().equals(destino))
            .min(Comparator.comparing(Vuelo::getFechaSalida));
    if (op.isPresent()) {
        v = op.get();
    }
    return v;
}
```

5. Interfaz Comparator
Optional 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.

Obtener el vuelo con mayor ocupación que sale en una fecha dada

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

5. Interfaz Comparator
Stream sorted(Comparator)

- El método **sorted** permite ordenar un Stream. Para indicar el orden le pasamos como parámetro un comparador.

Ordenar los vuelos según su fecha de salida y destino

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

 .thenComparing(Comparator.comparing(Vuelo::getDestino)));
```

- Si no pasamos ningún parámetro, se ordenan los elementos según su orden natural:

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

ÍNDICE

6. 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.
- **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.**

6. Interfaz Function


```
        Comparator<Vuelo> comparadorVueloFechaSalida = Comparator.comparing(x ->
x.getFechaSalida());
        Collections.sort(vuelos, comparadorVueloFechaSalida);
-------------------------------------------------------------
        Function<Vuelo, LocalDate> funcionVueloFecha = x -> x.getFechaSalida();
        Collections.sort(vuelos, Comparator.comparing(funcionVueloFecha));
```

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

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

- Podemos obtener la siguiente función por composición de ambas:

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

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

6. Interfaz Function
Stream 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.

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()) // .mapToInt(Vuelo::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.

6. Interfaz Function
Stream map(Function)

- 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 Optional, al igual que hacía el método min. El tratamiento de este objeto es similar al que se vio entonces, con la diferencia de que al ser un valor real se usa en este caso el método getAsDouble en lugar del método get.

6. Interfaz Function
Stream flatMap(Function)

- 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.

Obtener una lista con todos los pasajeros de los vuelos que salen en una fecha dada

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

ÍNDICE

7. 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.
- Retrasar un día la fecha de llegada de un vuelo; o cambiar el destino de un vuelo

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

7. Interfaz Consumer
Stream 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.

Supongamos que un temporal de nieve obliga a desviar todos los vuelos que se dirigen a un determinado aeropuerto hacia un aeropuerto alternativo

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

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

ÍNDICE

8. Interfaz Collector
Object <E> collect(Collector)

- 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.

Obtener una lista con el número de pasajeros de cada uno de los vuelos que parten en una fecha determinada:

1. Aplicamos en primer lugar un filtro con la fecha de salida.
2. Después aplicamos el método map para convertir el Stream de vuelos en otro Stream con el número de pasajeros de cada vuelo.
3. Por último, convertimos este Stream en una lista. Para ello utilizamos el método collect.

```
public List<Integer> numPasajerosVuelosFecha(LocalDate fecha) {
    return getVuelos().stream()     // 0. Stream<Vuelo>
            .filter(x -> x.getFechaSalida().equals(fecha))   // 1. Stream<Vuelo>
            .map(x -> x.getNumPasajeros())          // 2. Stream<Integer>
            .collect(Collectors.toList());          // 3. List<Integer>
}
```

- 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**.

8. Interfaz Collector
groupingBy(Function)

- La clase Collectors proporciona también los métodos **partitioningBy** y **groupingBy**, que permiten organizar la información de un Stream en un Map.
- 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.

Agrupar los vuelos según su destino

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

8. Interfaz Collector
groupingBy(Function, Collectors)

- El método **groupingBy** admite un segundo parámetro además de la función de agrupación. Este segundo parámetro es un objeto de tipo Collector y representa un acumulador que modifica la forma en la que se agrupan los elementos del Stream.

Obtener un Map que relacione los destinos con el número de vuelos que se dirigen a cada destino

- Hemos de contar cuántos vuelos tienen cada destino. Para ello usamos el acumulador **Collectors.counting()**

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

8. Interfaz Collector
partitionBy(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.

Clasificar los vuelos en dos grupos: los que están completos y los que no lo están

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

8. Interfaz Collector
groupingBy(Function, Collectors)

- **Collectors.summingInt(F)** suma los valores devueltos por la función que recibe como parámetro.

Calcular el número total de pasajeros que se dirigen a cada destino

```
public Map<String, Integer> numeroTotalPasajerosPorDestino() {
    return getVuelos().stream()
            .collect(Collectors.groupingBy(Vuelo::getDestino,
                    Collectors.summingInt(Vuelo::getNumPasajeros)));
}
```

- **Collectors.averagingInt(F)** podemos calcular valores medios.

Media de pasajeros que se dirigen a cada destino:

```
public Map<String, Double> numeroMedioPasajerosPorDestino() {
    return getVuelos().stream()
            .collect(Collectors.groupingBy(Vuelo::getDestino,
                    Collectors.averagingInt(Vuelo::getNumPasajeros)));
}
```

8. Interfaz Collector
groupingBy(Function, Collectors)

- **Collectors.mapping(Function, Collector)** se usa como argumento para construir el valor del mapa. Devuelve los valores marcados por el Function en la forma en la que se indica en el collector.

Obtener los diferentes modelos de avión por compañía

```
public Map<String, Set<String>> getModelosPorCompañia() {
    return vuelos.stream()
            .collect(Collectors.groupingBy(
                    Vuelo::getCompania, Collectors.mapping(
                            Vuelo::getModAvion, Collectors.toSet())));
}
```

8. Interfaz Collector
groupingBy(Function, Collectors)

- **collectingAndThen(Collector, Function)** realizar alguna acción sobre la Collection (List, Map, etc) que queramos sacar del stream antes de devolverla.

Obtener el vuelo con más minutos de retraso de cada compañía

{"Ryanair" = [Vuelo17], "Iberia" = [Vuelo712], "United" = [Vuelo123],....}

```
public Map<String, Vuelo> getVueloMasRetrasoPorCompañia() {
    return vuelos.stream()
            .filter(v -> v.getMinutosDiferencia() < 0)
            .collect(Collectors.groupingBy(
                    Vuelo::getCompañia, Collectors.collectingAndThen(
                            Collectors.minBy(Comparator.comparing(Vuelo::getMinutosDiferencia)),
                            o -> o.get()))
            );
}
```

8. Interfaz Collector
groupingBy(Function, Collectors)

- **La Combi Completa**

Obtener el número de destinos diferentes por mes

{Enero=4,Febrero=71, Marzo=41,...}

```
    public Map<Month, Integer> getNumeroDestinosDiferentesPorMes() {
        return vuelos.stream()
                .collect(Collectors.groupingBy(v ->
v.getFecha().getMonth(),
                        Collectors.mapping(Vuelo::getCiudad,
                                Collectors.collectingAndThen(
                                Collectors.toSet(), s -> s.size()))));
    }
```

8. Interfaz Collector
groupingBy(Function, Collectors) + max

Obtener el destino con más vuelos

```
public String getDestinoConMasVuelos() {
    Map<String, Long> mapa = vuelos.stream()
      .collect(Collectors.groupingBy(Vuelo::getDestino,
      Collectors.counting()));
      //Map=>{“Madrid”:4, “Barcelona”:9, “Albacete”: 2, … }
      //entrySet()=>[(“Madrid”,4), (“Barcelona”,9), (“Albacete”, 2), … ]

      return mapa.entrySet()     // Set<Entry<String, Long>>
      .stream()    //Stream<Entry<String, Long>>
        .max(Comparator.comparing(Entry::getValue)) //Optional<Entry<String, Long>>
        .orElse(null)//Entry<String, Long>
        .getKey();   //String
}
```

8. Interfaz Collector
groupingBy(Function, Supplier, Collectors)

Obtener un SortedMap con la lista de vuelos por destino

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

8. Interfaz Collector
groupingBy(Function, Supplier, Collectors)

Obtener un SortedMap con los n vuelos con más pasajeros por destino

```
  public SortedMap<String, List<Vuelo>> getNVuelosOrdenadosConMásPasajerosPorDestino(Integer n) {
      return vuelos.stream()
              .collect(Collectors.groupingBy(Vuelo::getDestino,
                      TreeMap::new,
                      Collectors.collectingAndThen(Collectors.toList(),
    x -> seleccionaVuelos(x, n))));
  }
private static List<Vuelo> seleccionaVuelos(List<Vuelo> vuelos, int n) {
    Comparator<Vuelo> cmp = Comparator.comparing(Vuelo::getNumPasajeros).reversed();
    return vuelos.stream()
            .sorted(cmp)
            .limit(n)
            .collect(Collectors.toList());
}
```

ÍNDICE

9. Otros métodos

- **Optional findAny().** Método terminal. Devuelve un objeto Optional que representa un elemento cualquiera del Stream.
- **Optional findFirst().** Método terminal. Devuelve un objeto Optional que representa el primer elemento del Stream.
- **Boolean noneMatch(Predicate).** Método terminal. Devuelve true si ningún elemento del Stream cumple el predicado.
- **Stream limit(Long).** Método intermedio. Devuelve un Stream con el tamaño máximo indicado.
- **Stream 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.